١. وهي مكتبة لجافاسكربت مسؤولة عن واجهات المستخدم user interfaces

٢. تم تصميمها من قبل facebook on 2011

٣. ماذا يمكن البناء فيها

a. اضافة أدوات لموقع متعدد الصفحات mpas

b. موقع من صفحة واحدة ويكون معتمد كليا على الجافاسكربت spas

c. موبايل ابلكيشن من خلال رياكت نيتف

٤. يعمل رياكت من خلال

a. المكونات components من خلال بناء مكونات ثم جمعها في مكون واحد

b. jsx وهي تقنية لبناء تمبلت للموقع حيث يتم كتابة html/css داخل الجافاسكربت وتقوم بتحويلها من خلال bable

c. virtual dom وهو دوم خاص بالرياكت و يعتمد مبدأ virtual dom على تخزين التطبيق في الرام ويتم المقارنة الأجزاء مع Dom الأصلي حيث أن أي تغيير في الأجزاء يتم تغيير الجزء القديم إلى الجديد فورا

d. يأخذ قوة الجافا سكربت الكلية لانه يكتب داخلها فهي عالية الأداء

٥. Declarative react : وهي مكتبة جاهزة للأستخدام توفر علينا عناء بناء نظام أدارة واجهة المستخدم

٦. Imperative : و معناها يجب علينا بناء نظام إدارة واجهة المستخدم من الصفر بكل تفاصيلها لذلك Declarative react توفر علينا العناء

2

التنصيب

لتنصيب يجب أن نميز أنواع التنصيب في الرياكت :

١. موقع متعدد الصفحات MPA
٢. موقع ويب من صفحة واحدة SPA

١. النوع الأول لموقع متعدد الصفحات MPA :

إذا كان لدينا موقع ويب جاهز أي موقع متعدد الصفحات MPA ونريد إضافة الرياكت أو نريد إضافة لوجك ديناميكي معين لموقع نستخدم هذه الطريقة للمواقع MPA :

١. نقوم بإضافة الوسمين الخاصين بمكتبة الرياكت و الرياكت دوم دوما :

```
<script src="https://unpkg.com/react@17/umd/react.production.min.js" crossorigin></script>
<script src="https://unpkg.com/react-dom@17/umd/react-dom.production.min.js"
crossorigin></script>
```

٢. jsx وهي تقنية خاصة بالقوالب وأضافتها يكون بشكل إختياري وهناك طريقتين لإضافة :

a. الطريقة الأولى نضيف الوسم الخاص CDN
b. <script src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>
c. ثم نضيف لكل سكربت جافاسكربت type="text/babel"
d. هذه الطريقة تقوم بالتحويل القالب الى جافاسكربت أثناء تحميل الموقع فهي بطيئة
e. الطريقة الثانية ننصب node.js
f. ثم نفتح الكوميد لاين ثم ندخل إلى المشروع داخل ملف الجافا سكربت ثم نضيف الوسم
g. npm init –y مع العلم ان اسم ملف المشروع يكون مراعي شروط التسمية ثم نضيف
h. npm install babel-cli@6 babel-preset-react-app@3
i. ثم ننشئ ملف src ونحط داخله ملف الجافا سكربت الذي نعدل عليه دوما ويتحول تلقائيا عن طريق jsx
j. نكتب الأمر في الكوميد لاين لكن نكون داخل ملف الجافاسكربت وخارج ملف src في كل مرة نفتح المشروع
k. npx babel --watch src --out-dir . --presets react-app/prod
l. هذه الطريقة تقوم بالتحويل القالب الى جافاسكربت أثناء برمجة الموقع أي التحويل مسبق فهي ذات سرعة حقيقية

ملاحظة : لتنصيب سيرفر محلي نقوم بتنصيف الباكج npm i -g serve من ثم لتشغيل السرفير نقف عند ملف index.html ونفتح نافذة كومند لاين ونكتب الأمر serve ثم نفتح المتصفح على الرابط http://localhost:3000

٢. النوع الثاني لموقع ويب من صفحة واحدة SPA نستخدم تطبيق (create-react-app) :

التطبيق وحيد الصفحة (Single-page Application) هو التطبيق الذي يُحمِّل صفحة HTML واحدة وكل ملحقاتها الضرورية مثل CSS و JavaScript المطلوبة لكي يعمل التطبيق. لا تتطلّب أيّة تفاعلات مع الصفحة أو الصفحات اللاحقة أي عودة للخادم مرّة أخرى، ممّا يعني عدم إعادة تحميل الصفحة .

يمكن العمل أولايين عن طريق codesanebox أو على الحاسب عن طريق

لتنصيب نثبت node.js من الموقع الرئيسي ثم ننشئ ملف المشروع ثم ندخل إلى الملف و نفتح الكومد لاين ونكتب الأمر

```
npx create-react-app appName
npm init react-app appName
yarn create react-app appName
```

هكذا تم إنشاء المشروع ثم ندخل عليه عن طريق كتابة الأمر cd appName ثم نشغله عن طريق كتابة الأمر npm start

ملاحظة : لتصيب yarn نكتب الأمر

```
npm install --global yarn
```

ملاحظة : لتحديث npm وهي باكج تحتوي على بكجات كثيرة لجافاسكريبت لتحديثها لأخر نسخة من الإصدار نكتب الأمر

```
npm install npm@latest -g
```

ملاحظة : نصب vs code ثم نضيف عليه الإضافات التالية / auto close tag / auto rename tag / bable es6 / bracket pair colorizer / ES7+ React/Redux/React-Native snippets / Material Icon Theme / IntelliSense for CSS class names in HTML / ESLint / (github) Settings Sync / Path Autocomplete / Path Intellisense

3 أنواع أدوات البناء

١. اذا تريد إنشاء تطبيق ذو صفحة واحدة استخدم بيئة create react app (SPA)

٢. إذا كنت تريد أنشاء تطبيق متعددة الصفحات (MPA) استخدم وسوم الريأكت مع بابل

٣. إن كنت تبني موقعًا يصيَّر من طرف الخادم (server-rendered website) مع Node.js, جرب استعمال Next.js .

٤. إن كنت تبني موقعًا ثابتًا يركز على المحتوى (static content-oriented website), جرب استعمال Gatsby .

٥. إن كنت تبني مكتبة لمكون ما أو دمج مع شيفرة أساسية (codebase), جرب استعمال سلاسل أدوات ذات مرونة أكبر.

4 File structure

١. package.json : وهو العامود الفقري للتطبيق و يحتوي على أسماء المكتبات التي يعمل فيها التطبيق حيث عند تثبيت node.js عم طريق الأمر npm install للمشروع يدخل git على داخل Package.json ويحمل المكتبات مع Node.js ليعمل المشروع ويحتوي على العديد من الأشياء مثل أسم المشروع و الأصدار و نوعه هل هو خاص أو عام و يحتوي على dependencies التي تضمن أسماء المكتبات عند تطوير التطبيق ويحتوي على react-scripts التي تقوم بتشغيل التطبيق من git حيث تحتوي على أوامر التشغيل و يحتوي على eslintConfig ليعمل التطبيق على مبدأ الريأكت و ليس تقنيات أخرى

٢. node-moduies : وهي تحتوي على مكتبات node.js من أجل أن يعمل التطبيق ولا نقوم برفعها على github أو أي استضافة بل نقوم بتنصيبها عن طريق الأمر npm install حيث المكتبات الضرورية لعمل التطبيق موجودة في package.json

٣. public : يحتوي على index.html وهو الملف الذي يفتح في المتصفح حيث يحتوي على و يقوم الريأكت بتوجيه كل التطبيق إليه ونضع داخل index.html الروابط الخارجية التي نحتاجها مثل الخطوط إلى أخره وعندما ننتهي من بناء التطبيق نقوم ببناء التطبيق من خلال الأمر npm build حيث يقوم node-moduies بسحب كل src و وضعها في public وينشئ ملف جديد أسمه build وهو التطبيق الجاهز لرفع على أستضافة ويحتوي public على ملف manifest.json ويحتوي على icon / img الخاصة بال index.html

٤. src : و هو الذي يحتوي على كل ملفات تطوير التطبيق حيث يحتوي على الملفات الأساسية التالية :
   a. index.js : وهي التي يوضع داخلها كومبونت App وتقوم بإرساله إلى index.html من خلال reactDom.render() حيث ترسل كل التطبيق إلى و كذلك يستدعي index.css
   b. index.css : وتحتوي على تنسيقات css التطبيق
   c. App.js : وهو الذي يوضع داخله كوبونات التطبيق الذي يتم إستداعه من قبل index.js وهو كذلك يستدعي App.css
   d. App.css : وتحتوي على تنسيقات css ال App

أوامر react-scripts هي :

npm start / npm run build / npm test / npm run eject

5

التعامل مع التمبلت

في ملف src ننشأ ملف اسمه index.js ثم نفتحه و نستدعي مكتبة React و ReactDom المسؤولة عن virtual dom لذلك نكتب

```
import React from “react”;
import ReactDOM from “reacr-dom”;
```

دوما نكتب التامبلت ثم نرسله الى ملف ال html عن طريق ReactDOM.render() حيث هاد الأمر يدخل إلى مكتبة ReactDOM

```
html ->   <div id="root"></div>
js ->   ReactDOM.render(التمبلت , document.getElementById('root'));
```

هناك طريقتين لكتابة التامبلت :

١. عن طريق React.createElement(“tag”, {attributs}, ”contant”)

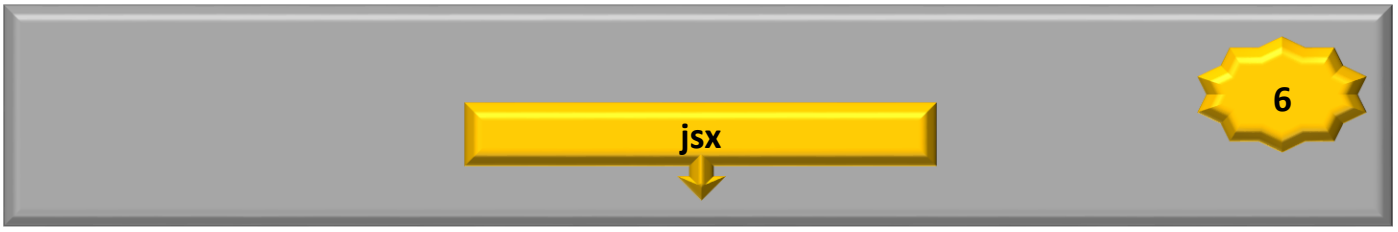
```
React.createElement('nameelement',{classname:' '},'content')

const app = React.createElement('div',{},'hello ammar');
ReactDOM.render(app,document.getElementById('root'));

const app2 = React.createElement('div',{}, React.createElement('div',{},'hello ammar'));
ReactDOM.render(app2,document.getElementById('root'));
```

وهي طريقة صعبة لكتابة التامبلت لهذا نسعيض عنها ب jsx

٢. عن طريق jsx سواء من لينك jsx أو من خلال تنزيل jsx أو من تطبيق SPA

```
1. const app2 = <h1>hello ammar2</h1>;
2. ReactDOM.render(app2,document.getElementById('root2'));
```

ملاحظة : jsx يقوم بتحويل شيفرة القالب إلى الطريقة الأولى وذلك عن طريق bable js وهو يستعمل لكتابة القالب بسهولة كأنه html مع إمكانيات الجافاسكربت



وهي تقنية لكتابة القوالب (التمبلت) في الرياكت حيث تقوم بتحويل القالب إلى جافاسكربت عن طريق bable حيث يحول القوالب ال html hello ammar لدالة React.createElement('div',{},'hello ammar'); وبتالي يعمل على كل المتصفحات :

هي تأخذ قوة html مع امكانيات الجافا سكربت الكلية لذلك فيمكن كتابة داخلها ..... if for

يتم كتابة القالب أما داخل رياكت دوم مباشرة أو داخل متغيير او دالة :

١. كتابة داخل رياكت دوم مباشرة :

```
ReactDOM.render(<h1>ammar</h1>,document.getElementById('root'));
```

٢. كتابة داخل متغيير لعنصر واحد بسطر واحد :

```
const app = <h1>hello word</h1>;
```

٣. كتابة داخل متغيير لعدة عناصر بأكثر من سطر نستخدم () :

```
const app =( <h1>hello word<div>ammar</div></h1> );
const element = (
  <div>
    <h1>Hello!</h1>
    <h2>Good to see you here.</h2>
  </div>
);
```

ملاحظات :

١. إذا كان العنصر فارغ أي لا نريد أن نكتب فيه أبناء نغلقه فورا بوسم إغلاق وهذا الشئ ينطبق على المكونات :

```
const element = <img src=” />;
```

٢. لكتابة أكواد جافا سكربت داخل jsx نكتبها داخل {} :

```
const name = 'Josh Perez';
```

```
const element = (<h1>Hello, {name}</h1>);
ReactDOM.render(element, document.getElementById('root'));
```

٣. نعرف مكونات القالب من خلال const لمنع هجمات الحقن :

```
const formatName = function(user) {return user.toUpperCase();};
function formatName(user) {}; << هذا يتعرض لهجمات الحقن لذلك نستخدم دالة انانموس أو دالة أرو
```

٤. طريقة الاستدعاء الدالة داخل ال jsx نستدعي الدالة داخل {} :

```
const element = (<h1>Hello, {formatName(user)} !</h1>);
```

٥. يمكن استدعاء دالة داخل دالة من خلال :

```
function getGreeting(user) {
  if (user) {
    return <h1>Hello, {formatName(user)}!</h1>;
   } else {
    return <h1>Hello, Stranger.</h1>;
   }
}
ReactDOM.render(getGreeting('ammar'), document.getElementById('root')); >> Hello, AMMAR!
ReactDOM.render(getGreeting(), document.getElementById('root')); >> Hello, Stranger.
```

٦. يمكن استخدام if داخل {} :

نستعمل جملة شرطية من سطر واحد condition ? true : false

```
const name = "ammar"
const element = <p>{ name === "ammar" ? "hello ammar" : "hello there}</p>
```

٧. طريقة استدعاء مصفوفة داخل jsx :

أولا سنعرف مصفوفة في متغيير خارجي

ثانيا عند استدعاء المتغيير داخل {} سوف يعمل ("")join لمصفوفة بشكل تلقائي

```
const name = ["ammar","abdalqader", "qassab"]
const element = <p>{name}</p>//ammarabdalqaderqassab
```

لحل هذه المشكله أو لنقدر الوصول إلى العناصر نستخدم ({}) .map(function(value)

```
const name = [“ammar”,”abdalqader”, “qassab”]
const element = <p>{name.map( (value) => {<p>{value}</p>} ) }</p>
>>ammar
abdalqader
qassab
```

تحديد خاصيّات ”HTML Attributes“ عن طريق JSX :

كل شيء في jsx هو عبارة عن كائن لذلك لتضمين الخاصية يكون أما من خلال نص "" أذا كانت مفردة أو كائن {} اذا كانت تحتوي على مسار أو ستيل أو متغيير أي تعابير جافا سكربت

بإمكانك استخدام علامتي الاقتباس لتحديد قيم ثابتة نصيّة لخاصيّات HTML :

```
const element = <div tabIndex="0"></div>;
```

بإمكانك أيضًا استخدام الأقواس لتضمين تعبير JavaScript بداخل خاصيّة HTML :

```
const element = <img src={user.avatarUrl}></img>;
const element = <div style={{color: “red”}}></div>;
                                        الكلمات التي تحتوي على جزئين أو عدة أجزاء تكتب بشكل camelCase
const element = <div style={{color: “red”, maginTop: “50px”}}></div>;
```

نعبر عن الخصائص كما هي من عدا className فهي تستخدم نظام camelCase وذلك بسبب أنها كلمة محجوزة :

```
const element = (<h1 className="greeting">Hello, world!</h1>);
```

ملاحظات :

١. حمل bable javascript للفيجوال استديو كود لتلوين الأكواد
٢. القالب يمكن ان يحتوي على عناصر داخلها عناصر و في كل منها ثمات للعناصر مثل syntex html بالضبط

```
const name1 = 'hello';
function formatname(user) {return user.toUpperCase(); };
const app =
<div>
ammar qassab
<br /> {name1 + ' ammar'}
<br /> {formatname(name1)}
<div id="id2" className="class2">ammar2</div> قيمة ثابتة للخواص
<div id={"id"+‘3’} className={"class3"}>ammar3</div> قيمة متغيرة للخواص ويمكن التحكم بها من خلال متغير
</div>
ReactDOM.render(app,document.getElementById('root'));
```

٣. يُترجِم Babel صيغة JSX إلى استدعاءات للتابع React.createElement() لذلك يكون المثالان التاليان متطابقين :

```
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);

const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!'
);
```

7

طريقة تصيير العناصر

نقوم بتعريف المكونات للقالب من خلال const لمنع الحقن ومن ثم تمريرها من خلال رياكت دوم بتالي يتم تمريرها لمرة واحدة والتالي عناصر القالب تكون ثابتة غير قابلة لتغيير لذلك لتغيرها نقوم بإعادة تمريرها فيقوم رياكت دوم بمقارنة العناصر والثمات والمحتويات ويقوم بالتغيير الأشياء الجديدة فقط من دون تغيير الجميع العناصر مثل الساعة نمرر القالب كل ثانية لمقارنة التغيرات في الساعة فيقوم بتغيير الأشياء الجديدة فقط مثل :

html

```
<div id="root"></div>
```

js

```
function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {new Date().toLocaleTimeString()}.</h2>
    </div>
  );
  ReactDOM.render(element, document.getElementById('root'));
}
setInterval(tick, 1000);
```

8

components / المكونات

تتيح لنا المكونات (component) تقسيم واجهة المستخدم أو القالب إلى قطع صغيرة قابلة لإعادة الاستخدام بأكثر من مكان وتفكير ببرمجة كل قطعة على إنفراد وبتالي تسهيل إعادة الهيكلة للقطع فالمكونات فهي تشبه دوال أو كلاس الجافا سكربت التي تقبل مدخلات المستخدم مثل الخاصيات (propertise) والمكون العبارة عن دالة أو كلاس يعيد ما يجب عرضه على الشاشة

هناك طريقتين لكتابة المكونات أما دوال (function) أو أصناف (class) :

١. function Name(props) {return (); }

٢. class Name extends React.Component {render() {return (this.props); } }

ملاحظة : أسم المكون يبدأ دوما بحرف كبير وعن استدعائه نكتب نفس الاسم الذي يبدأ بحرف كبير مثل <Name />

كل عملنا داخل ملف src حيث يوجد فيه ملف App.js الذي نجمع فيه كل المكونات ثم يمرر كل المكونات إلى ملف Index.js الذي بدوره يمرر كل التطبيق إلى ملف index.html

أولا ننشئ ملف داخل ملف src ونسميه Component ننشئ داخله مجلد لكل مكون ويكون يحمل أسم المكون حيث المكونات كل الموقع الذي بنهاية نمررها إلى ملف App.js حيث كل مكون ننشئه بملف لوحده ويكون أسم الملف هو أسم المكون مثل Namecomponent.js

و لاستدعاء المكون نكتب الجملة في الملف المراد استخدام المكون فيه ;'./' import Namecomponent from

طريقة استدعاء المكون :

١. إذا لم يكن يحتوي على على عناصر أطفال <Name />

٢. إذا كان داخله عناصر أطفال <Name> </Name>

٣. إذا كان يحتوي على خصائص <Name style="" src={} />

ملاحظة : طريقة الاستدعاء تنطبق على العناصر Html كذلك و يمكن إستدعاء المكونات والعناصر بعدد لانهائي من المرات

المكونات على شكل دوال function components :

يبدأ الأسم المكون بحرف كبير حصرا ويأخذ متغيير واحد على شكل كائن تتفرع منه متغييرات أخرى

```
import React from "react" ;

function Namecomponent(props) {
return (<img src={props.src} />);//لاحظ كيف قمنا باستدعاء القيم التي نرسلها عند استدعاء الكومبونت
}
إذا أردنا أن نستدعيه في ملف جافا سكربت أخر نجعله عام من خلال كتابة الأمر
```

```
export default Namecomponent ;
                                                    عندما نستدعي نكتب فقط
import Namecomponent from "./components/" ;
<Namecomponent src="" />
```

أو نستخدم أرو فانكشن لمنع الحقن :

```
import React from "react" ;

const Namecomponent = (props) => {
return <img src={props.src} />;// () إذا كان لدينا عنصر واحد لا يوجد حاجة لوضع
}
                                    إذا أردنا أن نستدعيه في ملف جافا سكربت أخر نجعله عام من خلال كتابة الأمر
export default Namecomponent ;
                                                    عندما نستدعي نكتب فقط
import Namecomponent from "./components/" ;
<Namecomponent src="" />
```

المكونات على شكل أصناف class components :

الأسم المكون دوما يبدأ بحرف كبير وله طريقتين للكتابة حسب import

الطريقة الأولى :

```
import React from "react" ;

class Namecomponent extend React.Component {
 render() {
   return (<img src={this.props.src});// لاحظ كيف قمنا باستدعاء القيم التي نرسلها عند استدعاء الكومبونت
 }
}
                                    إذا أردنا أن نستدعيه في ملف جافا سكربت أخر نجعله عام من خلال كتابة الأمر
export default Namecomponent ;
                                                    عندما نستدعي نكتب فقط
import Namecomponent from "./components/" ;
<Namecomponent src="" />
```

الطريقة الثانية :

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

class Namecomponent extend Component {
 render() {
   return (<img src={this.props.src});
```

```
  }
}
```

إذا أردنا أن نستدعيه في ملف جافا سكربت أخر نجعله عام من خلال كتابة الأمر

```
export default Namecomponent ;
```

عندما نستدعي نكتب فقط

```
import Namecomponent from "./components/" ;
<Namecomponent src="" />
```

ملاحظة : كل مكون يخرج منه عنصر واحد فقط و إذا كان بداخله عدد لانهائي من العناصر لا تفرق

```
function Welcome(props) {
 return (
  <div>
    <h1>hello</h1>
    <h1>hello</h1>
  </div>
 ) ;
}
```

ملاحظة : كل مكون يخرج منه عنصر واحد فقط و إذا كان بداخله عدد لانهائي من العناصر لا تفرق ولكن أنا لا أريد أن يخرج div عنصر رئيسي بل أريد أن يخرج العناصر التي في داخل فقط بدون عنصر container لذلك استخدم Fragment أو <> </> الأفضل Fragment

```
import React, {Fragment) from "react";
function Welcome(props) {
 return (
  <Fragment >
    <h1>hello</h1>
    <h1>hello</h1>
  </Fragment >
 ) ;
}
```

أو

```
import React from "react";
function Welcome(props) {
 return (
  < >
    <h1>hello</h1>
    <h1>hello</h1>
  </ >
 ) ;
}
```

ملاحظة : إذا كان لدينا أكثر من كمبونينت في ملف واحد وأردنا استدعائهم من خارج الملف نكتب export default قبل إنشاء الكومبونت

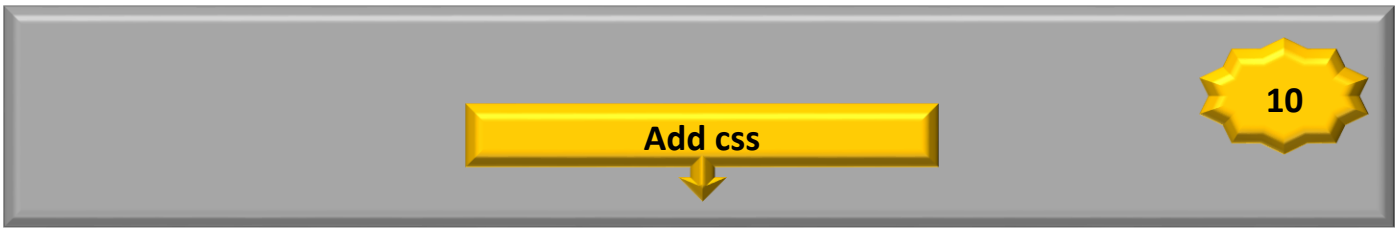
```
import React, { Component } from "react" ;

export default function Welcome(props) {
```

```
 return (
   <div>
     <h1>hello</h1>
     <h1>hello</h1>
   </div>
 ) ;
}

export default const Namecomponent = (props) => {
 return (
   <div>
     <h1>hello</h1>
     <h1>hello</h1>
   </div>
 ) ;
}

export default class Namecomponent extend Component {
 render() {
   return (<img src={this.props.src});
 }
}
```

معناها الخواص تستخدم في المكونات في العادة نقوم بتمريرها للمكون عند استدعائه وتستخدم في نوعي المكونات سواء كانت دوال أم أصناف لكن تختلف طريقة الاستقبال الخاصية من نوع لاخر عند تمريرها , لنستعرض طريقة بناء المكونات مع الخواص و طريقة الاستدعاء :

