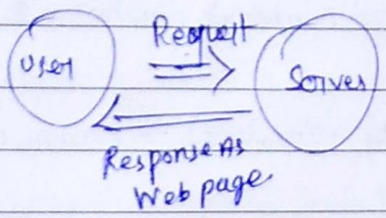
React:

A Javascript library for building user interface.

=> Runs smoothly, more interactive without lag.
=> Feel like mobileapp. becoz, highly interactive.

MobileApp & Desktopapp::

* Feels very "reactive": Things happen instantly. you don't wait for new page to load (or) action to start.
* Traditionally in web apps, you click a link & wait for a new page to load. You click a button wait for some action complete.



=> This request & response cycle we can breakup with Javascript.

Javascript:: => Run in browser on loaded page.
=> Manipulate HTML structure of the page.
=> No (visible) request to server.

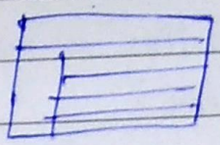
required no need to wait for a new HTML page as a response.

=> React.js will be a building block of javascript.

React.js::
=> client side javascript library.
=> All about building modern reactive user interface for the web.

=> Declarative. component focused approach.

Building single page Application (SPA's).



React can be used to control part of HTML pages (or) entire pages.

=> Widget approach on a multi page application. (some) page are still rendered on & served by a backend server.
=> React can also be used to control the entire front end of a web application.
=> 'Single Page Application' approach. Server only send one HTML page. Thereafter React takes over and controls the UI.

React.js Alternatives:-

Angular:- Complete component based