```
function Name(props) {return (<div>hello , {props.value} </div>);}
                                                                    الاستدعاء
<Name value="" />

class Name extends React.Component {render(){return(<div>hello {this.props.value}</div>);}}
                                                                    الاستدعاء
<Name value="" />

                    عند الاستدعاء و يكون هناك عناصر داخل العنصر المستدعى نقوم بالوصول اليهم عند طريق props.children مثل
function Name(props) {return (<div>hello , {props.value} from {props.children} </div>);}
<Name value=”ammar”><h1>abd qassab</h1></Name>
                                                  النتيجة تكون hello , ammar from ABD QASSAB

                                                  ملاحظة : يمكن تمرير أكثر من خاصية إلى الدالة مثل :
```

```
function Name(props) {return (<div>hello , {props.value+' '+props.value2} </div>);}
                                                                                الاستدعاء
<Name value="" value2="" />
```



يمكن إضافة ستايل داخلي أو خارجي سنستعرض الطرق

إضافة ستايل داخلي :

نعلم أن كل شيء في jsx هو عبارة عن كائن لذلك لتضمين الخاصية يكون أما من خلال نص "" أذا كانت مفردة أو كائن {} اذا كانت تحتوي على مسار أو ستيل أو متغيير أي تعابير جافا سكربت

```
const element = <div style={{color: “red”}}></div>;
                                                الكلمات التي تحتوي على جزئين أو عدة أجزاء تكتب بشكل camelCase
const element = <div style={{color: “red”, maginTop: “50px”}}></div>;
                                                                لوضعها ضمن متغيير
const sty = {color: “red”, maginTop: “50px”};
const element = <div style={sty}></div>;
```

لإضافة ستايل خارجي :

ننشئ مجلد لك كومبونت باسم الكومبونت داخل مجلد components و ننشئ ملف داخله مثلا App.css ونستدعيه عن طريق import ونستخدم الكلاسات عن طريق className=””

```
import React from "react";
import "./App/App.css";

export default function App (props) {
  return (
    <div className="cont">
      <h1>ammar qassab</h1>
    </div>
  );
}
```

module css :

عندما ننشئ ملفات css بالطريقة العادية مثل App.css / Card.css فأن الريأكت يقوم في نهاية الأمر بخلط كلاسات ملفات css ويجمعها مع بعضها و بالتالي عند تشابه أسماء كلاسين لملفات css سوف تختلط الخاصيات مع بعضها وتحصل مشكله لذلك لحل

المشكله ننشئ ملفات module لل css من خلال تسمية الملف App.module.css / Card.module.css وبتالي كل ملف css يكون مختلف عن الأخر و أذا تشابهت أسماء الكلاسات لن يؤثر ذلك على التنسيق ولكن تختلف طريقة الاستدعاء

```
import React from "react";
import styles from "./App/App.module.css";// لاحظ

export default function App (props) {
   return (
      <div className={styles.cont}>// لاحظ
         <h1>ammar qassab</h1>
      </div>
   );
}
```

11

الحالة ودورة حياة المكونات

لنتعرف على الحالة state :

وهي قيم خاصة محلية و تكون من نوع object و يمكن تغيرها مع الزمن و توضع فقط في كومبونت من نوع class

الفرق بين prope و state :

prope : وهي قيم خاصة فقط للقراءة ولا يمكن تعديلها ويتم تمريرها عند استدعاء الكومبونت ويمكن أنشائها من نمطين الكومبونت class و function

state : وهي قيم خاصة من نوع object للقراءة و الكتابة أي يمكن تغيرها و التعديل عليها و يظهر التعديل في الكومبونت مباشرة و ال react يتبع تغيرات state و عند تغيرها يقوم بتغيرها وتحديث الكومبونت وتوضع في كومبونات من نمط class

توضع ال state ضمن كومبونت ال class وتكون داخل الدالة البانية constructor(props) {}

```
export default class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {name: "ammar", lastname: "qassab"};
  }

  render() {
    return (
    <div>
      <div>name : {this.state.name}</div>
      <div>lasName : {this.state.lastname}</div>
    </div>
    );
    }
}
```

وظيفة super(prope) هو نقل القيم إلى React.Component ليعلم ال react بالقيم الموجودة

يتم تعريف قيم مبدأية للكائن ال this.state دوما داخل الدالة constructor وبعدها يتم تغيير القيم من الخارج من خلال الدالة this.setstate({name: "mohammad"})

يمكن عرض محتوى ال state من خلال استدعائه ككائن مثل this.state.name

فلنتذكر مثال الساعة من قسم تصيير العناصر، تعلّمنا في ذلك القسم فقط طريقة واحدة لتحديث واجهة المستخدم عن طريق استدعاء التابع ReactDOM.render() لتغيير الناتج :

```
function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {new Date().toLocaleTimeString()}.</h2>
    </div>
  );
  ReactDOM.render(element, document.getElementById('root'));
}
setInterval(tick, 1000);
```

سنتعلّم في هذا القسم كيف نجعل مُكوِّن الساعة Clock قابلًا لإعادة الاستخدام حقًّا مع تغليفه ضمن نفسه، حيث يُعيِّن عدّاد الوقت الخاص به ويُحدِّث نفسه في كل ثانية.

بإمكاننا البدء عن طريق تغليف شكل السّاعة:

```
function Clock(props) {
  return (
    <div>
      <h1>أهلًا بالعالم!</h1>
      <h2>الساعة الآن {props.date.toLocaleTimeString()}.</h2>
    </div>  );
}

function tick() {
  ReactDOM.render(
    <Clock date={new Date()} />,    document.getElementById('root')
  );
}

setInterval(tick, 1000);
```

ولكن على الرغم من ذلك يفتقد هذا المُكوِّن لمتطلب أساسي، وهو أنّ تعيين السّاعة لعدّاد الوقت وتحديثها لواجهة المستخدم في كل ثانية ينبغي أن يكون تفصيلًا داخليًّا خاصًّا بالمُكوِّن Clock وليس دالة خارجية تحدث المكون لذلك لنستخدم ال state

نريد بشكل مثالي أن نكتب هذه الشيفرة مرة واحدة ونجعل المُكوِّن Clock يُحدِّث نفسه في كل ثانية

أولا يجب تحويل الدالة إلى صنف :

بإمكانك تحويل مُكوِّنات الدوال مثل Clock إلى أصناف بخمس خطوات:

١. إنشاء صنف (ES6) بنفس الاسم والذي يمتد (extends) إلى React.Component.
٢. إضافة الدالة البانية
٣. إضافة تابع فارغ وحيد لهذا الصنف اسمه render()
٤. نقل جسم الدالة إلى التّابع render()
٥. تبديل props إلى this.props ومن ثم إلى this.state في جسم التّابع render()
٦. حذف باقي تصريح الدالة الفارغ

لتصبح النتيجة النهائية :

```
class Clock extends React.Component {
 constructor(props) {
  super(props);
  this.state = {date: new Date()};
 }
 render() {
  return (
   <div>
    <h1>أهلًا بالعالم!</h1>
    <h2>الساعة الآن {this.state.date.toLocaleTimeString()}.</h2>
   </div>
  );
 }
}

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

الأن المكون جاهز ولكن لا يحدث نفسه سنجعل الآن المُكوِّن Clock يُعيِّن عدّاد الوقت الخاص به ويُحدِّث نفسه في كل ثانية

إضافة توابع دورة الحياة إلى الصنف :

وهي خطافات خاصة بالكومبونات تراقب this.state

١. componentDidMount() { } وهو خطاف يحتوي على متغير لا على التعين مثل this.vab يتم حفظ فيه قيمة this.state الجديدة أو يستدعي دالة تغير this.state القديمة علما أن تغيير قيمة this.state يتم من خلال this.setstate({})
٢. componentWillUnmount() { clearInterval(this.vab); } وهو خطاف حيث عندما يتغيير المتغيير this.vab في componentDidMount() { } وذلك بتغيير قيمة this.state يفعل componentWillUnmount() ويقوم بحذف قيمة this.vab من الرام فهو يحافظ على الأداء

لتصبح النتيجة في النهاية :

```
class Clock extends React.Component {
 constructor(props) {
  super(props);
  this.state = {date: new Date()};
 }

 componentDidMount() {
  this.timerID = setInterval( () => this.tick(),1000);
 }

 componentWillUnmount() {
```

```
    clearInterval(this.timerID);
  }

  tick() {
    this.setState({
      date: new Date()
    });
  }

  render() {
    return (
      <div>
        <h1>أهلًا بالعالم</h1>
        <h2> الساعة الآن {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

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

تدق السّاعة الآن في كل ثانية.

فلنوجز بسرعة ما يجري ونذكر الترتيب الذي تُستدعى فيه التوابع:

١. عندما يُمرَّر العنصر <Clock /> إلى ReactDOM.render() تستدعي React الدالة البانية للمُكوِّن Clock وبما أنّ Clock يحتاج لإظهار الوقت الحالي سيُهيِّئ this.state بكائن يتضمّن الوقت الحالي، ولاحقًا يُحدِّث هذه الحالة.

٢. تستدعي بعدها React التّابع render() للمُكوِّن Clock وهكذا تعلم React ما الذي ينبغي عرضه على الشاشة. تُحدِّث React بعد ذلك DOM ليُطابِق ناتج Clock

٣. عندما يُدخَل ناتج Clock إلى DOM ، تستدعي React خُطاف دورة الحياة componentDidMount()، وبداخله يسأل المُكوِّن Clock المتصفّح أن يُعيِّن عدّاد الوقت لاستدعاء التابع tick() الخاص بالمُكوِّن مرّة كل ثانية.

٤. يستدعي المتصفّح في كل ثانية التّابع tick()، وبداخله يُجدول المُكوِّن Clock تحديثًا لواجهة المستخدم عن طريق استدعاء setState() مع كائن يحوي على الوقت الحالي. وبفضل استدعاء setState() تعلم React أنّ الحالة تغيّرت وبذلك تستدعي التّابع render() مرّة أخرى ليعلم ما الذي ينبغي أن يكون على الشاشة، ستكون this.state.date في التابع render() مختلفة هذه المرّة، وبهذا يتضمّن الناتج الوقت المُحدَّث. تُحدِّث React وفق ذلك DOM

٥. إن أُزيل المُكوِّن Clock من DOM تستدعي React دالة دورة الحياة componentWillUnmount() بحيث يتوقّف عدّاد الوقت.

نعين الوقت تبعا لحدث الضغط على الزر :

يتم تعين الوقت من استدعاء الخطاف componentDidMount وهناك طريقتين :

طريقة الأولى هي استدعاء الخطاف من دون تعريفه في الدالة البانية :

تتم من خلال تحويل الخطاف إلى أرو فانكشن componentDidMount = () => {} واستدعائو من خلال this.componentDidMount علما أنه يجب تحويله لأرو فانكشن لأن لم يتم تعريفه في الدالة البانية واستدعائه من خلال this.componentDidMount لتعلم الريأكت أن الدالة في نفس المكون

```
export default class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }
  componentDidMount = () => {
     this.timerID = this.setState({date: new Date()});
   }// لاحظ كيف عرفناه كأرو فانكشين وكيف يتحدث الستات

    componentWillUnmount() {
     clearInterval(this.timerID);
    }

  render() {
    return (
      <div>
        <h1>أهلًا بالعالم!</h1>
        <h2> الساعة الآن{this.state.date.toLocaleTimeString()}.</h2>
        <button onClick={this.componentDidMount}>enter</button>
      </div>
    );
  }
 }
```

هكذا يتم تحديث الوقت كلما ضغطنا على الزر

الطريقة الثانية هي استدعاء الخطاف مع تعريفه في الدالة البانية :

تتم من خلال الخطاف العادي componentDidMount () {} من دون تحويله إلى أرو فانكشين واستدعائو من خلال this.componentDidMount علما أنه يجب أن يتم تعريفه في الدالة البانية من خلال .bind(this) واستدعائه من خلال this.componentDidMount لتعلم الريأكت أن الدالة في نفس المكون

```
export default class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
    this.componentDidMount = this.componentDidMount.bind(this);
  }
  componentDidMount() {
     this.timerID = this.setState({date: new Date()});
```

```
    }// لاحظ كيف عرفناه كدالة عادية

    componentWillUnmount() {
      clearInterval(this.timerID);
    }

  render() {
    return (
      <div>
        <h1>أهلًا بالعالم!</h1>
        <h2> الساعة الآن {this.state.date.toLocaleTimeString()}.</h2>
        <button onClick={this.componentDidMount}>enter</button>
      </div>
    );
  }
}
```

هكذا يتم تحديث الوقت كلما ضغطنا على الزر

ملاحظات :

المكان الوحيد الذي يُمكنك فيه تعيين this.state كقيمة إبدائية هو الدالة البانية ولا يمكن تعينها في مكان أخر

لا تعدل الحالة state بشكل مباشر

```
// طريقة خاطئة
this.state.comment = 'أهلًا';
```

استخدم setState() بدلًا من ذلك

```
// الطريقة الصحيحة
this.setState({comment: 'أهلًا'});
```

ال state هي كائن يمكن أن يحتوي داخله على نص أو متغير أو كائن أخر أو دالة .....

```
constructor(props) {
    super(props);
    this.state = {
                    date: new Date(),
                    name: 'ammar',
                    obj:{},
                    arr:[ 'ammar', 'qassab'],
                    fun: () =>{}
                  };
}
```

قد تجمع React نداءات عديدة للتّابع setState() في تحديث واحد من أجل تحسين الأداء.

بما أنّ this.props و this.state قد تُحدَّث بشكل غير متزامن، فيجب ألّا تعتمد على قيمها لحساب الحالة التالية.

على سبيل المثال قد تفشل الشيفرة التالية بتحديث عدّاد الوقت:

```
// طريقة خاطئة
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

لإصلاح ذلك استخدم شكل آخر من setState() يقبل دالة بدلًا من كائن، حيث تستقبل هذه الدالة الحالة السّابقة كوسيط أول لها، والخاصيّات props في وقت تطبيق التحديث كوسيطٍ ثانٍ لها:

```
// الطريقة الصحيحة
this.setState((prevState, props) => ({
  counter: prevState.counter + props.increment
}));
```

استخدمنا الدوال السهمية في المثال السّابق، ولكنّها تعمل أيضًا مع الدوال الاعتياديّة:

```
// الطريقة الصحيحة
this.setState(function(prevState, props) {
  return {
    counter: prevState.counter + props.increment
  };
});
```

مثال عداد يقوم بالزيادة واحد أو تنقص واحد من الأعداد الموجبة فقط :

```
export default class App extends React.Component {
   constructor(prope) {
      super(prope);
      this.state = {number:0};
      this.increase = this.increase.bind(this);
      this.decrement = this.decrement.bind(this);
   }

   increase() {
      this.setState(() => ({number : this.state.number + 1}));
   }

   decrement() {
      if (this.state.number > 0) {
         this.setState(() => ({number : this.state.number - 1}));
      }
   }

   render() {
      return (
         <div>
            <div>number : {this.state.number}</div>
            <button onClick={this.increase}>increase</button>
            <button onClick={this.decrement}>decrement</button>
         </div>
      );
   }
}
```

ملاحظة : في setstate({}) يجب تمرير قيم ال this.state القديمة ثم نكتب الجديدة لان يعمل تحديث لكل ال state لذلك نكتب this.state... ثم تحديث القيمة الجديدة لكي لا يفقد المعلومات لكي لا ننظر لكتابة كل المعلومات فهو يكتب ثم يحدث الجديدة

مثل

```
this.setState(() => ({ ...this.state,
 number : this.state.number + 1})
);
```

تتدفق البيانات للمستويات الأدنى :

لا تعلم المُكوِّنات الآباء ولا حتى الأبناء إن كان مُكوِّن المُحدَّد لديه حالة أو بدون حالة، ولا يجب أن تُبالي إن كان مُعرَّفًا كدالة أو كصنف.

هذا هو السبّب وراء تسمية الحالة بالمحليّة أو المُغلَّفة، فهي غير قابلة للوصول من قبل أي مُكوِّن آخر غير المُكوِّن الذي يملكها ويُعيّنها.

قد يختار المُكوِّن تمرير حالته كخاصيّات props إلى عناصره الأبناء في المستوى الأدنى :

```
class Clock extends React.Component {
 constructor(props) {
  super(props);
  this.state = {date: new Date()};
 }
 render() {
  return (
   <div>
    <h1>أهلًا بالعالم!</h1>
    <FormattedDate date={this.state.date} />
   </div>
  );
 }
}
```

يستقبل المُكوِّن FormattedDate التاريخ date في خاصيّاته ولن يعلم ما إذا كان هذا التاريخ قد أتى عن طريق حالة المُكوِّن Clock أو من خاصيّات Clock، أو كُتِب بشكل يدوي:

```
function FormattedDate(props) {
 return <h2> الساعة الآن {props.date.toLocaleTimeString()}.</h2>;
}
```

يُدعى هذا عادةً بتدفق البيانات من المستوى الأعلى للأدنى (top-down) أو أحادي الاتجاه (unidirectional). حيث أي حالة يمتلكها مُكوِّن مُحدَّد، وأي بيانات أو واجهة مستخدم مُشتقَّة من تلك الحالة بإمكانها فقط التأثير على المُكوِّنات التي تحتها في شجرة المُكوِّنات.

تُشبه معالجة الأحداث لعناصر React معالجة الأحداث لعناصر DOM ، ولكن هنالك فروق تتعلّق بالصياغة :

١. تُسمَّى أحداث React باستخدام حالة الأحرف camelCase (أي عند وجود اسم مؤلف من عدة كلمات نجعل الحرف الأول من الكلمة الأولى بالشكل الصغير أمّا باقي الكلمات نجعل حرفها الأول بالشكل الكبير) بدلًا من استخدام الشكل الصغير للأحرف.

٢. نُمرِّر في JSX دالة كمُعالِج للأحداث بين {} بدلًا من سلسلة نصيّة "" وتكون الدالة بدون أقواس ()

٣. من الفروق الأخرى أنّه لا يمكنك إعادة القيمة false لمنع السلوك الافتراضي للحدث من العمل في React

٤. عند استخدام React بشكل عام لاينبغي استدعاء addEventListener لإضافة مُستمِع للأحداث إلى عنصر DOM بعد إنشائه، وبدلًا من ذلك نُضيف مُستمِعًا للأحداث عند تصيير العنصر (Rendering Element) و إنشاء state

مثال :

على سبيل المثال لنأخذ شيفرة HTML التالية :

```
<button onclick="activateLasers()">
  Activate Lasers
</button>
```

تكون الشيفرة السابقة مختلفة قليلًا في React :

```
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

ملاحظة : في الريأكت عندما نستدعي الفانكشن مع الأقواس فأن الريأكت سوف يعمل لها run بدون تفعيل الحدث و ذلك عند بناء الكومبوننات لذلك نكتبها بدون أقواس مثل السابق

```
<button onClick={activateLasers()}>// سوف يعمل لها رن عند بناء الكومبوننات وهي طريقة خاطئة
  Activate Lasers
</button>
```

ملاحظة : ولكن نحنا نريد أن نعمل معها أقواس لتعالج نفسها داخل الحدث في هذه الحالة نعملها داخل أرو فانكشن وبتالي لن يعمل لها run عند بدء بناء الكومبونات

```
<button onClick={(ev) => {activateLasers(ev) } }>// لن يعمل لها رن عند بناء الكومبوننات
  Activate Lasers
</button>
```

من الفروق الأخرى أنّه لا يمكنك إعادة القيمة false لمنع السلوك الافتراضي للحدث من العمل في React بل يجب عليك أن تستدعي preventDefault أو stopPropagation() بشكل صريح

فمثلًا في HTML لمنع السلوك الافتراضي للروابط في فتح صفحة جديدة بإمكانك كتابة ما يلي :

```
<form onsubmit="console.log('You clicked submit.'); return false">
  <button type="submit">Submit</button>
</form>
```

أمّا في React فنكتب بدلًا من ذلك ما يلي :

```
function Form() {
 function handleSubmit(e) {
  e.preventDefault();   console.log('You clicked submit.');
 }

 return (
  <form onSubmit={handleSubmit}>
   <button type="submit">Submit</button>
  </form>
 );
}
```

يُمثِّل المتغيّر e أو onSubmit هنا حدثًا مُصطنعًا، حيث تُعرِّف React هذه الأحداث المُصطنعة وفق معايير W3C spec، بحيث لا نهتم بمشاكل التوافقيّة بين المتصفحات . لا تعمل أحداث React تمامًا مثل الأحداث الأصلية

عند استخدام React بشكل عام لاينبغي استدعاء addEventListener لإضافة مُستمِع للأحداث إلى عنصر DOM بعد إنشائه، وبدلًا من ذلك نُضيف مُستمِعًا للأحداث عند تصيير العنصر (Rendering Element) و إنشاء state

كما تعلمنا في state يمكن استدعاء الدالة بأكثر من طريقة :

طريقة الأولى : استدعاء الدالة مع تعريفها في الدالة البانية وتمرير .bind(this) لها ويكون شكل الدالة أنينموس فانكشن

```
class Toggle extends React.Component {
 constructor(props) {
  super(props);
  this.state = {isToggleOn: true};
  this.handleClick = this.handleClick.bind(this);
 }

 handleClick() {
  this.setState(prevState => ({
   isToggleOn: !prevState.isToggleOn
  }));
 }
 render() {
  return (
   <button onClick={this.handleClick}>
    {this.state.isToggleOn ? 'ON' : 'OFF'}
   </button>
  );
 }
}

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

طريقة ثانية : استدعاء الدالة من دون تعريفها في الدالة البانية ويكون شكل الدالة أيرو فانكشن

تسمى هذه الطريقة باسم حقول الأصناف

```
class LoggingButton extends React.Component {

handleClick = () => {
   console.log('this is:', this);
 }
 render() {
   return (
     <button onClick={this.handleClick}>
       Click me
     </button>
   );
 }
}
```

الطريقة الثالثة : استدعاء الدالة من دون تعريفها في الدالة البانية ويكون شكل الدالة انانموس فانكشن ولكن نستدعيها داخل الحدث باستخدام دالة .bind(this) أي onEvent={this.fun.bind(this)} بدل onEvent={this.fun}

```
class LoggingButton extends React.Component {
 handleClick() {
   console.log('this is:', this);
 }

 render() {
   return (
     <button onClick={this.handleClick.bind(this)}>// لاحظ
       Click me
     </button>
   );
 }
}
```

الطريقة الرابعة : استدعاء الدالة من دون تعريفها في الدالة البانية ويكون شكل الدالة انانموس فانكشن ولكن نستدعيها داخل الحدث باستخدام دالة أرو فانكشن onEvent={() => this.fun()} بدل onEvent={this.fun}

هذه الطريقة فيها مشاكل في الأداء

```
class LoggingButton extends React.Component {
 handleClick() {
   console.log('this is:', this);
 }

 render() {
   return (
     <button onClick={() => this.handleClick()}>// لاحظ
       Click me
     </button>
```

```
    );
  }
}
```

المشكلة في هذه الصياغة هي إنشاء رد نداء مختلف في كل مرّة يُصيَّر فيها المُكوّن LoggingButton وفي معظم الحالات يكون هذا مقبولًا أي كل ما يتم أستدعاء المكون يعمل الحدث و كذلك إن مرَّرنا رد النداء هذا كخاصيّة prop إلى المُكوّنات الموجودة في المستوى الأدنى، فقد تقوم هذه المُكوّنات بعمل إعادة تصيير (re-rendering) إضافيّة.

نوصي بشكل عام الربط في الدالة البانية (constructor) أو استخدام صياغة حقول الأصناف لتجنّب مثل هذا النوع من مشاكل الأداء.

تمرير وسائط إلى معالجات الأحداث :

أن كان المتغيّر id يُمثِّل مُعرِّف العنصر لمعرفة من قام بالحدث فسيعمل كلا السطرين التاليين بنفس الكفاءة:

```
<button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>
<button onClick={this.deleteRow.bind(this, id)}>Delete Row</button>
```

في كلتا الحالتين سيُمرَّر الوسيط e الذي يُمثِّل حدث React كوسيطٍ ثانٍ بعد المُعرِّف ID في الدوال السهمية حيث يجب أن نُمرِّره بشكلٍ صريح، ولكن في حالة استخدام التابع bind فستُمرَّر أي وسائط أخرى تلقائيًّا.

13

التصيير الشرطي

العرض الشرطي في React يعمل بنفس طريقة عمل العرض الشرطي في لغة JavaScript. قم باستخدام المعاملات الخاصة بلغة JavaScript، مثل if أو conditional operator لإنشاء العناصر التي تمثل الحالة (State) ، وسوف يقوم React بتحديث الواجهه الأماميه (UI) لمطابقتها.

```
function UserGreeting(props) {
  return <h1>Welcome back!</h1>;
}

function GuestGreeting(props) {
  return <h1>Please sign up.</h1>;
}

function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn) {
    return <UserGreeting />;
  }
  return <GuestGreeting />;
}
```

```
ReactDOM.render(
 <Greeting isLoggedIn={false} />,
 document.getElementById('root')
);
```

في حال مررنا false يعرض <GuestGreeting /> أي Please sign up.

في حال مررنا true يعرض <UserGreeting /> أي Welcome back!

مثال أخر لنضيف حالة وكبسات

```
class LoginControl extends React.Component {
 constructor(props) {
  super(props);
  this.handleLoginClick = this.handleLoginClick.bind(this);
  this.handleLogoutClick = this.handleLogoutClick.bind(this);
  this.state = {isLoggedIn: false};
 }

 handleLoginClick() {
  this.setState({isLoggedIn: true});
 }

 handleLogoutClick() {
  this.setState({isLoggedIn: false});
 }

 render() {
  const isLoggedIn = this.state.isLoggedIn;
  let button;

  if (isLoggedIn) {
   button = <LogoutButton onClick={this.handleLogoutClick} />;
  } else {
   button = <LoginButton onClick={this.handleLoginClick} />;
  }

  return (
   <div>
    <Greeting isLoggedIn={isLoggedIn} />
    {button}
   </div>
  );
 }
}

function LoginButton(props) {
 return (
  <button onClick={props.onClick}>
   Login
```

```
    </button>
  );
}

function LogoutButton(props) {
  return (
    <button onClick={props.onClick}>
      Logout
    </button>
  );
}

function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn) {
    return <UserGreeting />;
  } else {
    return <GuestGreeting />;
  }
}

function UserGreeting(props) {
  return <h1>Welcome back!</h1>;
}

function GuestGreeting(props) {
  return <h1>Please sign up.</h1>;
}

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

الحالة الأفتراضية this.state هي false

استخدام معامل الشرطي condition ? true : false :

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      {isLoggedIn
        ? <LogoutButton onClick={this.handleLogoutClick} />
        : <LoginButton onClick={this.handleLoginClick} />
      }
    </div>  );
```

```
}
```

ذلك يجعل الأمر أقل وضوحاً لفهم ما يحدث

التعبير الشرطي المباشر بإستخدام معامل && المنطقي :

التعبير true && expression أي يجب أن تكون true من أجل عرض expression

و التعبير false && expression دائماً يعطي الناتج false

```
function Mailbox(props) {
  const unreadMessages = props.unreadMessages;
  return (
    <div>
      <h1>Hello!</h1>
      {unreadMessages.length > 0 &&
        <h2>
          You have {unreadMessages.length} unread messages.
        </h2>
      }
    </div>
  );
}

const messages = ['React', 'Re: React', 'Re:Re: React'];
ReactDOM.render(
  <Mailbox unreadMessages={messages} />,
  document.getElementById('root')
);
```

مثال أخر يعطي false

```
render() {
  const count = 0;  return (
    <div>
      { count && <h1>Messages: {count}</h1>}    </div>
  );
}
```

لاحظ أن إرجاع تعبير خاطئ سيؤدي إلى تخطي العنصر بعد && ولكنه سيعيد التعبير الخاطئ. في المثال أدناه ، سيتم إرجاع 0 من خلال طريقة العرض

منع المكوّن (Component) من التصيير :

يتم منع مكون من التصيير من خلال إرجاع القيمة null من خلال معامل شطري

```
function WarningBanner(props) {
  if (!props.warn) {
    return null;
  }
  return (
    <div className="warning">
```

```
    Warning!
   </div>
  );
}

class Page extends React.Component {
 constructor(props) {
  super(props);
  this.state = {showWarning: true};
  this.handleToggleClick = this.handleToggleClick.bind(this);
 }

 handleToggleClick() {
  this.setState(state => ({
   showWarning: !state.showWarning
  }));
 }// يعكس القيمة

 render() {
  return (
   <div>
    <WarningBanner warn={this.state.showWarning} />
    <button onClick={this.handleToggleClick}>
     {this.state.showWarning ? 'Hide' : 'Show'}
    </button>
   </div>
  );
 }
}

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

إعطاء الناتج null في التابع render الخاص بالمكوّن لا يؤثر على حدوث التوابع الخاصة بدورة حياة المكوّن (Lifecycle Methods). فمثلاً التابع componentDidUpdate سوف يتم استدعاءه كالمعتاد.

فلنتذكّر أولًا كيفيّة تحويل القوائم في JavaScript :

في المثال التالي سنستخدم الدالة map() لمضاعفة قيم مصفوفة من الأرقام اسمها numbers وسنُعيّن المصفوفة الجديدة التي تُعيدها الدالة map () إلى المتغير doubled ثم نعرض محتواه عبر التابع console.log()

```
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map((number) => number * 2);
console.log(doubled);
```

نتيجة تنفيذ هذا المثال هي [2, 4, 6, 8, 10]

للنفذ الفكرة في react

مكون يحتوي على قائمة بسيطة :

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li>{number}</li>
  );
  return (
    <ul>{listItems}</ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

النتيجة هي قائمة مرقمة من ١ إلى ٥

حيث نمرر المصفوفة إلى المكون بعدها نعرف متغير و نستدعي الدالة map() لفرط كل عنصر من المصفوفة وتنشئ مصفوفة جديدة بداخلها عناصر <li> ثم تمرر إلى القائمة <ul> التي يتم إرجاعها

عندما تُنفَّذ هذه الشيفرة ستتلقى تحذيرًا أنّه يجب تزويد مفتاح (key) لعناصر القائمة

المفاتيح key :

هو عبارة عن خاصيّة على شكل سلسلة نصيّة يجب إضافتها عند إنشاء قوائم من العناصر و كقاعدة عامة تحتاج العناصر المُستدعاة من قبل التابع map() إلى مفاتيح فتُساعِد المفاتيح React على معرفة العناصر التي تغيرت، أو أُضيفت، أو أُزيلت.

يجب أن تُعطَى المفاتيح للعناصر بداخل المصفوفة وذلك لإعطاء هذه العناصر هوية مستقرة id

لإضافة مفتاح للقائمة السابقة

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li key={number.toString()}>
      {number}
```

```
  </li>
 );
 return (
  <ul>{listItems}</ul>
 );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
 <NumberList numbers={numbers} />,
 document.getElementById('root')
);
```

وبتالي أي عنصر يتم إنشائه من قبل map() يحتاج إلى مفتاح kay مو شرط

الاستخدام الخاطئ للمفتاح :

```
function ListItem(props) {
 const value = props.value;
 return (
  // خطأ! فليس هنالك حاجة لتحديد المفتاح هنا خارج الماب
  <li key={value.toString()}>
   {value}
  </li>
 );
}

function NumberList(props) {
 const numbers = props.numbers;
 const listItems = numbers.map((number) =>
  // خطأ! هنا يجب وضع المفتاح داخل الماب
  <ListItem value={number} />
 );
 return (
  <ul>
   {listItems}
  </ul>
 );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
 <NumberList numbers={numbers} />,
 document.getElementById('root')
);
```

الاستخدام الصحيح للمفتاح :

```
function ListItem(props) {
  // . صحيح! فليس هنالك حاجة لتحديد المفتاح هنا
return <li>{props.value}</li>;
}
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    // صحيح! يجب تحديد المفتاح بداخل الماب التي تنشأ المصفوفة
<ListItem key={number.toString()} value={number} /> );
  return (
    <ul>
      {listItems}
    </ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

تسمح JSX بتضمين أي تعبير موجود بين قوسين لذا يمكننا وضع نتيجة التابع map() بشكل سطري(inline) :

```
function NumberList(props) {
  const numbers = props.numbers;
  return (
    <ul>
      {numbers.map((number) =>
        <ListItem key={number.toString()} value={number} />
      )}
    </ul>
  );
}
```

بإمكانك استخدام فهرس العنصر index كمفتاح map((value, index) => {}) :

```
const todoItems = todos.map((todo, index) =>
  // افعل ذلك فقط إن لم لكن للعناصر معرّفات مستقرة
  <li key={index}>
    {todo.text}
  </li>
);
```

افعل ذلك فقط إن لم لكن للعناصر معرّفات مستقرة

لا نُفضِّل استخدام فهارس العناصر إن كان ترتيبها عُرضةً للتغيير، فقد يُؤثِّر هذا بشكل سلبي على الأداء وقد يسبب مشاكل مع حالة المُكوِّن إن اخترت عدم تعيين مفتاح لعناصر القائمة فستستخدم React الفهارس كمفاتيح بشكل افتراضي.

المثال التالي ننشأ فيه فهرس لكل عنصر وهي الطريقة الأفضل لعناصر المصفوفة :

حيث يجب أن يكون لكل عنصر في المصفوفة له معرف id لا يشبه معرف عنصر أخير في المصفوفة

```
function Blog(props) {

 const content = props.posts.map((post) =>
  <div key={post.id}>
   <h3>{post.title}</h3>
   <p>{post.content}</p>
  </div>
 );
 return (
  <div>
   {content}
  </div>
 );
}

const posts = [
 {id: 1, title: 'Hello World', content: 'Welcome to learning React!'},
 {id: 2, title: 'Installation', content: 'You can install React from npm.'}
];
ReactDOM.render(
 <Blog posts={posts} />,
 document.getElementById('root')
);
```

هنا عرفنا لكل عنصر في المصفوفة id خاص فيه من أجل فهرسة العناصر وهي الطريقة الأفضل و المستخدمة

ملاحظة : تعمل المفاتيح كتلميح في React ، ولكنها لا تُمرَّر إلى المُكوِّنات. إن احتجت نفس القيمة في مُكوِّناتك فمرّرها بشكل صريح كخاصيّة prop مع استخدام اسم آخر

```
const content = posts.map((post) =>
 <Post
  key={post.id}
  id={post.id}
  title={post.title} />
);
```

بعدها نضمنها إلى المكون

مثال مع تمرير البيانات لمكونات أدنى أي أطفال :

ننشئ لكل كومبونت ملف خاص فيه داخل ملف components ويكون داخل كل مجلد ملف ستايل و جافاسكربت

App.js

```
import React from "react";
import Card from "../Card/Card";
import styles from "./App.module.css";

export default function App (props) {
   const dateServer = [
      {
      id : 1,
      date:{
         name : 'ammar',
         age : '25',
         adress : 'syria'}
      },
      {
      id : 2,
      date:{
         name : 'mohammad',
         age : '22',
         adress : 'syria2'}
      },
      {
      id : 3,
      date:{
         name : 'bashar',
         age : '16',
         adress : 'syria3'}
      },
   ];

   return (
      <div className={styles.cont}>
         <h1>boys</h1>
         <Card dateServerlist={dateServer}/>
      </div>
   );
}
```

App.module.css

```
.cont{
  width: 500px;
  margin: auto;
  padding: 20px;
  border: 2px solid red;
  display: flex;
  align-content: center;
  justify-content: center;
  flex-direction: column;
```

```
}
```

Card.js

```
import React from "react";
import styles from "./Card.module.css";

export default function Card (props) {
   const dateServerlist = props.dateServerlist;
   const cards = dateServerlist.map( (value) => {
      return (
<div className={styles.card} key={value.id} style={{backgroundColor : value.id % 2 === 0 ? "gold" : "wh
ite"}}>
            <div>name : {value.date.name}</div>
            <div>age : {value.date.age}</div>
            <div>adress : {value.date.adress}</div>
            <div className={styles.deletecard}>x</div>
         </div>
      );
   }
   );

   return (
      <>
      {cards}
      </>
   );
}
```

Card.module.css

```
.card{
   margin-bottom: 20px;
   padding: 10px;
   border: 2px solid red;
   position: relative;
 }
 .deletecard{
   position: absolute;
   width: 22px;
   height: 22px;
   top: -11px;
   right: -11px;
   background-color: red;
   font-weight: bold;
   border-radius: 50%;
   display: flex;
   align-content: center;
   justify-content: center;
 }
```

ملاحظة : عند تغيير المعلومات في state يقوم الكومبونت بعمل render() لكل الكومبونت الذي حصل فيه تغيير لذلك نبني الكومبونت الذي سوف سيحصل فيه تغيير بشكل منفصل عن باقي الكومبونات من أجل لا يعمل render() لكل الكمبونات المشتركه وذلك من أجل تسريع الكود

ملاحظة : ثبت الإضافة التالية للكروم react dev tool

ليكن لدينا عنصر <form> التالي :

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

يمتلك هذا النموذج نفس السّلوك الافتراضي لنماذج HTML من حيث الانتقال إلى صفحة جديدة عندما يضغط المستخدم على زر تقديم البيانات (Submit) ، وإن أردت فقط هذا السّلوك في React فسيعمل بشكل جيّد معك، ولكن في معظم الأحيان من الملائم أكثر أن نمتلك دالة في JavaScript تتعامل مع تقديم البيانات ولديها الوصول إلى البيانات التي أدخلها المستخدم في النموذج. الطريقة القياسيّة لتحقيق هذا الأمر هي باستخدام تقنيّة تدعى المُكوّنات المضبوطة (controlled components)

المكونات المضبوطة :

تُحافِظ عناصر النموذج في HTML مثل <input> و <textarea> و <select> على حالتها الخاصّة وتُحدِثها وفقًا لمُدخلات المستخدم

أمّا في React فيُحتفَظ بحالة قابلة للتعديل ضمن خاصيّة الحالة state للمُكوّنات وتُحدَّث فقط عن طريق التابع setState()

بإمكاننا الجمع بينهما بأن نجعل حالة state المصدر الوحيد للحقيقة، فبذلك يُصبِح مُكوّن React الذي يُصيِّر النموذج مُتحكِّمًا أيضًا بما يحدث في ذلك النموذج مع تتالي مُدخلات المستخدم

يُدعى عنصر الإدخال والذي تتحكّم React بقيمته بالمُكوّن المضبوط .

العنصر <form> :

على سبيل المثال إن أردنا في المثال السّابق أن نعرض الاسم بعد تقديمه فبإمكاننا كتابة النموذج كمُكوّن مضبوط :

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: ''};
```

```
  this.handleChange = this.handleChange.bind(this);
  this.handleSubmit = this.handleSubmit.bind(this);
 }

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

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

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

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

لمّا كانت خاصيّة القيمة value مُعيَّنة عن طريق عنصر النموذج فستكون قيمتها المعروضة دومًا هي this.state.value وبذلك نجعل حالة state المصدر الوحيد للحقيقة

وبما أنّ التابع handleChange يُنفَّذ عند كل تغير لحقل <form> من المستخدم ليُحدِّث حالة state فستتحدّث القيمة المعروضة بينما يكتب المستخدم .

باستخدام المكون المضبوط تكون قيمة المدخلات مدفوعة دائمًا بجهة الحالة state بينما يعني هذا أنه يجب عليك كتابة شيفرة أكثر قليلاً، كما يمكنك الآن تمرير القيمة إلى عناصر واجهة المستخدم الأخرى أيضًا ، أو إعادة تعيينها من معالجات الأحداث الأخرى.

العنصر <textarea> :

في HTML يُعرَّف نص العنصر <textarea> بشكلٍ مباشر كما يلي :

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

أما في ال react يعرف بشكل مشابه لل <input> مع وضع قيمة أبدائية لتظهر داخل العنصر

```
class NameForm extends React.Component {
 constructor(props) {
  super(props);
  this.state = {
   value: 'Please write an essay about your favorite DOM element'
  };

  this.handleChange = this.handleChange.bind(this);
  this.handleSubmit = this.handleSubmit.bind(this);
 }

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

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

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

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

العنصر <select> :

في HTML يُنشِئ العنصر <select> قائمة مُنسدِلة، فمثلًا تُنشِئ هذه الشيفرة قائمة مُنسدِلة ببعض أسماء الفاكهة

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

```
  <option value="mango">Mango</option>
</select>
```

لاحظ أنّ الخيار المبدئي هنا هو البرتقال بسبب وجود الخاصيّة selected بجانبه، ولكن في React بدلًا من استخدام الخاصيّة selected نستخدم الخاصيّة value ضمن العنصر <select> وهذا أسهل في المُكوّنات المضبوطة لأنّك ستحتاج لتعديلها فقط في مكانٍ واحد . ويمكننا إعطائها قيمة إبتدائية من value الخيارات وبذلك نحقق selected على سبيل المثال :

```
class FlavorForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: 'coconut'};

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

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

  handleSubmit(event) {
    alert('Your favorite flavor is: ' + this.state.value);
    event.preventDefault();
  }

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

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

ملاحظة : بإمكانك تمرير مصفوفة إلى الخاصيّة value حيث يُتيح لك ذلك انتقاء عدّة خيارات في العنصر <select> :

بإمكانك تمرير مصفوفة إلى الخاصيّة value حيث يُتيح لك ذلك انتقاء عدّة خيارات في العنصر <select> :

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

وبهذا نجد أنّ العناصر <input type="text"> و <textarea> و <select> تعمل بشكلٍ مماثل، فجميعها تقبل الخاصيّة value والتي نستخدمها لتنفيذ المُكوِّن المضبوط

التعامل مع إدخالات متعددة لمكونات المضبوطة :

عندما تحتاج إلى التعامل مع عناصر input مُتعدِّدة مضبوطة فبإمكانك إضافة الخاصيّة name إلى كل عنصر وتترك لدالة معالجة الأحداث أن تختار ما ستفعله بناءً على قيمة event.target.name.
فلنأخذ هذا المثال :

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

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

 handleInputChange(event) {
  const target = event.target;
  const value = target.type === 'checkbox' ? target.checked : target.value;
  const name = target.name;

  this.setState({
   [name]: value
  });// جعلنا القوسين من أجل حسب أسم الحالة يغيرها
 }

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

```
          value={this.state.numberOfGuests}
          onChange={this.handleInputChange} />
        </label>
        <br/>
        {`name isGoing : ${this.state.isGoing}`}
        <br/>
        {`name numberOfGuests : ${this.state.numberOfGuests}`}
      </form>
    );
  }
}

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

لاحظ كيف استخدمنا صياغة ES6 اسم الخاصيّة المحسوب لتحديث مفتاح الحالة بما يُوافِق الاسم المُدخَل :

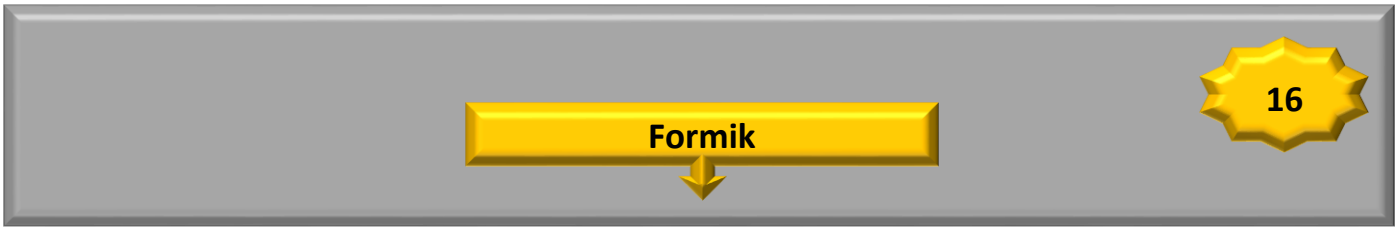
```
this.setState({
  [name]: value
});
```

بما أنّ الدالة setState() تدمج تلقائيًّا حالة جزئيّة مع الحالة الحاليّة سنحتاج فقط إلى استدعائها مع الأجزاء المتغيّرة

عنصر إدخال الملفات :

في HTML يُتيح العنصر <input type="file"> للمستخدم أن يختار ملفًّا واحدًا أو أكثر من جهازه لتحميلها إلى الخادم أو التعامل معها عن طريق JavaScript وذلك عبر واجهة برمجة التطبيق الخاصّة بالملف وبما أنّ قيمته هي قابلة للقراءة فقط فهو مُكوِّن غير مضبوط (uncontrolled component) في React ، سنناقش هذا المُكوِّن مع المُكوِّنات غير المضبوطة الأخرى في قسمها الخاص

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



17

رفع الحالات للمستوى الأعلى

غالبا ما يكون لدينا أثنان كومبونت أطفال أو أكثر لديهم نفس بيانات الحالة لذلك نقوم بدمج الحالة لكل منهما مع حالة الأول الذي يحتويهما

نُوصي برفع الحالة المشتركة بينها إلى أقرب عنصر أب مشترك بينها

سنُنشِئ آلة حاسبة للحرارة والتي تحسب إن كان الماء سيغلي في الدرجة المُعطاة ونعلم أن درجة حرارة غليان الماء عند 100

سنُنشِئ الآن مُكوّن الآلة الحاسبة Calculator والذي يُصيّر حقل إدخال <input> يُتيح لنا إدخال درجة الحرارة ويحتفظ بقيمتها في this.state.temperature ويمررها إلى المكون BoilingVerdict الذي يقوم بمقارنا درجة الحرارة المعطاة مع درجة حرارة الغليان و إعادة النتيجة

```
function BoilingVerdict(props) {
 if (props.celsius >= 100) {
  return <p>The water {props.celsius} would boil.</p>;
 }
 return <p>The water {props.celsius} would not boil.</p>;
}

class Calculator extends React.Component {
 constructor(props) {
  super(props);
  this.handleChange = this.handleChange.bind(this);
  this.state = {temperature: ''};
 }

 handleChange(e) {
  this.setState({temperature: e.target.value});
 }

 render() {
  const temperature = this.state.temperature;
  return (
   <fieldset>
    <legend>Enter temperature in Celsius:</legend>
    <input
     value={temperature}
     onChange={this.handleChange} />
    <BoilingVerdict celsius={temperature} />
   </fieldset>
  );
 }
}
```

المتطلّب الآخر الذي نريده إلى جانب إدخال الحرارة بالسيلزيوس هو تزويد المستخدم بحقل إدخال لدرجة الحرارة بالفهرنهايت وإبقائهما متزامنين معًا بحيث تحفظ قيمهما في حالة الأب

```
const scaleNames = {
  c: 'Celsius',
  f: 'Fahrenheit'
};

function toCelsius(fahrenheit) {
  return (fahrenheit - 32) * 5 / 9;
}

function toFahrenheit(celsius) {
  return (celsius * 9 / 5) + 32;
}

function tryConvert(temperature, convert) {
  const input = parseFloat(temperature);
  if (Number.isNaN(input)) {
    return '';
  }
  const output = convert(input);
  const rounded = Math.round(output * 1000) / 1000;
  return rounded.toString();
}

function BoilingVerdict(props) {
  if (props.celsius >= 100) {
    return <p>The water would boil.</p>;
  }
  return <p>The water would not boil.</p>;
}

class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange(e) {
    this.props.onTemperatureChange(e.target.value);
  }

  render() {
    const temperature = this.props.temperature;
    const scale = this.props.scale;
    return (
      <fieldset>
        <legend>Enter temperature in {scaleNames[scale]}:</legend>
        <input value={temperature}
               onChange={this.handleChange} />
      </fieldset>
```

```
    );
  }
}

class Calculator extends React.Component {
  constructor(props) {
    super(props);
    this.handleCelsiusChange = this.handleCelsiusChange.bind(this);
    this.handleFahrenheitChange = this.handleFahrenheitChange.bind(this);
    this.state = {temperature: '', scale: 'c'};
  }

  handleCelsiusChange(temperature) {
    this.setState({scale: 'c', temperature});
  }

  handleFahrenheitChange(temperature) {
    this.setState({scale: 'f', temperature});
  }

  render() {
    const scale = this.state.scale;
    const temperature = this.state.temperature;
    const celsius = scale === 'f' ? tryConvert(temperature, toCelsius) : temperature;
    const fahrenheit = scale === 'c' ? tryConvert(temperature, toFahrenheit) : temperature;

    return (
      <div>
        <TemperatureInput
          scale="c"
          temperature={celsius}
          onTemperatureChange={this.handleCelsiusChange} />
        <TemperatureInput
          scale="f"
          temperature={fahrenheit}
          onTemperatureChange={this.handleFahrenheitChange} />
        <BoilingVerdict celsius={parseFloat(celsius)} />
      </div>
    );
  }
}

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

لا يُهم الآن أي حقل إدخال تُعدِّل، فستُحدَّث this.state.temperature و this.state.scale الموجودة في المُكوِّن الأب Calculator حيث يأخذ أحد الحقلين القيمة التي تدخلها ويُعاد حساب قيمة الآخر بناءً عليها

فلنعد تلخيص ما يحدث عند تعديلك لحقل الإدخال :

- تستدعي React الدالة المُحدَّدة في الخاصيّة onChange كما هو الحال عند استخدام العنصر <input> في DOM في حالتنا التابع المطلوب هو handleChange الموجود في المُكوِّن TemperatureInput
- يستدعي التابع handleChange الموجود ضمن المُكوِّن TemperatureInput الدالة this.props.onTemperatureChange() مع القيمة الجديدة المطلوبة. مع العلم أنّ خاصيّات هذا المُكوِّن props بما في ذلك onTemperatureChange قد زوّدنا بها المُكوِّن الأب له وهو Calculator
- عندما صيّرنا Calculator مُسبقًا فقد حدَّد أنّ الدالة onTemperatureChange من المُكوِّن TemperatureInput ذو المقياس سيلزيوس هي نفسها التابع handleCelsiusChange الخاص بالمُكوِّن Calculator، والدالة onTemperatureChange من المُكوِّن TemperatureInput ذو المقياس فهرنهايت هي نفسها التابع handleFahrenheitChange الخاص بالمُكوِّن Calculator، لذلك يُستدعى أي من هذان التابعان اعتمادًا على حقل الإدخال الذي عدّلنا قيمته.
- بداخل هذين التابعين يطلب المُكوِّن Calculator من React أن تُعيد تصييره عن طريق استدعاء this.setState() بقيم حقول الإدخال الجديدة والمقياس الحالي لحقل الإدخال الذي عدّلناه.
- تستدعي React التابع render الخاص بالمُكوِّن Calculator لتعرف الشكل الذي ينبغي أن تكون عليه واجهة المستخدم. يُعاد حساب قيم حقول الإدخال بناءً على درجة الحرارة الحاليّة والمقياس قيد الاستخدام، تُحوَّل درجة الحرارة هنا.
- تستدعي React التابع render الخاص بكل مُكوِّن TemperatureInput مع خاصيّاتها props الجديدة المُحدَّدة عن طريق المُكوِّن Calculator، وبذلك تعرف الشكل الذي ينبغي أن تكون عليه واجهة المستخدم.
- تستدعي React التابع render الخاص بمُكوِّن BoilingVerdict، وتمرير درجة الحرارة في درجة مئوية كوسيط.
- تُحدِّث React DOM واجهة DOM لتُطابِق القيم المُدخَلَة المطلوبة، حيث يحتوي حقل الإدخال الذي عدّلناه القيمة التي أدخلناها بأنفسنا، أمّا حقل الإدخال الآخر فيُحدَّث بدرجة الحرارة بعد تحويلها للمقياس المُناسِب.

يجري كل تحديث بنفس الخطوات بحيث تبقى حقول الإدخال متزامنة.

الدروس المستفادة :

يجب أن يكون هناك "مصدر وحيد للحقيقة" لأيّة بيانات مُتغيّرة في تطبيق React. تُضاف الحالة عادةً إلى المُكوّن الذي يحتاجها للتصير أولًا، بعد ذلك إن كانت تحتاجها مُكوّنات أخرى فبإمكانك رفعها إلى أقرب مُكوِّن مشترك. وبدلًا من محاولة مزامنة الحالة بين مُكوّنات مختلفة ينبغي عليك الاعتماد على تدفق البيانات للمستويات الأدنى

يتضمّن رفع الحالة كتابة شيفرة سلسة أكثر من محاولة إجراء ربط ثنائي الاتجاه،

18

الفرق بين التركيب والوراثة

يتم توريث العناصر أو المكونات لمكونات أخرى عن طريق props.children أو props بشكل عام

ولكن لا يفضل استخدام الوراثة أبدا ونستخدم بدلا عنها التركيب

توريث عناصر الأبناء عن طريق props.children :

```
function FancyBorder(props) {
 return (
  <div className={'FancyBorder FancyBorder-' + props.color}>
   {props.children}
  </div>
 );
}

function WelcomeDialog() {
 return (
  <FancyBorder color="blue">
   <h1 className="Dialog-title">
    Welcome
   </h1>
   <p className="Dialog-message">
    Thank you for visiting our spacecraft!
   </p>
  </FancyBorder>
 );
}
```

يُمرَّر أي شيء بداخل العنصر <FancyBorder> إلى المُكوِّن FancyBorder عبر الخاصيّة children وبما أنّ المُكوِّن FancyBorder يُصيِّر {props.children} بداخل عنصر فستظهر العناصر المُمرَّرة بداخل الناتج النهائي

نستخدم التركيب في التطبيق السابق بدلا من الوراثة

```
function FancyBorder(props) {
 return (
  <div className={'FancyBorder FancyBorder-' + props.color}>
   <h1 className="Dialog-title">
    Welcome
   </h1>
   <p className="Dialog-message">
    Thank you for visiting our spacecraft!
   </p>
  </div>
 );
}

function WelcomeDialog() {
 return (
  <FancyBorder color="blue" />
 );
}
```

توريث عناصر الأبناء عن طريق props :

```
function SplitPane(props) {
  return (
    <div className="SplitPane">
      <div className="SplitPane-left">
        {props.left}
      </div>
      <div className="SplitPane-right">
        {props.right}
      </div>
    </div>
  );
}

function App() {
  return (
    <SplitPane
      left={
        <Contacts />
      }
      right={
        <Chat />
      } />
  );
}
```

إنّ عناصر React مثل <Contacts /> و <Chat /> هي مُجرّد كائنات، لذلك بإمكانك تمريرها كخاصيّات props مثل أي بيانات أخرى. قد يُذكِّرك ذلك بمفهوم المداخل (slots) في مكتبات أخرى، ولكن لا توجد حدود لما يُمكِنك تمريره كخاصيّات props في React

نستخدم التركيب في التطبيق السابق بدلا من الوراثة

```
function SplitPane(props) {
  return (
    <div className="SplitPane">
      <div className="SplitPane-left">
        <Contacts />
      </div>
      <div className="SplitPane-right">
        <Chat />
      </div>
    </div>
  );
}

function App() {
  return (
    <SplitPane />
  );
}
```

إذًا ماذا عن الوراثة ؟

نستخدم في فيسبوك آلاف مُكوّنات React ، ولم نجد أي حالة نُفضِّل فيها استخدام الوراثة.

يمنحك التركيب والخاصيّات props المرونة التي تحتاجها لتخصيص مظهر وسلوك المُكوّنات بطريقة مضبوطة وآمنة. تذكّر أنّ المُكوّنات قد تستقبل خاصيّات من محتوى مُتعدِّد، مثل القيم المبدئية، وعناصر React ، والدوال.

إن أردت إعادة استخدام بعض الوظائف بين المُكوّنات غير المُتعلّقة بواجهة المستخدم فنقترح استخراجها إلى واحدات JavaScript مُنفصِلة، حيث يُمكِن للمُكوّن أن يستورد ويستخدم الدوال والكائنات والأصناف بدون الامتداد لها عن طريق الكلمة extend.

19

تطبيقات على ما سبق

تطبيق todo لنشاء قائمة ديناميكية :

ستكون قادر على إضافة عناصر للقائمة وحذفها وبتالي يمكنك وضع مهامك اليومية في القائمة

```
export default class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = { items: [], text: '' };// وظيفة النص حفظ القيمة التي في حقل الأدخال لنقلها إلى الحالة
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
    this.handledelete = this.handledelete.bind(this);
  }

  handleChange(e) {
     this.setState({ text: e.target.value });
    }
  handledelete(e) {
    let item = [];
    for (let x=0;x<this.state.items.length;x++) {
      if (this.state.items[x].id !== e) {
        item = item.concat(this.state.items[x]);
      }
    }// أنشاء مصفوفة جديدة من دون عنصر الذي قمنا بحذفه من أجل تحديث الحالة

    this.setState(() => ({
        items: item
      }));

  }

  handleSubmit(e) {
    e.preventDefault();
    if (this.state.text.length === 0) {
      return;
```

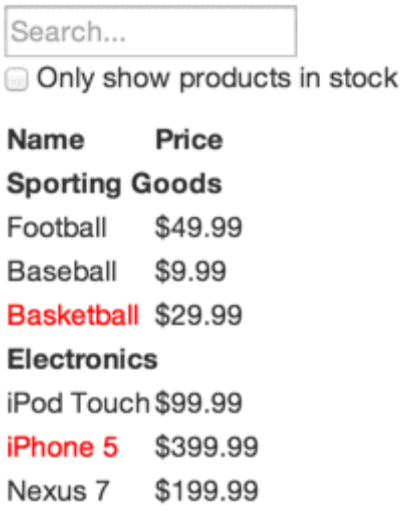
```
    }
    const newItem = {
      text: this.state.text,
      id: Date.now()
    };// وظيفتها أنشاء كائن جديد لنضيفه للحالة
    this.setState(state => ({
      items: state.items.concat(newItem),
      text: ''
    }));
  }

  render() {
    return (
      <div>
        <h3>TODO</h3>
        <ul>
        {this.state.items.map(item => (
          <li key={item.id}>
            {item.text + "  "}
            <button onClick={() => this.handledelete(item.id)}>delete</button>
          </li>
        ))}
      </ul>
        <form onSubmit={this.handleSubmit}>
          <label htmlFor="new-todo">
            What needs to be done?
          </label>
          <br/>
          <input
            id="new-todo"
            onChange={this.handleChange}
            value={this.state.text}
          />
          <button>
            Add #{this.state.items.length + 1}
          </button>
        </form>
      </div>
    );
  }

}
```

التفكير على طريقة React

واحدة من المزايا العظيمة لـ React هي كيف أنه يجعلك تفكر في التطبيقات أثناء بناءها

ابدأ بنموذج التطبيق :

تصور أننا نملك واجهة برمجة تطبيقات (API JSON) جاهزة ونموذج من المصمم.



واجهة برمجة التطبيقات ترسل بعض البيانات للواجهة ولكن البيانات غير منظمة كالآتي:

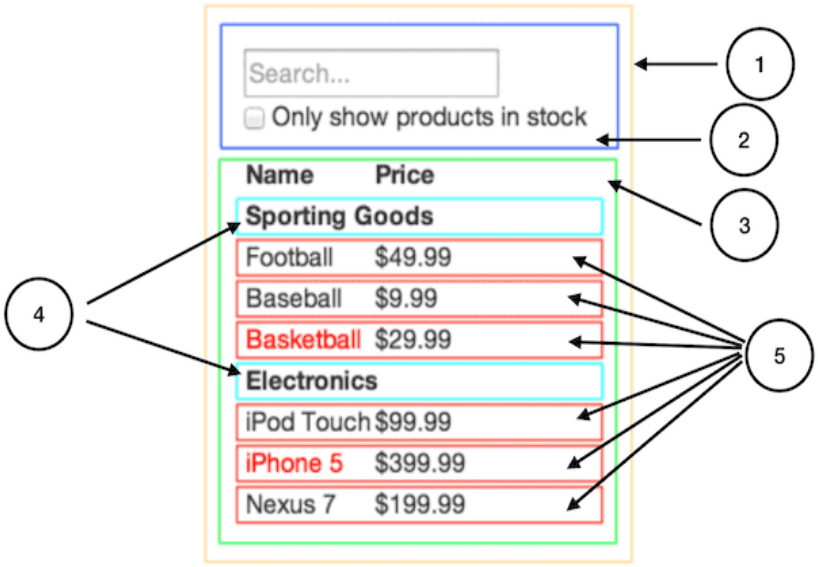
```
[
  {category: 'Sporting Goods', price: '$49.99', stocked: true, name: 'Football'},
  {category: 'Electronics', price: '$99.99', stocked: true, name: 'iPod Touch'},
  {category: 'Sporting Goods', price: '$29.99', stocked: false, name: 'Basketball'},
  {category: 'front end', price: '$1999.99', stocked: true, name: 'react'},
  {category: 'Electronics', price: '$399.99', stocked: true, name: 'iPhone 5'},
  {category: 'Sporting Goods', price: '$9.99', stocked: true, name: 'Baseball'},
  {category: 'front end', price: '$999.99', stocked: true, name: 'css'},
  {category: 'Electronics', price: '$199.99', stocked: false, name: 'Nexus 7'}
];
```

الخطوة الأولى: قم بتقسيم واجهة المستخدم (UI) لتسلسل هرمي لمكونات الواجهة (Component) :

أول شيء يجب عليك فعله هو رسم مستطيلات حول كل مكون (component) ومكون فرعي (subcomponent) في النموذج وإعطاء كل منهم اسما

ولكن كيف تعرف ما يجب أن تحدده بصفته مكونا ؟

وهي أن المكون (component) بشكل مثالي يجب أن يكون مسئولا عن فعل شيء واحد فقط، وإذا بدأ في التنامي يجب تقسيمه إلى مكونات فرعية (subcomponent) أصغر.



سنرى هنا أن لدينا خمس مكونات (components) في تطبيقنا الصغير، ولقد قمنا بالكتابة بخط عريض في كل مكون وما يمثله من بيانات :

١. FilterableProductTable باللون البرتقالي :(يحتوي المثال بكامله)
٢. SearchBar باللون الأزرق :(يستقبل ما يدخله المستخدم ***(user input)***)
٣. ProductTable باللون الأخضر :(يعرض وينقح (filter) مجموعة البيانات ***(data collection)*** طبقاً لما أدخله المستخدم ***(user input)***
٤. ProductCategoryRow باللون الفيروزي :(يعرض عنوانا (heading) لكل *فئة* ***(category)***)
٥. ProductRow باللون الأحمر :(يعرض صفا لكل *منتج* ***(product)***)

والآن بعد أن حددنا المكونات (components) في نموذج التصميم خاصتنا، لنقم بترتيبهم في تسلسل هرمي وهذا سهل، المكونات التي تظهر بداخل مكونات أخرى في النموذج يجب أن تكون ابنا (child) داخل التسلسل :

- FilterableProductTable
  - SearchBar
  - ProductTable
    - ProductCategoryRow
    - ProductRow

الخطوة الثانية: بناء نسخة ثابتة (static version) بـ(React) :

لنقم ببناء نسخة ثابتة لا تحتوي على حالة وذلك من أجل عدم تعقيد الأمور لذلك نبني مكونات الواجهة بدون حالة

والآن ونحن نملك التسلسل الهرمي للمكونات، حان وقت تنفيذ التطبيق. الطريقة السهلة هي بناء نسخة تستخدم نموذج البيانات (data model) لتصيير (renders) واجهة المستخدم (UI) ولكن بلا إمكانية للتفاعل مع التطبيق، من الأفضل فصل هذه العمليات لأن بناء نسخة ثابتة تحتاج للكثير من الكتابة دون تفكير، وإضافة التفاعلية (interactivity) تحتاج للكثير من التفكير والقليل من الكتابة، سنرى لماذا

لبناء نسخة ثابتة من التطبيق ستحتاج إلى بناء مكونات تستخدم مكونات أخرى وترسل لها البيانات باستخدام *الخاصيات* ***(props)*** ، وهي طريقة لتمرير البيانات من المكون الأب إلى المكون الابن، إذا كنت على معرفة بمبدأ *الحالة* ***(state)*** لا تستخدم الحالة **(state)** أبداً لبناء نسخة ثابتة، الحالة **(state)** تستخدم لغرض التفاعلية فقط، ما يعني أن البيانات تتغير باستمرار، وبما أن هذه النسخة ثابتة فأنت لا تحتاجها.

يمكنك البناء من الأعلى إلى الأسفل أو من الأسفل إلى الأعلى، وذلك أنه يمكنك البدء ببناء المكونات في أعلى التسلسل الهرمي (مثلا ابدأ بـ **FilterableProductTable)** أو بمكون في أسفله **(ProductRow)** ، في الأمثلة الغير معقدة من الأسهل عادة البدء من الأعلى إلى الأسفل، وفي المشاريع الأكبر من الأسهل البدء من الأسفل إلى الأعلى مع كتابة اختبارات **(tests)** خلال البناء.

مع نهاية هذه الخطوة، سيكون لديك مكتبة من المكونات القابلة لإعادة الاستخدام

سيحصل علي نموذج البيانات بصفته خاصية **(prop)** ، إذا قمت بعمل تغيير في نموذج البيانات وقمت باستدعاء الدالة **(ReactDOM.render())** مرة أخرى فإن واجهة المستخدم سيتم تحديثها

طريقة تدفق البيانات في اتجاه واحد **(one-way data flow)** الخاصة بـ **(React)** وتدعي أيضاً بـ ***(one-way binding)*** تحافظ على كل شيء سريع ووحدة **(modular)** واحدة

سنقوم بإنشاء مجلد باسم **components** وبداخله سوف ننشأ مجلدات لكل كومبونت ويكون أسم المجلد بنفس أسم الكومبونت وبداخل كل مجلد ملف الجافاسكربت ويكون أسمه باسم الكومبونت لنبدأ

ملف index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import FilterableProductTable from './components/FilterableProductTable/FilterableProductTable';

const PRODUCTS = [
  {category: 'Sporting Goods', price: '$49.99', stocked: true, name: 'Football'},
  {category: 'Electronics', price: '$99.99', stocked: true, name: 'iPod Touch'},
  {category: 'Sporting Goods', price: '$29.99', stocked: false, name: 'Basketball'},
  {category: 'front end', price: '$1999.99', stocked: true, name: 'react'},
  {category: 'Electronics', price: '$399.99', stocked: true, name: 'iPhone 5'},
  {category: 'Sporting Goods', price: '$9.99', stocked: true, name: 'Baseball'},
  {category: 'front end', price: '$999.99', stocked: true, name: 'css'},
  {category: 'Electronics', price: '$199.99', stocked: false, name: 'Nexus 7'}
];

ReactDOM.render(
  <FilterableProductTable products={PRODUCTS} />,
  document.getElementById('root')
);
```

يكون بداخله البيانات ويستدعي المكون FilterableProductTable مع تمرير البيانات له

مجلد FilterableProductTable بداخله ملف FilterableProductTable.js

```
import React from "react";
import ProductTable from "../ProductTable/ProductTable";
import SearchBar from "../SearchBar/SearchBar";
import styles from "./FilterableProductTable.module.css";
```

```
export default class FilterableProductTable extends React.Component {
 render() {
  return (
   <div className={styles.body}>
    <SearchBar />
    <ProductTable products={this.props.products} />
   </div>
  );
 }
}
```

المكون FilterableProductTable يستدعي المكون البحث SearchBar و مكون الحقول ProductTable مع تمرير البيانات ك props

المجلد SearchBar بداخله ملف SearchBar.js

```
import React from "react";

export default class SearchBar extends React.Component {
  render() {
   return (
    <form>
     <input type="text" placeholder="Search..." />
     <p>
      <input type="checkbox" />
      {' '}
      Only show products in stock
     </p>
    </form>
   );
  }
 }
```

ليس له وظيفة حاليا في هذه الخطوة ولكن وظيفته المستقبلية هي البحث داخل البيانات

مجلد ProductTable بداخله ملف ProductTable.js

```
import React from "react";
import ProductCategoryRow from "../ProductCategoryRow/ProductCategoryRow";
import ProductRow from "../ProductRow/ProductRow";

export default class ProductTable extends React.Component {
  render() {
   const rows = [];

   let nameCategory = [];
   for (let x=0;x<this.props.products.length;x++) {
    if (nameCategory.includes(this.props.products[x].category) === false) {
```

```
      nameCategory.push(this.props.products[x].category);
    }
  }

  // console.log(nameCategory);

  for (let x=0;x<nameCategory.length;x++) {

   rows.push(
    <ProductCategoryRow
     category={nameCategory[x]}
     key={nameCategory[x]} />
   );
   for (let y=0;y<this.props.products.length;y++) {
    if (nameCategory[x] ===this.props.products[y].category) {
     rows.push(
      <ProductRow
       product={this.props.products[y]}
       key={this.props.products[y].name} />
     );
    }
   }

  }


  return (
   <table>
    <thead>
     <tr>
      <th>Name</th>
      <th>Price</th>
     </tr>
    </thead>
    <tbody>{rows}</tbody>
   </table>
  );
 }
}
```

وظيفة الحلقة الأولى هي البحث في this.props.products[x].category والحصول على أسماء category من أجل التنصيف المنتجات وتخزينها في المصفوفة nameCategory لأن البيانات قد تكون غير منظمة وهكذا أصبح لدينا أسماء المنتجات في البيانات

وظيفة الحلقة الثانية هي تخزين البيانات في المصفوفة rows والتي لها وظيفتين

الوظيفة الأولى هي استدعاء المكون ProductCategoryRow لإنشاء رأس لأسم المنتج ويتم تمرير له أسم المنتج category والمفتاح باسم المنتج category وبداخل هذه الحلقة حلقة ثانية

وظيفة الثانية هي استدعاء المكون ProductRow الذي يعرض المنتجات الذين ينتمون لل category ويتم تخزينها في المصفوفة rows ويمرر له المنتجات وأسم المنتج نفسه كمفتاح

وهكذا أصبح لدينا مصفوفة rows منظمة بحيث تضمن رأس المنتج category وبعده البيانات التي تنتمي أليه وهكذا

وبعدها ننشئ جدول ونمرر له rows

المجلد ProductCategoryRow بداخله ملف ProductCategoryRow.js

```
import React from "react";

export default class ProductCategoryRow extends React.Component {
  render() {
    const category = this.props.category;
    return (
      <tr>
        <th colSpan="2">
          {category}
        </th>
      </tr>
    );
  }
}
```

وظيفة المكون بناء رأس الجدول الخاص باسم المنتج category

المجلد ProductRow بداخله ملف ProductRow.js

```
import React from "react";

export default class ProductRow extends React.Component {
  render() {
    const product = this.props.product;
    const name = product.stocked ?
      product.name :
      <span style={{color: 'red'}}>
        {product.name}
      </span>;

    return (
      <tr>
        <td>{name}</td>
        <td>{product.price}</td>
      </tr>
    );
  }
}
```

وظيفة المكون هي عرض المنتجات وتلون المنتجات بالأحمر التي يكون فيها product.stocked = false أي المنتج غير متوفر بالمستودع

الخطوة الثالثة: تحديد الحد الأدنى (ولكن المكتمل) الممثل لحالة واجهة المستخدم :

لجعل واجهة المستخدم تفاعلية ستحتاج للقدرة على عمل تغييرات فى نموذج البيانات الخاص بتطبيقك، (React) تجعل هذا سهلاً باستخدام الحالة (state)

لبناء تطبيقك بشكل صحيح، ستحتاج أولاً للتفكير فى الحد الأدنى من الحالة القابلة للتغيير (mutable state) التي سيحتاجها التطبيق، المفتاح هنا هو حدد الحد الأدنى قدر الإمكان الممثل للحالة التي يحتاجها تطبيقك بحيث لا نكرر الحالة في مكون أخر

فكر في كل أجزاء البيانات في مثالنا، لدينا :

- القائمة الأصلية للمنتجات (تمرر البيانات ولا تتغير من قبل المستخدم لا تحتاج لحالة)
- كلمة البحث التي أدخلها المستخدم (تتغير من قبل المستخدم وتحتاج لحالة)
- حالة الـ(checkbox) (تتغير من قبل المستخدم وتحتاج لحالة)
- قائمة المنتجات المنقحة (تمرر البيانات ولا تتغير من قبل المستخدم لا تحتاج لحالة)

دعنا نحدد أي منهم تصلح حالةً، فقط اسأل ثلاث أسئلة عن كل جزء من البيانات :

١. هل يتم تمريرها من مكون أب بصفتها خاصية(props) ؟ إذا كان نعم، فمن المحتمل هي ليست حالة.
٢. هل هي ثابتة لاتتغير مع مرور الزمن؟ إذا كان نعم، فمن المحتمل هى ليست حالة.
٣. هل يمكنك حسابها بناء على حالة أو خاصية (props) أخرى فى هذا المكون؟ إذا كان نعم، فمن المحتمل هي ليست حالة.

القائمة الأصلية للمنتجات يتم تمريرها بصفتها خاصية (props) إذاً فهي ليست حالة، كلمة البحث والـ (checkbox) يتضح أنهم حالة حيث أنهم يتغيرون مع الزمن ولا يمكن حسابهم من أي شيء، وأخيراً القائمة المنقحة للمنتجات ليست حالة لأنه يمكن حسابها من دمج القائمة الأصلية للمنتجات مع كلمة البحث وحالة الـ(checkbox)

وأخيراً الحالة هي :

- كلمة البحث التى أدخلها المستخدم
- حالة الـ(checkbox)

الخطوة الرابعة: حدد أين يجب أن تكون الحالة :

حسناً لقد حددنا ماهو الحد الأدنى للحالة، في الخطوة التالية سنحدد ماهو المكون المسئول عن تحويل (mutates) أو *يملك* هذه الحالة

تذكر: في (React) تتدفق البيانات في اتجاه واحد (one-way flow) إلى أسفل التسلسل الهرمى للمكونات، قد لا يكون واضحا في هذه اللحظة أي مكون يجب أن يملك أية حالة وهذه غالباً أكثر الأجزاء تحدياً للفهم على القادمين الجدد

لذلك اتبع هذه الخطوات للكشف لكل جزء من الحالة في تطبيقك :

- حدد كل مكون يقوم بتصيير (render) شيء ما بناءً على هذه الحالة.
- ابحث عن مكون مشترك ليملك هذه الحالة (مكون واحد أعلى فى التسلسل الهرمي من كل المكونات التي تحتاج لهذه الحالة).
- إما المكون المشترك أو مكون آخر أعلى فى التسلسل الهرمي يجب أن يملك هذه الحالة.
- إذا لم تجد مكونا يصلح لأن يملك هذه الحالة، أنشئ واحداً جديداً فقط ليملك هذه الحالة وأضفه في مكان ما في التسلسل الهرمى أعلى المكون المشترك.

لنتبع تلك الإستراتيجية في تطبيقنا:

- المكون ProductTable يحتاج لتنقيح قائمة المنتجات بناء على الحالة والمكون SearchBar يحتاج لإظهار كلمة البحث وحالة الـ(checkbox).
- المكون المشترك المالك للحالة هو FilterableProductTable
- نظرياً من المنطقي أن تتواجد كلمة البحث وقيمة الـ (checkbox) في المكون FilterableProductTable

رائع، لقد قررنا أن الحالة ستكون في المكون FilterableProductTable أولاً أضف this.state (instance property) {filterText: '', inStockOnly: false}= للـ constructor للمكون FilterableProductTable لتكون الحالة الإبتدائية للتطبيق ثم مرر الحالتين filterText و inStockOnly للمكونين ProductTable و SearchBar بصفتها خاصيات (props) وفي النهاية، استخدم هذه الخاصيات (props) لتنقيح صفوف المنتجات في المكون ProductTable وضع القيم للـ (form fields) في المكون SearchBar.

هكذا تم أنشاء الحالة ولكنها في أتجاه واحد وبتالي حقول الإدخال لا يمكنها تغيير الحالة في FilterableProductTable لذلك نضيف تدفق عكسي لتتمكن من تغيير الحالة في FilterableProductTable

الخطوة الخامسة: أضف التدفق العكسي للبيانات :

إذا حاولت الكتابة أو الضغط على الـ (checkbox) بالإصدار الحالي للتطبيق سترى أن React سيتجاهل ذلك، وذلك مقصود حيث أننا قمنا بوضع قيمة الخاصية (value) لل (input) لتكون دائماً مساوية للحالة التى تم تمريرها من المكون FilterableProductTable

لنفكر بما نريد أن يحدث، نريد التأكد كلما قام المستخدم بتغيير الـ (form) يتم تحديث الحالة لإظهار ما أدخله المستخدم وبما أن المكونات يجب أن تغير الحالة الخاصة بها فقط، المكون FilterableProductTable سيمرر الدالة (callback) للمكون SearchBar والتى سيتم استدعائها أينما وجب تحديث الحالة، يمكننا استخدام الحدث (onChange event) على الـ (inputs) لنعرف ذلك، الدالة (callback) التي تم تمريرها بواسطة المكون FilterableProductTable تقوم باستدعاء setState() ويتم تحديث التطبيق

ليكون التطبيق النهائي على الشكل التالي :

ملف index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import FilterableProductTable from './components/FilterableProductTable/FilterableProductTable';

const PRODUCTS = [
  {category: 'Sporting Goods', price: '$49.99', stocked: true, name: 'Football'},
  {category: 'Electronics', price: '$99.99', stocked: true, name: 'iPod Touch'},
  {category: 'Sporting Goods', price: '$29.99', stocked: false, name: 'Basketball'},
  {category: 'front end', price: '$1999.99', stocked: true, name: 'react'},
  {category: 'Electronics', price: '$399.99', stocked: true, name: 'iPhone 5'},
  {category: 'Sporting Goods', price: '$9.99', stocked: true, name: 'Baseball'},
  {category: 'front end', price: '$999.99', stocked: true, name: 'css'},
  {category: 'Electronics', price: '$199.99', stocked: false, name: 'Nexus 7'}
];
```

```
ReactDOM.render(
 <FilterableProductTable products={PRODUCTS} />,
 document.getElementById('root')
);
```

يكون بداخله البيانات ويستدعي المكون FilterableProductTable مع تمرير البيانات له

مجلد FilterableProductTable بداخله ملف FilterableProductTable.js

```
import React from "react";
import ProductTable from "../ProductTable/ProductTable";
import SearchBar from "../SearchBar/SearchBar";
import styles from "./FilterableProductTable.module.css";

export default class FilterableProductTable extends React.Component {
 constructor(props) {
  super(props);
  this.state = {
   filterText: '',
   inStockOnly: false
  };
                    تم أنشاء الحالة و إعطاء قيمة إبتدائية filterText لحقل البحث وقيمة إبتدائية inStockOnly لحقل checkbox
  this.handleFilterTextChange = this.handleFilterTextChange.bind(this);
  this.handleInStockChange = this.handleInStockChange.bind(this);
 }

 handleFilterTextChange(filterText) {
  this.setState({
   filterText: filterText
  });
 }
                                                    وظيفته تغيير قيمة filterText عندما يتم استدعائه
 handleInStockChange(inStockOnly) {
  this.setState({
   inStockOnly: inStockOnly
  })
 }
                                                    وظيفته تغيير قيمة inStockOnly عندما يتم استدعائه
 render() {
  return (
   <div className={styles.body}>
    <SearchBar
     filterText={this.state.filterText}
     inStockOnly={this.state.inStockOnly}
     onFilterTextChange={this.handleFilterTextChange}
     onInStockChange={this.handleInStockChange}
    />
                                                    يتم تمرير الحالات الإبتدائية و التوابع لتعمل تدفق عكسي
    <ProductTable
     products={this.props.products}
```

```
        filterText={this.state.filterText}
        inStockOnly={this.state.inStockOnly}
      />
```
يتم تمرير البيانات والحالات ليتم مقارنة الحالات وتعرض البيانات حسب المقارنة
```
    </div>
  );
 }
}
```

المجلد SearchBar بداخله ملف SearchBar.js

```
import React from "react";

export default class SearchBar extends React.Component {
 constructor(props) {
  super(props);
  this.handleFilterTextChange = this.handleFilterTextChange.bind(this);
  this.handleInStockChange = this.handleInStockChange.bind(this);
 }
```
يتم تعريف توابع ليتم استخدامها في حقول و وظيفة هذه التوابع تعمل تدفق عكسي للبيانات
```
 handleFilterTextChange(e) {
  this.props.onFilterTextChange(e.target.value);
 }
```
وظيفة التابع هو تمرير قيمة حقل البحث إلى تابع handleFilterTextChange في المكون FilterableProductTable وهكذا يعمل التدفق العكسي و تتغيير الحالة
```
 handleInStockChange(e) {
  this.props.onInStockChange(e.target.checked);
 }
```
وظيفة التابع هو تمرير قيمة حقل checkbox إلى تابع handleInStockChange في المكون FilterableProductTable وهكذا يعمل التدفق العكسي و تتغيير الحالة
```
  render() {
   return (
    <form>
     <input
      type="text"
      placeholder="Search..."
      value={this.props.filterText}
      onChange={this.handleFilterTextChange}
     />
```
يتم وضع الحالة الأبتدائية filterText التي في المكون FilterableProductTable و أي تغيير بقيمة الحقل يستدعي التابع handleFilterTextChange
```
     <p>
     <input
      type="checkbox"
      checked={this.props.inStockOnly}
      onChange={this.handleInStockChange}
     />
```
يتم وضع الحالة الأبتدائية inStockOnly التي في المكون FilterableProductTable و أي تغيير بقيمة الحقل يستدعي التابع handleInStockChange

```
        {' '}
        Only show products in stock
      </p>
    </form>
  );
 }
}
```

مجلد ProductTable بداخله ملف ProductTable.js

```
import React from "react";
import ProductCategoryRow from "../ProductCategoryRow/ProductCategoryRow";
import ProductRow from "../ProductRow/ProductRow";

export default class ProductTable extends React.Component {
  render() {
    const filterText = this.props.filterText;
    const inStockOnly = this.props.inStockOnly;

    const rows = [];

    let nameCategory = [];
    for (let x=0;x<this.props.products.length;x++) {
      if (nameCategory.includes(this.props.products[x].category) === false) {
        nameCategory.push(this.props.products[x].category);
      }
    }

    for (let x=0;x<nameCategory.length;x++) {


      if (filterText === "") {
        rows.push(
          <ProductCategoryRow
            category={nameCategory[x]}
            key={nameCategory[x]} />
        );
      }

      for (let y=0;y<this.props.products.length;y++) {

        if (this.props.products[y].name.indexOf(filterText) === -1) {
          continue;
        }

        if (inStockOnly && !this.props.products[y].stocked) {
          continue;
        }
```

```
      if (nameCategory[x] ===this.props.products[y].category) {
        rows.push(
          <ProductRow
            product={this.props.products[y]}
            key={this.props.products[y].name} />
        );
      }
    }

  }


  return (
    <table>
      <thead>
        <tr>
          <th>Name</th>
          <th>Price</th>
        </tr>
      </thead>
      <tbody>{rows}</tbody>
    </table>
  );
 }
}
```

وظيفة الحلقة الأولى هي البحث في this.props.products[x].category والحصول على أسماء category من أجل التنصيف المنتجات وتخزينها في المصفوفة nameCategory لأن البيانات قد تكون غير منظمة وهكذا أصبح لدينا أسماء المنتجات في البيانات

وظيفة الحلقة الثانية هي تخزين البيانات في المصفوفة rows والتي لها وظيفتين

الوظيفة الأولى هي استدعاء المكون ProductCategoryRow لإنشاء رأس لأسم المنتج ويتم تمرير له أسم المنتج category والمفتاح باسم المنتج category وبداخل هذه الحلقة حلقة ثانية

وظيفة الثانية هي استدعاء المكون ProductRow الذي يعرض المنتجات الذين ينتمون لل category ويتم تخزينها في المصفوفة rows ويمرر له المنتجات وأسم المنتج نفسه كمفتاح

وتم إنشاء شروط حسب الحالة في FilterableProductTable ويتم عرض البيانات على حسب الحالة

وهكذا أصبح لدينا مصفوفة rows منظمة بحيث تضمن رأس المنتج category وبعده البيانات التي تنيتمي أليه وهكذا

وبعدها ننشئ جدول ونمرر له rows

المجلد ProductCategoryRow بداخله ملف ProductCategoryRow.js

```
import React from "react";

export default class ProductCategoryRow extends React.Component {
  render() {
    const category = this.props.category;
```

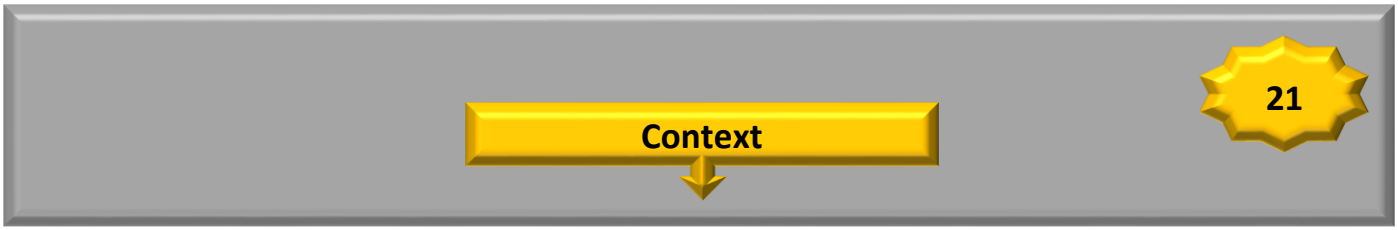
```
    return (
      <tr>
        <th colSpan="2">
          {category}
        </th>
      </tr>
    );
  }
}
```

وظيفة المكون بناء رأس الجدول الخاص باسم المنتج category

المجلد ProductRow بداخله ملف ProductRow.js

```
import React from "react";

export default class ProductRow extends React.Component {
  render() {
    const product = this.props.product;
    const name = product.stocked ?
      product.name :
      <span style={{color: 'red'}}>
        {product.name}
      </span>;

    return (
      <tr>
        <td>{name}</td>
        <td>{product.price}</td>
      </tr>
    );
  }
}
```

وظيفة المكون هي عرض المنتجات وتلون المنتجات بالأحمر التي يكون فيها product.stocked = false أي المنتج غير متوفر بالمستودع



كنا في السابق نمرر ال props من الأب إلى الطفل و الطفل نفسه يمررها إلى طفل أخر وهكذا وهذه الطريقة معقدة و متعبة

لكن أتاح لنا السياق context تمرير البيانات من الأب إلى الأطفال مباشرة من دون تمرير البيانات عبر الأطفال

يكفي تعريف البيانات context في الأب وتغليفها لمكون الأبن ويمكن استخدام البيانات في أي طفل في الشجرة دون الحاجة إلى تمرير البيانات بين الأطفال فقط نهيئه ونستعمله

يُزوِّدنا السياق (Context) بطريقة لتمرير البيانات عبر شجرة المُكوّنات دون الحاجة لتمرير الخاصيّات props يدويًا من الأعلى إلى الأسفل في كل مستوى

متى نستخدم السياق context :

يكون السياق مُصمَّمًا لمشاركة البيانات التي تُعتبر عامّة (global) لشجرة مكوّنات React ، مثل المستخدم قيد المصادقة حاليًا، أو القالب، أو تفضيلات اللغة

أنشاء React.createContext:

يتم أنشاء context خارج المكون و إعطائه قيمة أفتراضية عبر

```
const nameContext = React.createContext(defaultValue);
```

تغليف الطفل مع تغيير القيمة nameContext.Provider :

```
<nameContext.Provider value={/* some value */}>
</ nameContext.Provider>
```

استخدام context :

يتم استخدام context من خلال تهيئة القيمة ClassName.contextType وهناك طريقتين

الطريقة الأولى : تهيئة خارج الصنف من خلال

```
MyClassName.contextType = nameContext;
```

وبعدها يمكننا استخدام context داخل دوال الصنف من خلال this.context

```
class MyClass extends React.Component {
  componentDidMount() {
    let value = this.context;
    /* perform a side-effect at mount using the value of MyContext */
  }
  componentDidUpdate() {
    let value = this.context;
    /* ... */
  }
  componentWillUnmount() {
    let value = this.context;
    /* ... */
  }
  render() {
    let value = this.context;
    /* render something based on the value of MyContext */
  }
```

```
}
MyClass.contextType = MyContext;
```

الطريقة الثانية : تهيئة داخلية للصنف من خلال static و الاستعمال من خلال this.context

```
class MyClass extends React.Component {
  static contextType = MynameContext;
  render() {
    let value = this.context;
    /* value صيّر شيئًا بناءً على القيمة */
  }
}
```

ملاحظة : يتم تغليف و استدعاء context في مكونات الأصناف فقط ويمكنك الاستدعاء بمكون دالة من خلال Consumer بدون الجاجة إلى contextType مثل

```
<nameContext.Consumer>
  {(value) => (/* صيّر شيئًا بناء على قيمة السياق */) }
</nameContext.Consumer>
```

مثال يشرح السابق :

```
const ThemeContext = React.createContext('light');// إنشاء

class App extends React.Component {
  render() {

    return (
      <ThemeContext.Provider value="dark">
        <Toolbar />
      </ThemeContext.Provider>// تغليف مع قيمة جديدة
    );
  }
}

function Toolbar() {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}

class ThemedButton extends React.Component {

  static contextType = ThemeContext;// تهيئة
  render() {
    return <Button theme={this.context} />;// استدعاء
  }
```

```
}
```

مثال أخر وهو عبارة عن كبسة تغيير التنسيق بكل ضغطة

نقوم بأنشاء context بملف خارجي ثم نستدعيها حيث أردنا

theme-context.js

```
import React from "react";

export const themes = {
  light: {
    foreground: '#000000',
    background: '#eeeeee',
  },
  dark: {
    foreground: '#ffffff',
    background: '#222222',
  },
};

 export const ThemeContext = React.createContext(
  themes.dark // default value
 );
```

App.js

```
import React from "react";
import {ThemeContext, themes} from './theme-context';
import Cont from "./Cont";
import Toolbar from "./Toolbar";


export default class App extends React.Component {
 constructor(props) {
  super(props);
  this.state = {
    theme: themes.light,
  };

  this.toggleTheme = () => {
    this.setState(state => ({
      theme:
        state.theme === themes.dark
          ? themes.light
          : themes.dark,
    }));
  };
 }
```

```
 render() {

  return (
   <div>
    <ThemeContext.Provider value={this.state.theme}>
     <Cont />
     <Toolbar changeTheme={this.toggleTheme} />
    </ThemeContext.Provider>
   </div>
  );
 }
}
```

حيث يتم استدعاء بيانات context من theme-context وتخزينها في state للمكون App ثم يتم تغليف المكونات التي سوف نستدعيها بقيمة ال state وبتالي أي تغيير في ال state سوف يغيير قيمة التغليف وتتغير القيمة بكل المكونات المستدعات المغلفة

Cont.js

```
import React from "react";
import {ThemeContext} from './theme-context';

export default class Cont extends React.Component {

  render() {
    let theme = this.context;
    return (
      <div style={{backgroundColor: theme.background,color: theme.foreground}}>
        <h1>ammar</h1>
      </div>
    );
  }

}

Cont.contextType = ThemeContext;
```

أي تغيير في state الذي بداخل المكون App سوف تتغير قيمة الستايل لان غلفنا المكون ب state المكون App

Tollbar.js

```
import React from "react";
import ThemedButton from './ThemedButton';

export default function Toolbar(props) {
  return (
   <ThemedButton onClick={props.changeTheme}>
    Change Theme
```

```
    </ThemedButton>
  );
}
```

وظيفتها استدعاء مكون وأعطائه الدالة التي في state في المكون App لتغيير الحالة عندما يفعل الدالة وبتالي تغيير التغليف وبتالي تغيير contaxt

ThemedButton.js

```
import React from "react";
import {ThemeContext} from './theme-context';

export default class ThemedButton extends React.Component {
 render() {
   let props = this.props;
   let theme = this.context;
   return (
     <button
       {...props}// لإضافة اونكليك المسؤولة عن تغيير الحالة
       style={{backgroundColor: theme.background,color: theme.foreground}}
     />
   );
 }
}
ThemedButton.contextType = ThemeContext;
```

تحديث context وعرضه داخل مكون دالة من خلال nameContext.Consumer :

```
<nameContext.Consumer>
 {(value) => (/* صيّر شيئًا بناء على قيمة السياق */) }
</nameContext.Consumer>
```

تتطلب الخاصية Consumer دالةً على أنَّها ابنٌ .إذ تستقبل هذه الدالة قيمة السياق الجديد وتعيد ال React الوسيط value المُمرَّر إلى الدالة سيكون مساويًا إلى قيمة الخاصية value لأقرب مزود (Provider) لغيير قيمة التغلف لهذا السياق في الشجرة أعلاه. إن لم يكن هنالك مزود (Provider) لهذا السياق أعلاه، فسيكون الوسيط value مساويًا إلى القيمة defaultValue التي مُرِّرت إلى createContext()

كما يقوم بعرض context بدون الحاجة لتعرف مكون صنف بل مكون دالة فقط أي لم نعد بحاجة إلى contextType التي تدخل context إلى داخل مكون الصنف وبدون الحاجة لتحديثه بل فقط عرضه

يستخدم ويستدعي دوما أسفل التغليف Provider

التحكم بالذي سيعرضه context من خلال nameContext.displayName :

كائن السياق يقبل خاصية displayName التي نساويها مع سلسلة و يستخدم هذه السلسلة لتحديد ما يجب عرضه للسياق .

```
const nameContext = React.createContext(/* some value */);
nameContext.displayName = 'MyDisplayName';
```

```
<nameContext.Provider> // "MyDisplayName.Provider" in DevTools
<nameContext.Consumer> // "MyDisplayName.Consumer" in DevTools
```

مثال عن تحديث context وعن عرض context داخل مكون دالة بدون الحاجة إلى contexttype التي تدخل context إلى داخل مكون الصنف

theme-context.js

```
import React from "react";

export const themes = {
  light: {
    foreground: '#000000',
    background: '#eeeeee',
  },
  dark: {
    foreground: '#ffffff',
    background: '#222222',
  },
 };

export const ThemeContext = React.createContext({
 theme: themes.dark,
 toggleTheme: () => {},
});
```

App.js

```
import React from "react";
import {ThemeContext, themes} from './theme-context';
import Cont from "./Cont";

export default class App extends React.Component {
 constructor(props) {
  super(props);

  this.toggleTheme = () => {
   this.setState(state => ({
    theme:
     state.theme === themes.dark
      ? themes.light
      : themes.dark,
   }));
  };

  this.state = {
   theme: themes.light,
   toggleTheme: this.toggleTheme,
  };
```

```
  }

  render() {

    return (
      <ThemeContext.Provider value={this.state}>
        <Cont />
      </ThemeContext.Provider>
    );
  }
}
```

يتم تعريف حالة داخل المكون App وبداخله حاله فيها شيئين التنسيق ودالة تغيير الحالة نفسها بعدها يتم التغليف المكون Cont بهذه الحالة التي تحتوي على التنسيق والدالة التي تغيير الحالة

Cont.js

```
import React from "react";
import ThemeTogglerButton from "./ThemeTogglerButton";
import {ThemeContext} from './theme-context';

export default function Cont (){
  return (
    <div>
      <ThemeContext.Consumer>
        {(theme) =>
            (<div style={{backgroundColor: theme.theme.background ,color: theme.theme.foreground}}>
          AMMAR QASSAB
        </div>) }
      </ThemeContext.Consumer>
      <ThemeTogglerButton />
    </div>
);
}
```

الوظيفة الأولى لمكون Cont هي عرض context في دالة بدون الحالجة لل contextType التي تقوم بتهئة context داخل مكونات الصنف

الوظيفة الثانية هي استدعاء المكون ThemeTogglerButton

ThemeTogglerButton.js

```
import React from "react";
import {ThemeContext} from './theme-context';

export default function ThemeTogglerButton() {

  return (
    <ThemeContext.Consumer>
```

```
    {({theme, toggleTheme}) => (
      <button
        onClick={toggleTheme}
        style={{backgroundColor: theme.background ,color: theme.foreground}}>
        Toggle Theme
      </button>
    )}
  </ThemeContext.Consumer>
 );
}
```

وظيفة المكون هي استخدام Consumer لعرض زر وتزويدها ب context التنسيق و الدالة التي غلفناها في المكون App المسؤولة عن تغيير الحالة

وبتالي أي ضغطة على الزر سوف تغيير التغليف بسبب تغيير الحالة في المكون App وبتالي تغيير context في جميع المكونات المغلفة بالحالة

خلاصة context :

أنشئ context بملف خارجي ثم مرره إلى المكون الرئيسي مثل App ثم حزنه بالحالة وداخل هذه الحالة أنشئ دالة لتغيير الحالة ثم غلف المكونات بكل الحالة

استخدام سياقات context متعددة :

لإبقاء قدرة السياق على إعادة التصيير بشكل سريع، تحتاج React إلى جعل كل مستهلك سياق على شكل عقدة منفصل في الشجرة:

```
// Theme context, default to light theme
const ThemeContext = React.createContext('light');

// Signed-in user context
const UserContext = React.createContext({
 name: 'Guest',
});

class App extends React.Component {
 render() {
  const {signedInUser, theme} = this.props;

  // App component that provides initial context values
  return (
   <ThemeContext.Provider value={theme}>
    <UserContext.Provider value={signedInUser}>
     <Layout />
    </UserContext.Provider>
   </ThemeContext.Provider>
  );
 }
}

function Layout() {
 return (
```

```
    <div>
      <Sidebar />
      <Content />
    </div>
  );
}

// A component may consume multiple contexts
function Content() {
  return (
    <ThemeContext.Consumer>
      {theme => (
        <UserContext.Consumer>
          {user => (
            <ProfilePage user={user} theme={theme} />
          )}
        </UserContext.Consumer>
      )}
    </ThemeContext.Consumer>
  );
}
```

إن كانت قيمة سياقين أو أكثر مستخدمة معًا، فقد ترغب بالنظر إلى إنشاء مكون خاصية التصيير الخاص بك والذي يزودك بكليهما معًا.

محاذير :

دوما أحتفظ ببيانات context في state المكون الأب الذي يغلف المكونات المستدعات لكي لا تحدث تصييرات إضافية عندما يتحدث المكون الأب عندما تتغير القيمة

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: nameContext
    };
  }

  render() {
    return (
      <MyContext.Provider value={this.state.value}>
        <Toolbar />
      </MyContext.Provider>
    );
  }
}
```

22

حدود الأخطاء

لا يجب أن يؤدّي خطأ JavaScript الحاصل في جزء من واجهة المستخدم إلى تعطيل كامل التطبيق

ولحل هذه المشكلة لمستخدمي React ، قدّمت React في الإصدار ١٦ مفهومًا جديدًا وهو حد الخطأ (error boundary)

حدود الأخطاء هي مكوّنات في React والتي تلتقط أخطاء JavaScript في أي مكان من شجرة المكوّنات الأبناء لها، وتُسجِّل هذه الأخطاء، وتعرض واجهة مستخدم بديلة وذلك بدلًا من عرض شجرة المكوّنات التي انهارت. تلتقط حدود الأخطاء هذه الأخطاء خلال التصيير، وفي توابع دورة حياة المكوّن، وفي الدوال البانية لكامل الشجرة الموجودة تحتها.

ملاحظة : لا تلتقط حدود الأخطاء أخطاءً من أجل :

- معالجات الأحداث
- الشيفرة غير المتزامنة (مثل ردود نداء setTimeout أو requestAnimationFrame)
- التصيير من جانب الخادم .
- الأخطاء المرميّة من قبل حد الخطأ نفسه (بدلًا من أخطاء المكوّنات الأبناء له).

تُصبِح مكوّنات الأصناف حدودًا للأخطاء إن عرّفت تابعًا جديدًا لدورة الحياة يُدعى

static getDerivedStateFromError() أو componentDidCatch()

استعمل static getDerivedStateFromError() لعرض واجهة مستخدم بها أخطاء.

واستعمل componentDidCatch() لتسجيل معلومات عن الخطأ

```
class ErrorBoundary extends React.Component {
 constructor(props) {
  super(props);
  this.state = { hasError: false };
 }

 static getDerivedStateFromError(error) {
  //   عرض واجهة مستخدم بديلة
  return { hasError: true };
 }
 componentDidCatch(error, errorInfo) {
  //     تستطيع تسجيل الخطأ إلى خدمة التبليغ عن الأخطاء
  logErrorToMyService(error, errorInfo);
 }
 render() {
  if (this.state.hasError) {
    //       تستطيع تصيير أي واجهة مستخدم بديلة مخصصة
   return <h1>حدث خطأ ما.</h1>;
  }
  return this.props.children;
 }
```

```
}
```

بعدها تستطيع استخدامها بصفتها مكوّنات اعتياديّة حيث تغلف المكون الذي قد يحدث خطأ

```
<ErrorBoundary>
 <MyWidget />
</ErrorBoundary>
```

مثال عن التعامل مع الأخطاء

```
class ErrorBoundary extends React.Component {
 constructor(props) {
  super(props);
  this.state = { error: null, errorInfo: null };
 }

 componentDidCatch(error, errorInfo) {
  // Catch errors in any components below and re-render with error message
  this.setState({
   error: error,
   errorInfo: errorInfo
  })
  // You can also log error messages to an error reporting service here
 }

 render() {
  if (this.state.errorInfo) {
   // Error path
   return (
    <div>
     <h2>Something went wrong.</h2>
     <details style={{ whiteSpace: 'pre-wrap' }}>
      {this.state.error && this.state.error.toString()}
      <br />
      {this.state.errorInfo.componentStack}
     </details>
    </div>
   );
  }
  // Normally, just render children
  return this.props.children;
 }
}

class BuggyCounter extends React.Component {
 constructor(props) {
  super(props);
  this.state = { counter: 5 };
  this.handleClick = this.handleClick.bind(this);
 }
```

```
  handleClick() {
    this.setState(({counter}) => ({
      counter: counter + 1
    }));
  }

  render() {
    if (this.state.counter === 5) {
      // Simulate a JS error
      throw new Error('I crashed!');
    }
    return <h1 onClick={this.handleClick}>{this.state.counter}</h1>;
  }
}

function App() {
  return (
    <div>
      <p>
        <b>
          This is an example of error boundaries in React 16.
          <br /><br />
          Click on the numbers to increase the counters.
          <br />
          The counter is programmed to throw when it reaches 5. This simulates a JavaScript error in a
component.
        </b>
      </p>
      <hr />
      <ErrorBoundary>
        <p>These two counters are inside the same error boundary. If one crashes, the error boundary will
replace both of them.</p>
        <BuggyCounter />
        <BuggyCounter />
      </ErrorBoundary>
      <hr />
      <p>These two counters are each inside of their own error boundary. So if one crashes, the other is
not affected.</p>
      <ErrorBoundary><BuggyCounter /></ErrorBoundary>
      <ErrorBoundary><BuggyCounter /></ErrorBoundary>
    </div>
  );
}



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

عندما نضغط على الرقم سيؤدي إلى زيادة الرقم بمقدار واحد وعندما نصل إلى الرقم 5 سيؤدي إلى خطأ في المكون وبتالي عرض مكون الخطأ البديل

لاحظ أنّ حدود الأخطاء تَلتقط فقط الأخطاء في المكوّنات التي تقع تحتها في شجرة المكوّنات فلا تستطيع التقاط خطأ موجود ضمنه

أين نضع حدود الأخطاء :

تستطيع وضع حدود الأخطاء أينما شئت. فقد تضعها في المكوّنات ذات المستوى الأعلى لعرض رسالة "حدث خطأ ما" للمستخدمين، مثلما تتعامل أُطُر عمل من طرف الخادم مع الانهيار. بإمكانك أيضًا تغليف الأدوات الذكية (widgets) ضمن حدود أخطاء لكي لا تؤدي لانهيار كامل التطبيق معها.

ماذا عن معالجات الأحداث Event Handlers ؟

إن احتجت لالتقاط الأخطاء بداخل مُعالِج للأحداث، فاستخدم الجملة الاعتيادية في JavaScript وهي: try / catch

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { error: null };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    try {
      // فعل أي شيء قد يرمي خطأ
   } catch (error) {
      this.setState({ error });
    }
  }

  render() {
    if (this.state.error) {
      return <h1>التقط خطأ.</h1>
    }
    return <button onClick={this.handleClick}>انقر هنا</button>
  }
}
```

23

استخدام الأجزاء (Fragments)

تحتاج الريأكت عند أستدعاء المكون أن تتلقى عنصر واحد من المكون ولو كان بداخله عدد من العناصر لا تفرق

فإذا احتجنا إلى أن يخرج عنصر فارغ وبداخله عناصرنا نستخدم عنصر (Fragments)

مثال

```
return (
  <React.Fragment>
    <ChildA />
    <ChildB />
    <ChildC />
  </React.Fragment>
 );
```

صياغة مختصرة :

```
return (
    <>
      <div>Hello</div>
      <div>World</div>
    </>
  );
```

بإمكانك استخدام <></> بنفس الطريقة التي تستخدم بها أي عنصر آخر عدا أنها لا تدعم المفاتيح أو الخاصيّات.

Keyed Fragments :

يُمكِن للأجزاء المُصرَّح عنها عن طريق الصياغة <React.Fragment> أن تمتلك مفاتيح. إحدى حالات الاستخدام لها هي ربط مجموعة إلى مصفوفة من الأجزاء، على سبيل المثال لإنشاء قائمة للوصف:

```
function Glossary(props) {
 return (
  <dl>
   {props.items.map(item => (
    // Without the `key`, React will fire a key warning
    <React.Fragment key={item.id}>
      <dt>{item.term}</dt>
      <dd>{item.description}</dd>
    </React.Fragment>
   ))}
  </dl>
 );
}
```

الخاصية الوحيدة التي يمكن تمريرها للأجزاء Fragment هي المفتاح key. قد نُضيف مستقبلًا دعم لخاصيّات إضافية مثل مُعالجات الأحداث

24

axios

وهي مكتبة خاصة لتعامل مع Api وأرسال request و استقبال response طلبات http requests فيها من نوع Promise
وبتالي تقرأ الكود وترسل الطلب ولا تنتظر الاستجابة بل تكمل قراءة الأكواد الأخرى حتى يتم استقبال الاستجابه

ميزاتها :

١. تجعل الطلبات XMLHttpRequests من المتصفح
٢. تجعل طلبات http requests من Node.js نفسها
٣. تدعم طلبات the Promise API
٤. تقاطع ارسال و استقبال الطلبات
٥. تحول الطلبات المرسلة و المستقبلة إلى بيانات
٦. إلغاء الطلبات
٧. تحويل تلقائي لبيانات Json
٨. دعم جانب العميل من خلال الحماية XSRF

التثبيت :

```
Using npm :
npm install axios

Using yarn :
yarn add axios
```

ثم نستدعي axios

```
import axios from "axios";
```

أنواع method في axios :

١. axios.request(config)
٢. axios.get(url[, config])
٣. axios.delete(url[, config])
٤. axios.head(url[, config])
٥. axios.options(url[, config])
٦. axios.post(url[, data[, config]])
٧. axios.put(url[, data[, config]])
٨. axios.patch(url[, data[, config]])

get(req) :

وهو طلب خاص فقط لقراءة البيانات بدون التعديل عليها ويحتوي على دالتين

then(res => return res). الذي يحصل على استجابة الطلب ويقوم بالتعامل معها

.catch(error => console.log(error)) تفعل عند عدم الحصول على then أي عدم الحصول على استجابة

```
import axios from "axios";

const res = axios.get(“http://”)
                .then(res => {return res} )
                .catch(error => {console.log(error)} )
                                                                    ويتم الاستدعاء من خلال
res.then(res => res);
```

مثال على get() :

```
import React from "react";
import axios from "axios";

let ammar = axios.get('https://jsonplaceholder.typicode.com/users')
  .then(responsee => {return responsee} )
  .catch(error => {console.log(error)} );

export default class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {users:[]};
  }

  componentDidMount ()  {

  ammar.then(response =>
    this.setState({
      users: response.data
    }));

  }

  render() {
    return (
      <div>
        <ul>
          {this.state.users.map(user =>
          <li key={user.id}>{user.name}</li>
          )}
        </ul>
        ammar
      </div>
    );
  }
}
```

ملاحظة : ال axios يتضمن داخله async / await بشكل أفتراضي أي يقدم الطلب ولا ينتظر الاستجابه بل يكمل قراءة الكود وعند الحصول على استجابه تفعل then() وعند عدم الحصول على استجابه تفعل catch()

لذلك في المثال السابق إذا تم حذف جزء من اللينك ستفعل catch() ولكن ستنتظر شي خمس ثوان لتفعل ولكن سيتم قراءة الكود كامل خلال هذه المدى أي ستظهر كلمة ammar

يمكنك كتابة async / await بشكل يدوي :

مثال

```
import React from "react";
import axios from "axios";

async function getUsers() {
 try {
  const response = await axios.get('https://jsonplaceholder.typicode.com/users');
  console.log(response.data);
  return response;
 } catch (error) {
  console.error(error);
 }
}

export default class App extends React.Component {
 constructor(props) {
  super(props);
  this.state = {users:[]};
 }

 componentDidMount ()  {
  getUsers().then(response =>
   this.setState({
    users: response.data
   }));

 }

 render() {
  return (
   <div>
    <ul>
     {this.state.users.map(user =>
     <li key={user.id}>{user.name}</li>
     )}
    </ul>
    ammar
   </div>
  );
 }
}
```

ملاحظة : لا تستخدم async / await بشكل يدوي لان المتصفحات قبل سنة 2017 لم تكن تدعمها لان هي جزء من ECMAScript 2017

كما يمكن استخدام Api جاهز من موقع json placeholder و ذلك من أجل تجريب الموقع

الرابط الديناميكي :

يمكننا جعل الرابط الطلب ديناميكي من خلال وضع متغييرات له مثل

```
axios.get('/user', {
   params: {
     ID: 12345
   }
  })
  .then(function (response) {
   console.log(response);
  })
  .catch(function (error) {
   console.log(error);
  })
```

أو طلب الحصول على صورة

```
axios({
  method: 'get',
  url: 'http://bit.ly/2mTM3nY',
  responseType: 'stream'
})
  .then(function (response) {
   response.data.pipe(fs.createWriteStream('ada_lovelace.jpg'))
  });
```

يمكننا إضافة then تفعل دوما

```
axios.get('/user')
  .then(function (response) {
   console.log(response);
  })
  .catch(function (error) {
   console.log(error);
  })
  .then(function (response) {
   console.log(response);
  });// تفعل دوما
```

default method :

default method في axios هي من نوع get()

مثال :

```
// Send a GET request (default method)
axios('/user/12345');
```

: delete()

وهي خاصة بحذف شيء من قاعدة البيانات من خلال تمرير id أو token المستخدم

```
export const deleteuser = (id) => {
   const responsee = axios.delete('https://jsonplaceholder.typicode.com/users/'+id);
   return responsee;
};
```

مثال :

يتم الحذف من خلال خطوتين

الأولى نرسل رابط الحذف

ثم الثانية نحدث الحالة state وأذا لم يتم أرسال رابط الحذف بسبب مشكلة ما تفعل catch ولا يتم تحديث الحالة

Userapi.js

```
import axios from "axios";

export const getuser = () => {
   const responsee = axios.get('https://jsonplaceholder.typicode.com/users');
   return responsee;
};

export const deleteuser = (id) => {
   const responsee = axios.delete('https://jsonplaceholder.typicode.com/users/'+id);
   return responsee;
};
```

وهو ملف خالص بأرسال روابط

App.js

```
import React from "react";
import {getuser,deleteuser} from "./UserApi";

export default class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      users:[],
      user:{}
    };
  }

  componentDidMount ()  {
  getuser()
  .then(response =>{
    this.setState({
```

```
    users: response.data
   });
   console.log(response.data);})
 .catch( () => alert("حدث خطأ في التحميل"));

 }

 setview = (user) => {
  this.setState({
    user: user
  })
 }

 deleteuser = (user) => {
  deleteuser(user.id)
  .then( () => {
    let users = this.state.users;
    const index = users.indexOf(user);
    users.splice(index, 1);
    this.setState({users});
    console.log(users);
  })
  .catch( () => alert("حدث خطأ في الحذف"))
 }

 render() {
  return (
    <div>
      <ul>
       {this.state.users.map(user =>
       <li key={user.id}>
        {user.name} {" "}
       <button onClick={() => this.setview(user)}>view</button> {" "}
       <button onClick={() => this.deleteuser(user)}>delete user</button>
       </li>
       )}
      </ul>

      <h3>user view</h3>
      {
       this.state.user.id > 0 ? (
         <>
          <div>Name: {this.state.user.name}</div>
          <div>Email: {this.state.user.email}</div>
         </>
       ) : (<div>please select a view</div>)
      }
    </div>
   );
 }
}
```

حيث توضع دوال تحديث الحالة state داخل then حيث تفعل إذا تم أرسال الرابط وقام بارجاع استجابة

ملاحظة : لا نتعامل مع then و catch في ملف أرسال الروابط بل في المكون نفسه

: put()

وهي خاصة بتعديل شيء من قاعدة البيانات من خلال تمرير id أو token المستخدم والقيم الجديدة

```
export const updateuser = (id, values) => {
   const responsee = axios.put('https:// '+id, values);
   return responsee;
};
```

مثال السابق معدل :

ينقوم بأرسال التعديل ثم يتم استقبال استجابة ككائن معدل هذا الكائن نستخدمه لتعديل المصفوفة ثم نعدل الحالة

APP.js

```
import React from "react";
import Editview from "./Editview";
import {getuser,deleteuser,updateuser} from "./UserApi";
import Userview from "./Userview";

export default class App extends React.Component {
 constructor(props) {
   super(props);
   this.state = {
     users:[],
     user:{}
   };
 }
 //get user
 componentDidMount ()  {
 getuser()
 .then(response =>{
   this.setState({
     users: response.data
   });
   console.log(response.data);})
 .catch( () => alert("حدث خطأ في التحميل"));

 }
 // set user
 setview = (user) => {
   this.setState({
     user: user
   })
 }
 // delete user
```

```
 deleteuser = (user) => {
  deleteuser(user.id)
  .then( () => {
    let users = this.state.users;
    const index = users.indexOf(user);
    users.splice(index, 1);
    this.setState({users});
    console.log(users);
  })
  .catch( () => alert("حدث خطأ في الحذف"))
 }
 //Change state user
 handleChange = (event) => {
  if(event.target.type === 'email') {
    this.setState({user:{
      ...this.state.user,
      email: event.target.value
    }
    });
  }
  if(event.target.type === 'text') {
    this.setState({user:{
      ...this.state.user,
      name: event.target.value
    }
    });
  }

 }
 //edit user
 edituserSubmit = (event) => {
  const id = this.state.user.id;
  const values = this.state.user;
  let users = this.state.users;

  updateuser(id, values)
  .then( (responsee) => {
    users.splice(id - 1, 1, responsee.data);
    console.log(responsee.data);
    this.setState({
      users:users
    });
  })
  .catch( () => alert("حدث خطأ في التحديث"));

  event.preventDefault();
 }

 render() {
  return (
    <div>
```

```
      <ul>
        {this.state.users.map(user =>
        <li key={user.id}>
          {user.name} {" "}
        <button onClick={() => this.setview(user)}>view</button> {" "}
        <button onClick={() => this.deleteuser(user)}>delete user</button>
        </li>
        )}
      </ul>

      <Userview value={this.state.user}/>

      <Editview value={this.state.user} handleChange={this.handleChange}
edituserSubmit={this.edituserSubmit} />

    </div>
  );
 }
}
```

UserApi.js

```
import axios from "axios";

export const getuser = () => {
  const responsee = axios.get('https://jsonplaceholder.typicode.com/users');
  return responsee;
};

export const deleteuser = (id) => {
  const responsee = axios.delete('https://jsonplaceholder.typicode.com/users/'+id);
  return responsee;
};

export const updateuser = (id, values) => {
  const responsee = axios.put('https://jsonplaceholder.typicode.com/users/'+id, values);
  return responsee;
};
```

Editview.js

```
import React from "react";

export default function Editview(props) {
  const value = props.value;
  return (
    <div>
      <h3>edit user</h3>
```

```
        <form onSubmit={props.edituserSubmit}>
          <label>
            Name :
          </label>
          <br/>
          <input type="text" value={value.name} onChange={props.handleChange} />
          <br/>
          <label>
            Email :
          </label>
          <br/>
          <input type="email" value={value.email} onChange={props.handleChange} />
          <br/>
          <input type="submit" value="edit" />
        </form>
      </div>
    ) ;
}
```

Userview.js

```
import React from "react";

export default function Userview(props) {
   const value = props.value;
   return (
      <div>
         <h3>user view</h3>
         {
            value.id > 0 ? (
            <div>
               <div>Name: {value.name}</div>
               <div>Email: {value.email}</div>
            </div>
         ) : (<div>please select a view</div>)
         }
      </div>
   ) ;
}
```

: post()

نقوم من خلالها بعمل register أو أضافة مستخدم جديد لا جيب أرسال token

```
axois.post("https://",object)
```

بشكل أخر

```
axios.post('/user', {
   firstName: 'Fred',
```

```
  lastName: 'Flintstone'
 })
 .then(function (response) {
  console.log(response);
 })
 .catch(function (error) {
  console.log(error);
 });
```

شكل أخر

```
axios({
 method: 'post',
 url: '/user/12345',
 data: {
  firstName: 'Fred',
  lastName: 'Flintstone'
 }
});
```

شكل أخر

```
axios.post('/user/12345', {
 name: 'new name'
}, {
 cancelToken: source.token
})

// cancel the request (the message parameter is optional)
source.cancel('Operation canceled by the user.');
```

مثال وهو تعديل عن المثال السابق :

APP.js

```
import React from "react";
import Adduser from "./Adduser";
import Editview from "./Editview";
import {getuser, deleteuser, updateuser, adduser} from "./UserApi";
import Userview from "./Userview";

export default class App extends React.Component {
 constructor(props) {
  super(props);
  this.state = {
   users:[],
   user:{},
   adduser:{
    id:null,name:null,username:null,email:null,phone:null,website:null,
    address:{city:null,street:null,suite:null,zipcode:null,geo:{lat:null,lng:null}},
    company:{name:null,catchPhrase:null,bs:null}
   }
  };
 }
```

```
//get user
componentDidMount ()  {
getuser()
.then(response =>{
  this.setState({
    users: response.data
  });
  console.log(response.data);})
.catch( () => alert("حدث خطأ في التحميل"));

}
// set user
setview = (user) => {
  this.setState({
    user: user
  })
}
// delete user
deleteuser = (user) => {
  deleteuser(user.id)
  .then( () => {
    let users = this.state.users;
    const index = users.indexOf(user);
    users.splice(index, 1);
    this.setState({users});
    console.log(users);
  })
  .catch( () => alert("حدث خطأ في الحذف"))
}
//Change state user
handleChange = (event) => {
  if(event.target.type === 'email') {
    this.setState({user:{
      ...this.state.user,
      email: event.target.value
    }
    });
  }
  if(event.target.type === 'text') {
    this.setState({user:{
      ...this.state.user,
      name: event.target.value
    }
    });
  }

}
//edit user
edituserSubmit = (event) => {
  const id = this.state.user.id;
  const values = this.state.user;
```

```
  let users = this.state.users;

  updateuser(id, values)
  .then( (responsee) => {
    users.splice(id - 1, 1, responsee.data);
    console.log(responsee.data);
    this.setState({
      users:users
    });
  })
  .catch( () => alert("حدث خطأ في التحديث"));

  event.preventDefault();
}

//Add user
adduserSubmit = (event) => {

  let id = this.state.users.length;
    this.setState({adduser:{
      ...this.state.adduser,
      id:id
    }
    });

  adduser(this.state.adduser)
  .then( (responsee) => {
    let newusers = this.state.users;
    newusers.push(responsee.data);
    console.log(this.state.users);
    this.setState({
      users:newusers
    });
  })
  .catch( () => alert("حدث خطأ في الأضافة"));

  event.preventDefault();
}
//Change state adduser
handleaddChange = (event) => {
  if(event.target.type === 'email') {
    this.setState({adduser:{
      ...this.state.adduser,
      email: event.target.value
    }
    });
  }
  if(event.target.type === 'text') {
    this.setState({adduser:{
      ...this.state.adduser,
      name: event.target.value
```

```
    }
    });
  }

 }

 render() {
  return (
    <div>
      <ul>
       {this.state.users.map(user =>
       <li key={user.id}>
        {user.name} {" "}
       <button onClick={() => this.setview(user)}>view</button> {" "}
       <button onClick={() => this.deleteuser(user)}>delete user</button>
       </li>
       )}
      </ul>

      <Userview value={this.state.user}/>

      <Editview valueuser={this.state.user} handleChange={this.handleChange}
edituserSubmit={this.edituserSubmit} />

      <Adduser valueuser={this.state.adduser} handleaddChange={this.handleaddChange}
adduserSubmit={this.adduserSubmit} />

    </div>
  );
 }
}
```

UserApi.js

```
import axios from "axios";

export const getuser = () => {
  const responsee = axios.get('https://jsonplaceholder.typicode.com/users');
  return responsee;
};

export const deleteuser = (id) => {
  const responsee = axios.delete('https://jsonplaceholder.typicode.com/users/'+id);
  return responsee;
};

export const updateuser = (id, values) => {
  const responsee = axios.put('https://jsonplaceholder.typicode.com/users/'+id, values);
  return responsee;
```

```
};

export const adduser = (values) => {
   const responsee = axios.post('https://jsonplaceholder.typicode.com/users/',values);
   return responsee;
};
```

Adduser.js

```
import React from "react";

export default function Adduser(props) {
   const valueuser = props.valueuser;
   return (
      <div>
         <h3>Add user</h3>
         <form onSubmit={props.adduserSubmit}>
            <label>
               Name :
            </label>
            <br/>
            <input type="text" value={valueuser.name === null ? "":valueuser.name}
onChange={props.handleaddChange} />
            <br/>
            <label>
               Email :
            </label>
            <br/>
            <input type="email" value={valueuser.email === null ? "":valueuser.email}
onChange={props.handleaddChange} />
            <br/>
            <input type="submit" value="Add" />
         </form>
      </div>
   ) ;
}
```

Editview.js

```
import React from "react";

export default function Editview(props) {
   const valueuser = props.valueuser;
   return (
      <div>
         <h3>Edit user</h3>
         <form onSubmit={props.edituserSubmit}>
            <label>
               Name :
```

```
            </label>
            <br/>
            <input type="text" value={valueuser.name === null ? "":valueuser.name}
onChange={props.handleChange} />
            <br/>
            <label>
               Email :
            </label>
            <br/>
            <input type="email" value={valueuser.email === null ? "":valueuser.email}
onChange={props.handleChange} />
            <br/>
            <input type="submit" value="Edit" />
         </form>
      </div>
   ) ;
}
```

Userview.js

```
import React from "react";

export default function Userview(props) {
   const value = props.value;
   return (
      <div>
         <h3>user view</h3>
         {
            value.id > 0 ? (
            <div>
               <div>Name: {value.name}</div>
               <div>Email: {value.email}</div>
            </div>
         ) : (<div>please select a view</div>)
         }
      </div>
   ) ;
}
```

: all()

نستخدم لأرسال أكثر من طلب بوقت واحد أي تنفيذ عدة طلبات متزامنة

```
function getUserAccount() {
  return axios.get('/user/12345');
}

function getUserPermissions() {
```

```
  return axios.get('/user/12345/permissions');
}

Promise.all([getUserAccount(), getUserPermissions()])
  .then(function (results) {
    const acct = results[0];
    const perm = results[1];
  });
```

: axios.create()

لأنشاء axios بقيم معينة ومن يتم استدعائه من قبل ... get / post / put / delete

```
const instance = axios.create({
  baseURL: 'https://some-domain.com/api/',
  timeout: 1000,// مهله الاستجابه قبل عرض الخطأ
  headers: {'X-Custom-Header': 'foobar'}
});
```

: Request Config

يحتوي الطلب على عدت معلومات وتكون بشكل إفتراضي

```
{
  // `url` is the server URL that will be used for the request
  url: '/user',

  // `method` is the request method to be used when making the request
  method: 'get', // default

  // `baseURL` will be prepended to `url` unless `url` is absolute.
  // It can be convenient to set `baseURL` for an instance of axios to pass relative URLs
  // to methods of that instance.
  baseURL: 'https://some-domain.com/api',

  // `transformRequest` allows changes to the request data before it is sent to the server
  // This is only applicable for request methods 'PUT', 'POST', 'PATCH' and 'DELETE'
  // The last function in the array must return a string or an instance of Buffer, ArrayBuffer,
  // FormData or Stream
  // You may modify the headers object.
  transformRequest: [function (data, headers) {
    // Do whatever you want to transform the data

    return data;
  }],

  // `transformResponse` allows changes to the response data to be made before
  // it is passed to then/catch
  transformResponse: [function (data) {
    // Do whatever you want to transform the data
```

```
  return data;
}],

// `headers` are custom headers to be sent
headers: {'X-Requested-With': 'XMLHttpRequest'},

// `params` are the URL parameters to be sent with the request
// Must be a plain object or a URLSearchParams object
// NOTE: params that are null or undefined are not rendered in the URL.
params: {
  ID: 12345
},

// `paramsSerializer` is an optional function in charge of serializing `params`
// (e.g. https://www.npmjs.com/package/qs, http://api.jquery.com/jquery.param/)
paramsSerializer: function (params) {
  return Qs.stringify(params, {arrayFormat: 'brackets'})
},

// `data` is the data to be sent as the request body
// Only applicable for request methods 'PUT', 'POST', 'DELETE', and 'PATCH'
// When no `transformRequest` is set, must be of one of the following types:
// - string, plain object, ArrayBuffer, ArrayBufferView, URLSearchParams
// - Browser only: FormData, File, Blob
// - Node only: Stream, Buffer
data: {
  firstName: 'Fred'
},

// syntax alternative to send data into the body
// method post
// only the value is sent, not the key
data: 'Country=Brasil&City=Belo Horizonte',

// `timeout` specifies the number of milliseconds before the request times out.
// If the request takes longer than `timeout`, the request will be aborted.
timeout: 1000, // default is `0` (no timeout)

// `withCredentials` indicates whether or not cross-site Access-Control requests
// should be made using credentials
withCredentials: false, // default

// `adapter` allows custom handling of requests which makes testing easier.
// Return a promise and supply a valid response (see lib/adapters/README.md).
adapter: function (config) {
  /* ... */
},

// `auth` indicates that HTTP Basic auth should be used, and supplies credentials.
// This will set an `Authorization` header, overwriting any existing
```

```
  // `Authorization` custom headers you have set using `headers`.
  // Please note that only HTTP Basic auth is configurable through this parameter.
  // For Bearer tokens and such, use `Authorization` custom headers instead.
  auth: {
    username: 'janedoe',
    password: 's00pers3cret'
  },

  // `responseType` indicates the type of data that the server will respond with
  // options are: 'arraybuffer', 'document', 'json', 'text', 'stream'
  //   browser only: 'blob'
  responseType: 'json', // default

  // `responseEncoding` indicates encoding to use for decoding responses (Node.js only)
  // Note: Ignored for `responseType` of 'stream' or client-side requests
  responseEncoding: 'utf8', // default

  // `xsrfCookieName` is the name of the cookie to use as a value for xsrf token
  xsrfCookieName: 'XSRF-TOKEN', // default

  // `xsrfHeaderName` is the name of the http header that carries the xsrf token value
  xsrfHeaderName: 'X-XSRF-TOKEN', // default

  // `onUploadProgress` allows handling of progress events for uploads
  // browser only
  onUploadProgress: function (progressEvent) {
    // Do whatever you want with the native progress event
  },

  // `onDownloadProgress` allows handling of progress events for downloads
  // browser only
  onDownloadProgress: function (progressEvent) {
    // Do whatever you want with the native progress event
  },

  // `maxContentLength` defines the max size of the http response content in bytes allowed in node.js
  maxContentLength: 2000,

  // `maxBodyLength` (Node only option) defines the max size of the http request content in bytes
allowed
  maxBodyLength: 2000,

  // `validateStatus` defines whether to resolve or reject the promise for a given
  // HTTP response status code. If `validateStatus` returns `true` (or is set to `null`
  // or `undefined`), the promise will be resolved; otherwise, the promise will be
  // rejected.
  validateStatus: function (status) {
    return status >= 200 && status < 300; // default
  },

  // `maxRedirects` defines the maximum number of redirects to follow in node.js.
```

```
  // If set to 0, no redirects will be followed.
  maxRedirects: 5, // default

  // `socketPath` defines a UNIX Socket to be used in node.js.
  // e.g. '/var/run/docker.sock' to send requests to the docker daemon.
  // Only either `socketPath` or `proxy` can be specified.
  // If both are specified, `socketPath` is used.
  socketPath: null, // default

  // `httpAgent` and `httpsAgent` define a custom agent to be used when performing http
  // and https requests, respectively, in node.js. This allows options to be added like
  // `keepAlive` that are not enabled by default.
  httpAgent: new http.Agent({ keepAlive: true }),
  httpsAgent: new https.Agent({ keepAlive: true }),

  // `proxy` defines the hostname, port, and protocol of the proxy server.
  // You can also define your proxy using the conventional `http_proxy` and
  // `https_proxy` environment variables. If you are using environment variables
  // for your proxy configuration, you can also define a `no_proxy` environment
  // variable as a comma-separated list of domains that should not be proxied.
  // Use `false` to disable proxies, ignoring environment variables.
  // `auth` indicates that HTTP Basic auth should be used to connect to the proxy, and
  // supplies credentials.
  // This will set an `Proxy-Authorization` header, overwriting any existing
  // `Proxy-Authorization` custom headers you have set using `headers`.
  // If the proxy server uses HTTPS, then you must set the protocol to `https`.
  proxy: {
    protocol: 'https',
    host: '127.0.0.1',
    port: 9000,
    auth: {
      username: 'mikeymike',
      password: 'rapunz3l'
    }
  },

  // `cancelToken` specifies a cancel token that can be used to cancel the request
  // (see Cancellation section below for details)
  cancelToken: new CancelToken(function (cancel) {
  }),

  // `decompress` indicates whether or not the response body should be decompressed
  // automatically. If set to `true` will also remove the 'content-encoding' header
  // from the responses objects of all decompressed responses
  // - Node only (XHR cannot turn off decompression)
  decompress: true // default

}
```

: Response Schema

تحتوي الاستجابه على عدة معلومات وتكون بشكل إفتراضي

```
{
  // `data` is the response that was provided by the server
  data: {},

  // `status` is the HTTP status code from the server response
  status: 200,

  // `statusText` is the HTTP status message from the server response
  // As of HTTP/2 status text is blank or unsupported.
  // (HTTP/2 RFC: https://www.rfc-editor.org/rfc/rfc7540#section-8.1.2.4)
  statusText: 'OK',

  // `headers` the HTTP headers that the server responded with
  // All header names are lower cased and can be accessed using the bracket notation.
  // Example: `response.headers['content-type']`
  headers: {},

  // `config` is the config that was provided to `axios` for the request
  config: {},

  // `request` is the request that generated this response
  // It is the last ClientRequest instance in node.js (in redirects)
  // and an XMLHttpRequest instance in the browser
  request: {}
}
```

للوصول إليهم من then

```
axios.get('/user/12345')
  .then(function (response) {
    console.log(response.data);
    console.log(response.status);
    console.log(response.statusText);
    console.log(response.headers);
    console.log(response.config);
  });
```

: Handling Errors

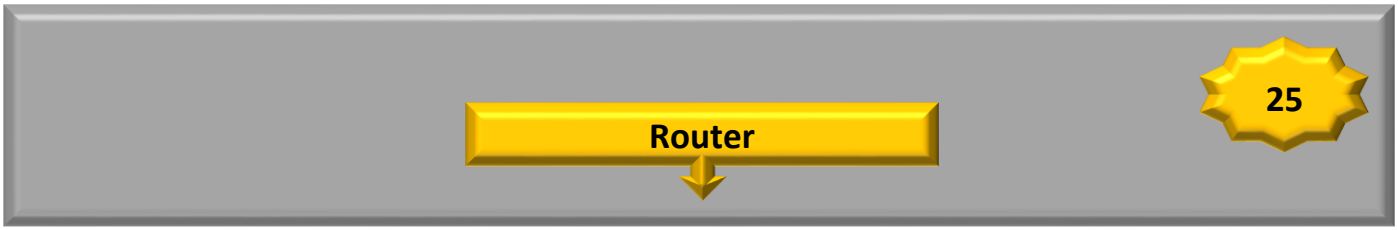
يتم معالجة الأخطاء الحالصة عن الطلب و الأستجابه

```
axios.get('/user/12345')
  .catch(function (error) {
    if (error.response) {
      // The request was made and the server responded with a status code
      // that falls out of the range of 2xx
```

```
    console.log(error.response.data);
    console.log(error.response.status);
    console.log(error.response.headers);
  } else if (error.request) {
    // The request was made but no response was received
    // `error.request` is an instance of XMLHttpRequest in the browser and an instance of
    // http.ClientRequest in node.js
    console.log(error.request);
  } else {
    // Something happened in setting up the request that triggered an Error
    console.log('Error', error.message);
  }
  console.log(error.config);
});
```

معالجة أخطاء البيانات التي لا تكون على شكل json()

```
axios.get('/user/12345')
 .catch(function (error) {
  console.log(error.toJSON());
 });
```



وهي أحلى مكتبه بالريأكت حيث تمكننا من الأنتقال بين الصفحات من خلال تغيير لينك الصفحة بدون عمل render و الأتصال بالسيرفير وبتالي يكون الموقع فائق السرعة

للتثبيت :

```
npm
npm install react-router-dom@6
yarn
yarn add react-router-dom@6
```

البدء بالعمل :

BrowserRouter :

وهو الوسم الرئيسي حيث يغلف الصفحات التي تتغيير حسب اللينك حيث يوضع داخله Routes وبداخل Routes يوجد الصفحات Route التي تستدعي الصفحات

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

```
export default function App () {
  return (
    <BrowserRouter>

    </BrowserRouter>
  );
}
```

: Routes

توضع داخل التغليف BrowserRouter ومهمتها هي أحتواء Route التي تستدعي الصفحات حسب اللينك

```
import React from "react";
import {BrowserRouter, Routes, Route} from "react-router-dom";
import Home from "./Home";
import About from "./About";

export default function App () {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </BrowserRouter>
  );
}
```

: Route

توضع داخل Routes ومهمتها هي استدعاء المكونات حسب اللينك وتأخذ عدة بارامترات أمها

path=”/” التي تقارن اللينك الرئيسي في حال التطابق يعمل ال Route ويظهر المكون

element={<component />} وظيفتها استدعاء المكون ويمكن الكتابة داخلها jsx مثل element={<dv> }

: Link

و هو عبارة عن زر وظيفته تغيير اللينك الرئيسي وذلك عند الضغط عليه تفعل ال to وتعطي اللينك وعندما يتغيير اللينك الرئيسي تستشعر ال path التي في Route التغيرات وتستدعي المكون الموافق

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

مثال عن الأفكار :

App.js

```
import React from "react";
import {BrowserRouter, Routes, Route} from "react-router-dom";
import Home from "./Home";
import About from "./About";

export default function App () {
   return (
     <BrowserRouter>
       <Routes>
         <Route path="/" element={<Home />} />
         <Route path="/about" element={<About />} />
       </Routes>
     </BrowserRouter>
   );
}
```

Home.js

```
import React from "react";
import { Link } from "react-router-dom";

export default function Home() {
   return (
      <>
         <main>
         <h2>Welcome to the homepage!</h2>
         <p>You can do this, I believe in you.</p>
         </main>
         <nav>
         <Link to="/about">About</Link>
         </nav>
      </>
   );
}
```

يتم استدعاء هذا المكون عندما يكون الرابط الرئيسي "path="/" وعند الضغط على Link يتغيير الرابط إلى "path="/about

About.js

```
import React from "react";
import { Link } from "react-router-dom";

export default function About() {
   return (
      <>
         <main>
         <h2>Who are we?</h2>
         <p>
            That feels like an existential question, don't you
            think?
```

```
        </p>
        </main>
        <nav>
        <Link to="/">Home</Link>
        </nav>
      </>
   );
}
```

يتم استدعاء هذا المكون عندما يكون الرابط الرئيسي "path="/about وعند الضغط على Link يتغيير الرابط إلى "path="/

Nested Routes :

وهي مجموعة صفحات أو روابط تشترك بالجزء الأول من الرابط

```
<Routes>
      <Route path="/" element={<Home />} />
      <Route path="/home/" >
       <Route path="1" element={<><h1>ammar1</h1><Link to="/">return to Page
Home</Link></>}/> // "/home/1"
       <Route path="2" element={<><h1>ammar2</h1><Link to="/">return to Page
Home</Link></>}/>// "/home/2"
      </Route>
</Routes>
```

الراوبط الغير موجودة :

يمكن للمستخدم أدخال روابط غير موجودة وبتالي يحصل error لذلك نضيف route التالي يوضع أخر شئ

```
<Route path="*" element={<><h1>404</h1><Link to="/">return to Page Home</Link></>} />
```

وبتالي أي رابط غير موجود سوف يفعل هذا ال route وستخدم لصفحات 404 والتي تحول إلى الصفحة الرئيسية

Route الديناميكي :

حيث يكون عدد لا نهائي من الوابط بنفس الصيغة

لنفرض لدينا الروابط التالية

```
http://localhost:3000/Userview/1
http://localhost:3000/Userview/2
http://localhost:3000/Userview/3
```

وهكذا لدينا عدد لا نهائي من المستخدمين أفضل طريقة لتعريفهم داخل Link هي

```
{this.state.users.map(user =>
      <li key={user.id}>
      <Link to={`Userview/${user.id}`} >view</Link>
      </li>
      )}
```

طريقة الاستدعاء تكون

```
<Route path="/Userview" element={<Userview value={this.state.user}/>}/>
```

```
<Route path="/Userview/:id" element={<Userview value={this.state.user}/>}/>
```

هكذا يكون الرابط ديناميكي بعدد لا نهائي من id للوصول لل id الذي فيي الرابط نستخدم useParams

Active Links :

أسمها NavLink وهي تشبه Link ولكن فيها ميزة الضغط أي يفعل الستايل مثل للون الخلفية و ذلك عندما يفعل اللينك المطلوب يفعل الستايل الذي يكتب كدالة أروفانكشن

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

<NavLink to={"/"} style={({ isActive }) => {return {backgroundColor: isActive ? "red" : ""};}}>
  Home
</NavLink>
```

أو يمكن أضافة كلاس عند التفعيل الزر

```
<NavLink className={({ isActive }) => isActive ? "colorred" : "colorblue"} />
```

حيث isActive تكون true عندما يكون اللينك مطابق وتكون false عندما يكون اللينك غير مطابق

ملاحظة خطيرة عن Link و NavLink :

لاحظنا أنه وقت عم نكتب Link أو NavLink عم نكرر حالنا بسبب تكرار الخاصيات يلي عم نمررها

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

<NavLink to={"/"} style={({ isActive }) => {return {backgroundColor: isActive ? "red" : ""};}}>
  Home
</NavLink>
<NavLink to={"/user"} style={({ isActive }) => {return {backgroundColor: isActive ? "red" : ""};}}>
  Home
</NavLink>
<NavLink to={"/login"} style={({ isActive }) => {return {backgroundColor: isActive ? "red" : ""};}}>
  Home
</NavLink>
```

فعليا عم يتغيير فقط to={""} لذلك لح نأنشئ كومبوننت ونصير نستدعيها بدل NavLink ويكون يحوي الخاصيات المكررة

```
export const MyNavLink = (props) =><NavLink className="navlink" style={({ isActive }) => {return
{backgroundColor: isActive ? "red" : ""};}} {...props}>{props.children}</NavLink>;
```

نلاحظ وضعنا {...props} لتستقبل ال to أو أي خاصيات جديدة و وضعنا {props.children} لتستقبل نص أسم المكون ويمكننا استدعاء هذا المكون لك المشروع

الاستدعاء

```
<MyNavLink to={"/"}> Home </MyNavLink>
<MyNavLink to={"/Userview"}>Userview</MyNavLink>
<MyNavLink to={"/Editview"}>Editview</MyNavLink>
<MyNavLink to={"/Adduser"}>Adduser</MyNavLink>
```

useParams() :

تستخدم للوصول إلى المتغيير في الرابط مثل id أو أي متغيير نسميه

```
http://localhost:3000/Userview/1121
```

للوصول إلى المتغييرحيث كان ال Route

```
<Route path="/Userview/:id" element={<Userview value={this.state.user}/>}/>
```

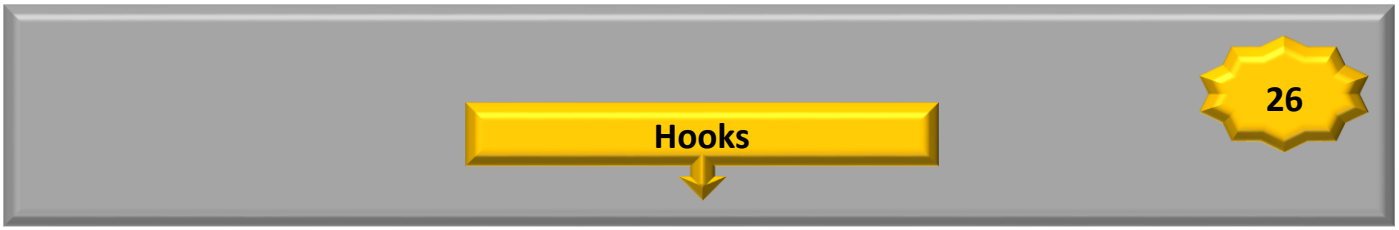
داخل المكون <Userview /> نستدعي useParams كتالي

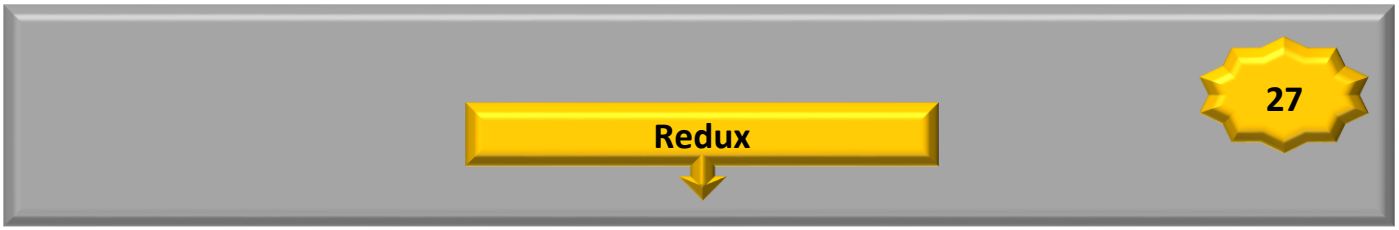
```
import { useParams } from "react-router-dom";

export default function Invoice() {
  let params = useParams();
  return <h2>idNumber : {params.id}</h2>;
}
```

: Index Routes

وهو Route الأفتراضي و الأساسي للموقع يفتح عند استدعاء domin ويوضع أول شيء حيث نبدل path ب index

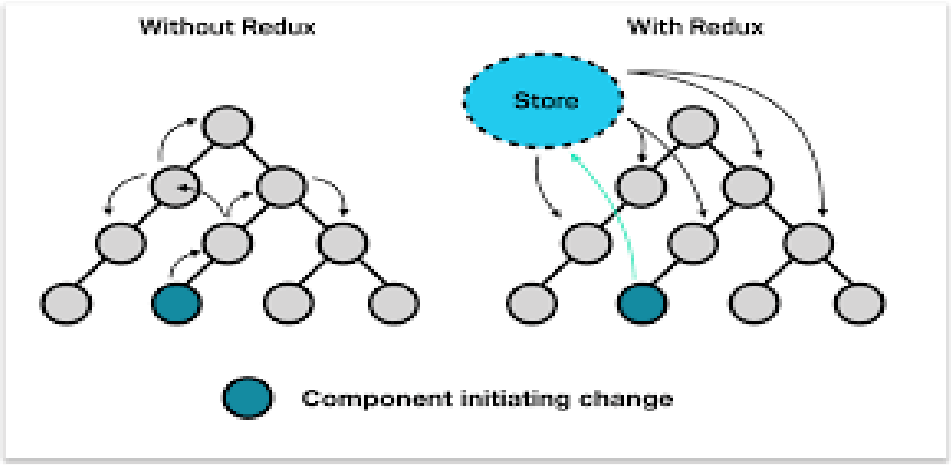
```
<Route index element={<Home />}/>
```

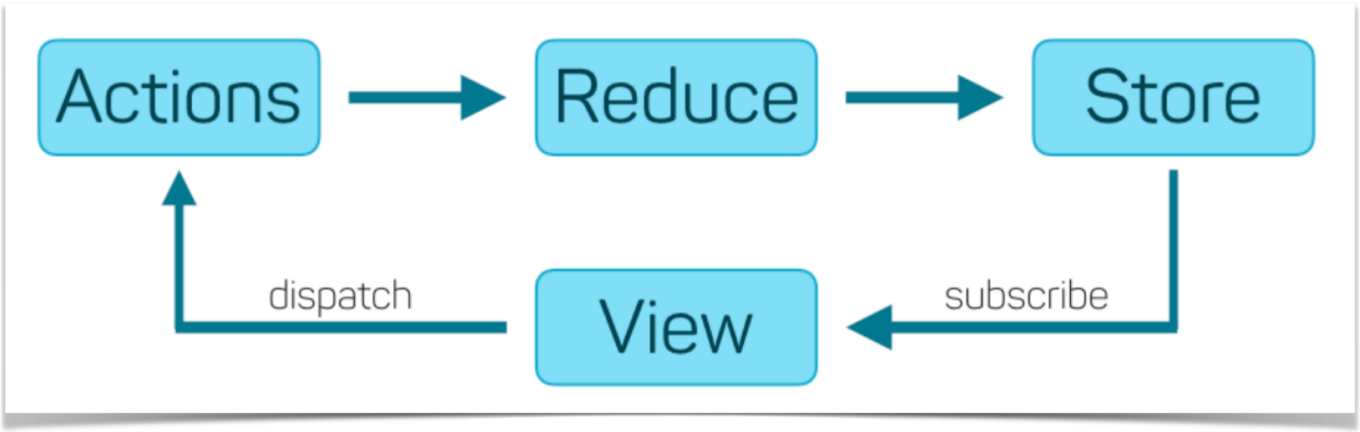


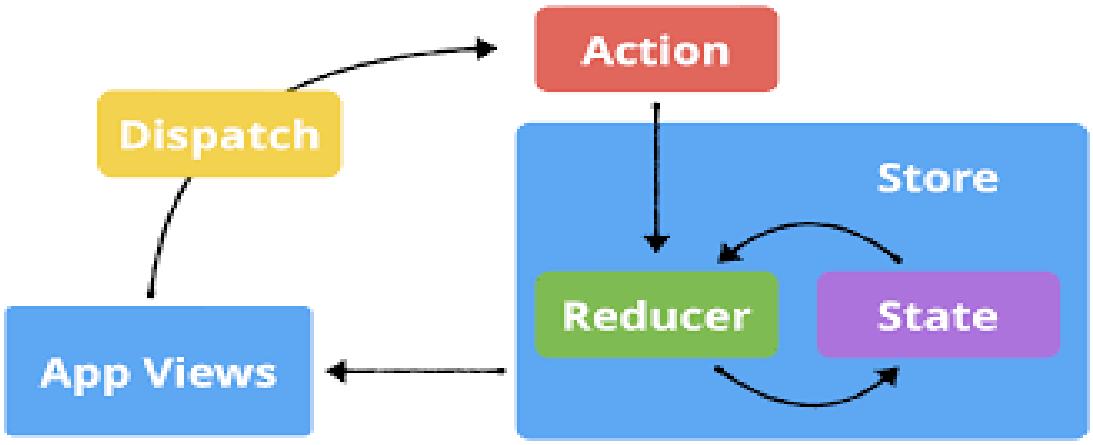


و هي حالة عامة Store (Global state) أي هي حالة يمكن لكل المكونات الوصول لها مباشرة ويمكن التعديل على الحالة أي هي خزان عام للبيانات لكل المكونات تشبه Context حيث لا يكون هناك حاجة لتمرير البيانات عبر المكونات للوصول للحالة الرئيسية بل يمكن الوصول لها مباشرة

وبتالي store هو خزان يحتوي على الحالة state التي نحتاج أن نشاركها مع كل المكونات مثل بيانات login







subscribe : وهي عملية ربط المكون مع **store** حيث يمكننا قراءة المعلومات الحالة **state** التي في **store** في بداية أقلاع التطبيق و أي تحديث في المعلومات الحالة **state** في **store** سوف نحصل عليها و يتحدث المكون ولا يمكن لل **subscribe** التعديل على بيانات الحالة ال **state** في **store** أبدا

state : وهي الحالة التي تكون عبارة عن مجموعة من البيانات في **store** ويتم أنشائها و التعديل عليها من قبل **Reducer**

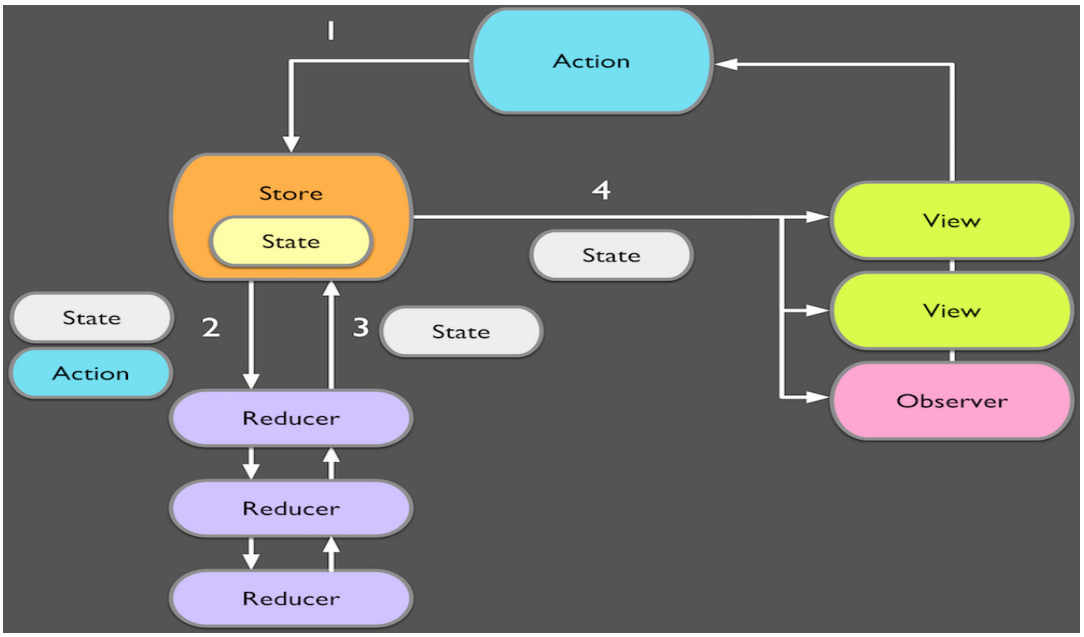
Reducer : يمكنه أنشاء الحالة state في store و يمكنه التعديل على بيانات الحالة state في store و أرجاع التعديل للبيانات الحالة state إلى store ويتلقى التعديل من المكون عن طريق Dispatch Action أي بمعنى لتعديل الحالة لا نضع أكواد التعديل داخل المكون بل نضعها داخل Reducer ونتلقى أمر التعديل عبر Dispatch Action من المكون وبتالي الذي يقوم بأنشاء و التعديل على بيانات state في store هو Reducer فقط ويحتوي هو فقط على دوال متنوعة حسب الغرض لتعديل الحالة state

Dispatch Action Function : وهو رد فعل على شكل دالة يستقبل الكائن Action الذي يتم إطلاقه من المكون عندما يقع حدث ما و يتضمن الكائن Action النوع type هو نوع أو الأسم الدالة التي سوف تفعل داخل Reducer لتقوم بتعديل الحالة state في store و كذلك يتضمن الكائن Action على البيانات الجديدة payload (الحمل)

دوما عند بناء تطبيق جديد نبدأ ببناء Reducer و Action للمكونات أولا

شرح الأجزاء بعمق :

Reducer : وهو عبارة عن دالة أو مكان لأنشاء والتعديل على الحالة state في store و بداخلها دوال تعدل على الحالة state و يمكن أن تحتوي على عدة Reducer و يتجمعوا كلهم في Reducer واحد



شكل ال Reducer :

```
const initstate = {username: “ammar”};

const user = (state = initstate, action) {
 if(action.type = “update”) {
  some logic to handle action
  return new state + action.payload
 }
  return State;
}
```

تأخذ بارامترين فقط action / state وأعطينا لل state قيمة أفتراضية

وبتالي هو يستقبل الكائن action الذي يحتوي على { {type:"update", payload:{name: "qassab"} } مع قيمة ال state القديمة ويعدل على الحالة ويرجعها ليتلقاها ال store وأذا لم تتحقق أي if يرجع الحالة القديمة

ملاحظات عن Reducer :

١. هي دالة
٢. pure : ليس من وظائفها الأتصال ب api أو تعدل ملف أو تقرأ من ملف
٣. وظيفتها تأخذ action / state وتعدل عليه ثم ترجع state فقط
٤. action / state تبقى تسميتها هكذا لا تغيير بالأحرف
٥. الحالة state يجب أن تكون عبارة عن كائن object
٦. نأخذ الحالة القيدمة ونعدل عليها ونرجع حالة جديدة
٧. معناها دوما يجب أن يكون هناك return للحالة state من أجل ان يأخذه ال store
٨. يمكن أن يكون لدينا عدة Reducer حيث لكل حالة state يجب أن يكون لها Reducer
٩. لدينا Reducer يجمع كل Reducer
١٠. كل Reducer يرجع حالة return state ليتلقاها ال store
١١. ال store يمتلك Reducer و الحالة state أي يكونوا داخل ال store
١٢. ال store هو الذي يستلم Dispatch Action Function ويرسلها لل Reducer المناسب عن طريق البحث داخل كل Reducer من خلال حلقة for
١٣. عندما يعمل store لأول مرة يعمل فور لوب لكل Reducer و لا يكون هناك اي Action وبتالي لن تعمل أي if وبتالي سيرجع الحالة الأفتراضية return state وهكذا يتم أنشاء الحالة state في store عن بداية أقلاع التطبيق لذلك يجب وضع قيمة أفتراضية للحالة حتى لو كانت كائن فارغ state = {}
١٤. state = current تأخذ أخر حالة من أجل التعديل عليها حيث store يلغي القيمة الأفتراضية ويضع أخر قيمة current لل state
١٥. دوما يدخل لل Reducer حالة قديمة وأكشن action / state من عدا في الأقلاع يدخل فقط state ألا أذا أردت Action يعمل في الأقلاع

تثبيت Redux :

لتثبيت Redux Toolkit وهي أحدث أصدار لل Redux وتشمل الأصدارة يلي قبل التحديث Redux Core

```
# NPM
npm install @reduxjs/toolkit

# Yarn
yarn add @reduxjs/toolkit
```

لتثبيت Redux Core وهي الأصدار الأقدم من Redux

```
# NPM
npm install redux

# Yarn
yarn add redux
```

مثال :

مراحل عمل redux في جافاسكربت عادية :

١. ننشأ حالة state أفتراضية
٢. ننشأ Reducer وندخل داخله الحالة state الأفتراضية
٣. ننشأ store وندخل داخله Reducer
٤. ننشأ المكون الذي سوف نربطه مع store ونعطيه القيمة الأفتراضية لل state من خلال store.getState().value
٥. نربط المكون مع ال store من خلال subscribe()

```
<div><span id=”root”></span></div>
<button id=”increase”></button>
<button id=”decrease”></button>
```

```
const initstate = {value: 0};

const counterReducer = (state = initstate, action) => {
 // logic
 return state ;
};

let store = createStore(counterReducer);

const counterApp = () => {
document.getElementById(“root”).innerText = store.getState().value ;
};
counterApp();

store.subscribe(counterApp);
```

١. ننشأ action داخل الحدث
٢. نرسل ال action إلى store من خلال store.dispatch(action)
٣. عندما نرسل action يستقبلها Reducer ويقارن type مع logic عند التوافق ينفذ logic
٤. نضع state... من أجل وضع كل قيم الحالة state من أجل عدم حذف القيم التي لا تعدل

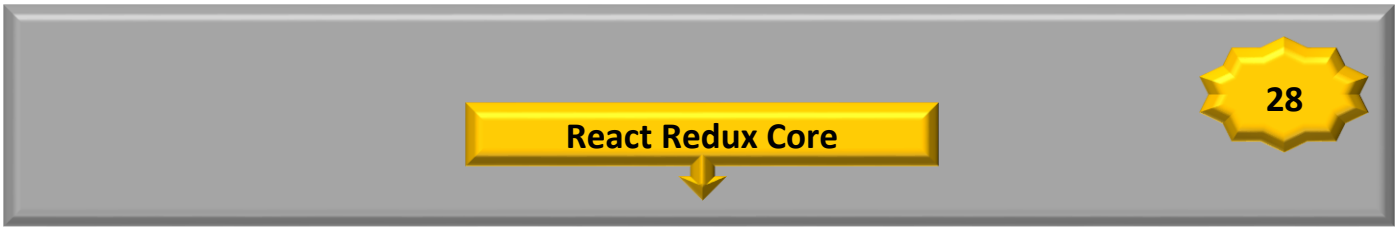
```
const initstate = {value: 0};

const counterReducer = (state = initstate, action) => {

 if (action.type === “increase”) {
  return {…state, value: state.value + 1} ;
 }

 if (action.type === “decrease”) {
  return {…state, value: state.value - 1} ;
 }

 return state ;
};

let store = createStore(counterReducer);
```

```
const counterApp = () => {
document.getElementById(“root”).innerText = store.getState().value ;
};
counterApp();

store.subscribe(counterApp);

document.getElementById(“increase”).addEventListener(“click” , () => {
consr action = {type: “increase”};
store.dispatch(action);
}
);

document.getElementById(“decrease”).addEventListener(“click” , () => {
consr action = {type: “decrease”};
store.dispatch(action);
}
);
```

١. نضيف payload لل action
٢. تزويد 5 وتنقيص 1

```
const initstate = {value: 0};

const counterReducer = (state = initstate, action) => {

 if (action.type === “increase”) {
  return {…state, value: state.value + action.payload.number } ;
 }

 if (action.type === “decrease”) {
  return {…state, value: state.value - action.payload.number } ;
 }

 return state ;
};

let store = createStore(counterReducer);

const counterApp = () => {
document.getElementById(“root”).innerText = store.getState().value ;
};
counterApp();

store.subscribe(counterApp);

document.getElementById(“increase”).addEventListener(“click” , () => {
consr action = {type: “increase”, payload:{number:5}};
store.dispatch(action);
```

```
}
);

document.getElementById("decrease").addEventListener("click" , () => {
consr action = {type: "decrease", payload:{number:1}};
store.dispatch(action);
}
);
```



وهو أصدار من Redux Core خاص ب React ولكن المفاهيم الأساسية في redux تبقى نفسها مع أختلاف طفيف في الصياغة

أولا نثبت Redux toolkit في المشروع :

```
# NPM
npm install @reduxjs/toolkit

# Yarn
yarn add @reduxjs/toolkit
```

ثانيا نثبت React redux في المشروع :

```
# If you use npm:
npm install react-redux

# Or if you use Yarn:
yarn add react-redux
```

سنأخذ نفس المثال وهو عداد :

ننشأ ملف نسميه store ثم ننشأ ملف داخله أسمه store.js ننشأ بداخله store مع Reducer مع state

store.js

```
import { createStore } from "@reduxjs/toolkit";

const initstate = {value:0};

const counterReducer = (state = initstate, action) => {
   return state;
};

const store = createStore(counterReducer);

export default store;
```

تختلف عن الجافاسكربت العادية بأنها تحتاج إلى import / export

الأن نحتاج الحالة state أن تصل إلى المكون الذي نحتاجه و أن تبقى محدثة وبتالي نعمل subscribe لل store وذلك من خلال provider وهي تعمل على الأستماع لل store و تقوم بمشاركته مع المكون الذي نريد أو على مستوى التطبيق ولكن الأفضل نشاركه إلى المكون الذي يحتاج للحالة state فقط

index.js

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
import store from './store/store';
import { Provider } from 'react-redux';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
   <Provider store={store}>
      <App />
   </Provider>
);
```

الأن حقنا المكون App بال store من خلال provider الان نحتاج لاستخدام الحالة state في المكون لذلك نستخدم useSelector((state) => {})

App.js

```
import React from 'react';
import { useSelector } from 'react-redux';

export default function App() {

 const count = useSelector( (state) => state.value);
 return (
   <div>
    {`counter : ${count}`}
    <br/>
    <button>increase</button>
```

```
    <br/>
    <button>decrease</button>
  </div>
 );
}
```

الن نريد عمل action وارساله عبر Dispatch لذلك نستدعي useDispatch ونرسل عبرها action

ونصنع Reducer حسب كل action في store ليصبح التطبيق

App.js

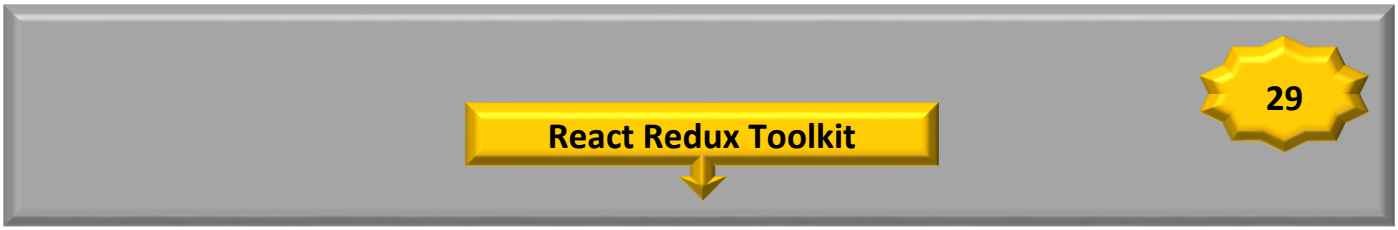
```
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';

export default function App() {

 const globalstate = useSelector( (state) => state );

 const dispatch = useDispatch();

 const increase = React.useCallback( () => {
  const action = {type: 'increase', payload: 4};
  dispatch(action);
 } , [dispatch])

 React.useEffect( () => { increase() }, [increase])

 const decrease = () => {
  const action = {type: 'decrease', payload: 2};
  dispatch(action);
 }

 const toggle = () => {
  dispatch({type: 'toggle'});
 }

 return (
  <div>
   {globalstate.toggleCounter ?
   <>
   {`counter : ${globalstate.value}`}
   <br/>
   <button onClick={increase}>increase</button>
   <br/>
   <button onClick={decrease}>decrease</button>
   </> :""
   }
   <br/>
   <button onClick={toggle}>toggle</button>
```

```
    </div>
  );
}
```

علما أنه قد قمنا بعمل action أول ما المشروع يعمل من خلال useEffect و useCallback

store.js

```
import { createStore } from "@reduxjs/toolkit";

const initstate = {value: 0, toggleCounter: true};

const counterReducer = (state = initstate, action) => {

   if (action.type === "increase") {
      return {...state, value: state.value + action.payload} ;
   }

   if (action.type === "decrease") {
      return {...state, value: state.value - action.payload} ;
   }

   if (action.type === "toggle") {
      return {...state, toggleCounter: !state.toggleCounter} ;
   }

   return state;
};

const store = createStore(counterReducer);

export default store;
```



وهو أحدث أصدار من Redux ويسمى Redux Toolkit خاص ب React ولكن المفاهيم الأساسية في redux تبقى نفسها مع أختلاف طفيف في الصياغة

ميزات Redux Toolkit عن Redux Core :

١. صياغة أفضل لل Reducer
٢. تحديث الحالة state دون الحاجة لوضع الحالة القديمة
٣. توفير مكتبة لل Api

التثبيت :

```
Add Redux Toolkit
npm install @reduxjs/toolkit

Add React Redux
npm install @reduxjs/toolkit react-redux
```

طريقة بناء المشروع :

أولا بناء شريحة Reducer :

تتضمن أسم الحالة الرئيسية و الحالة ودوال Reducer مع العلم أن الشريحة نضعها في متتغيير ويكون أسمه له دلالة عن عمل الشريحة وذلك من أجل استدعاء الشريحة و وضعها في ال Store

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

export const counterSlice = createSlice({
  name: 'counter',
  initialState: {
    value: 0,
  },
  reducers: {
    increment: (state) => {
      state.value += 1
    },
    decrement: (state) => {
      state.value -= 1
    },
    incrementByAmount: (state, action) => {
      state.value += action.payload
    },
  },
})

// Action creators are generated for each case reducer function
export const { increment, decrement, incrementByAmount } = counterSlice.actions

export default counterSlice.reducer
```

أولا لا يمكن تغيير أسماء key داخل الشريحة

وظيفة name هي وصول action لل Reducer تبعه الصح

وظيفة initialState وضع الحالة ويمكنك وضع الحالة في الخارج واستدعائها كمتغيير

وظيفة reducers وهو كائن يجمع داخله دوال Reducer

الدوال داخل Reducer يجب أن يكون لها أسم type لل action ونلاحظ أنها تعرف كأرو فانكشن وتأخذ بارمترين state / action و أنه لا يوجد حاجة لاستدعاء الحالة القديمة لتحديث الحالة الجديدة لان ال Redux تستخدم مكتبة Immer لتحديث الحالة

الأن نوزع دوال Reducer الموجودة في counterSlice.actions على متغييرات من نفس الأسم ليتم استخدام الدوال في الخارج

الأن نجعل الشريحة مع Reducer قابلة للاستخدام من الخارج ليتم وضعها في store

ثانيا ننشأ store متعدد يمكنه أخذ أكثر من شريحة :

يتم ذلك من خلال ({ reducer:{counter: counterslice} })configureStore ويمكنه أخذ من شريحة حيث يخزن الحالة لكل شريحة في داخله ويكون لها أسم مشابه للأسم داخل الشريحة

```
import { configureStore } from "@reduxjs/toolkit";
import counterReducer from "./counterSlice";


const store = configureStore({
   reducer:{
      counter: counterReducer
   }
});

export default store;
```

خزنا ال store في متغيير store وجعلناه قابل للاستخدام في الخارج من أجل عمل subscribe لل store وذلك من خلال provider وهي تعمل على الأستماع لل store

ثالثا عمل provider لل store :

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
import store from './store/store';
import { Provider } from 'react-redux';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
   <Provider store={store}>
      <App />
   </Provider>
);
```

رابعا عمل Dispatch للدوال Reducer :

يكفي هنا عمل import لدالة Reducer من ملف الشريحة واستخدامها

وهناك أختلاف في استدعاء الحالة حيث يجب وضع اسم الحالة التي أسميناها في store وفي الشريحة مثل state.counter لأن سوف يكون لدينا حالات متعددة

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

```
import { useSelector, useDispatch } from 'react-redux';
import { increment , decrement, incrementByAmount } from './store/counterSlice';

export default function App() {

 const counterstate = useSelector( (state) => state.counter );

 const dispatch = useDispatch();

 return (
  <div>
   {`counter : ${counterstate.value}`}
   <br/>
   <button onClick={ () => dispatch(increment())}>increase</button>
   <br/>
   <button onClick={ () => dispatch(decrement())}>decrease</button>
   <br/>
   <button onClick={ () => dispatch(incrementByAmount(4))}>increase 4</button>
  </div>
 );
}
```

شرائح متعددة :

سوف نضيف شريحة للـ auth ونقوم بـ action أبتدائي من خلال useEffect / useCallback ليصبح التطبيق النهائي

authSlice.js

```
import { createSlice } from "@reduxjs/toolkit";

export const authSlice = createSlice({
   name:"auth",
   initialState: {isloggedin:true},
   reducers:{
      login: (state) =>{
         state.isloggedin = true;
      },
      logout: (state) =>{
         state.isloggedin = false;
      }
   }
});

export const {login, logout} = authSlice.actions;

export default authSlice.reducer;
```

counterSlice.js

```
import { createSlice } from "@reduxjs/toolkit";

export const counterSlice = createSlice({
   name: "counter",
   initialState: {value:0},
   reducers: {
      increment: (state) => {
         state.value += 1;
      },
      decrement: (state) => {
         state.value -= 1;
      },
      incrementByAmount: (state, action) => {
         state.value += action.payload
      }
   }
});

export const {increment, decrement, incrementByAmount} = counterSlice.actions;

export default counterSlice.reducer;
```

store.js

```
import { configureStore } from "@reduxjs/toolkit";
import counterReducer from "./counterSlice";
import authSlice from "./authSlice";


const store = configureStore({
   reducer:{
      counter: counterReducer,
      auth: authSlice
   }
});

export default store;
```

App.js

```
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { login, logout } from './store/authSlice';
import { increment , decrement, incrementByAmount } from './store/counterSlice';

export default function App() {

  const globalestate = useSelector( (state) => state );
```

```
  const dispatch = useDispatch();

  const handellogin = React.useCallback((states) => {
    if(states) {
      dispatch(logout());
    } else {
      dispatch(login());
    }
  }, [dispatch]);

  React.useEffect( () => {
    handellogin(true);
  }, [handellogin]);

  // React.useEffect( () => {
  //   dispatch(logout());
  // }, [dispatch]);

  return (
    <div>
    {`auth : ${globalestate.auth.isloggedin}`}<br/>
      {globalestate.auth.isloggedin ?
      <>
      {`counter : ${globalestate.counter.value}`}
      <br/>
      <button onClick={ () => dispatch(increment())}>increase</button>
      <br/>
      <button onClick={ () => dispatch(decrement())}>decrease</button>
      <br/>
      <button onClick={ () => dispatch(incrementByAmount(4))}>increase 4</button>
      </>:""
      }
      <br/>
      <button onClick={ () => handellogin(globalestate.auth.isloggedin) }>{globalestate.auth.isloggedin ?
"logout" : "login"}</button>
    </div>
  );
}
```

قمنا ب action ابتدائي من خلال useEffect / useCallback وذلك عندما نقوم باستدعاء دالة خارجية للعمل dispatch

ولكن إذا لم نحتاج إلى استدعاء دالة خارجية يكفي استخدام useEffect لعمل dispatch مثل الكومينت

: extraReducers

وهي خاصية جديدة في Redux Toolkit توضع في شريحة التي نرغب بتغيير حالتها بناء على action من شريحة أخرى و نحط داخلها دوال شرائح التي نرغب باستماع أليها وعندما تنفذ هذه الدوال تنفذ بشريحة أخرى وتغيير الحالة لديها

```
import { createSlice } from "@reduxjs/toolkit";
import { login, logout } from "./authSlice";
```

```
export const counterSlice = createSlice({
  name: "counter",
  initialState: {value:0},
  reducers: {
    increment: (state) => {
      state.value += 1;
    },
    decrement: (state) => {
      state.value -= 1;
    },
    incrementByAmount: (state, action) => {
      state.value += action.payload
    }
  },
  extraReducers:  builder => {
      builder
      .addCase(logout, (state,action) => {
        state.value = 0;
        console.log(action.payload);
      }
      )
      .addCase(login, (state,action) => {
        state.value = 0;
        console.log(action.payload);
      }
      )
    }
});

export const {increment, decrement, incrementByAmount} = counterSlice.actions;

export default counterSlice.reducer;
```

وظيفة addcase() إضافة دوال Reducer خارجية لا تنسى عمل import لها تأخذ أول بارمتر أسم الدالة الخارجية و ثاني بارمتر دالة أرو فانكشن لتحكم بحالة state الشريحة نفسها و action الدالة المستمع لها

state عائدة للحالة الشريحة الحالية counterSlice

action عائدة لدالة المستمع إليها مثل login / logout

ملاحظة هامة : نصب الإضافة إلى الكروم الخاصة ب Redux وتسمى Redux dev tool

اسم الوظيفة	الرمز

ملاحظة: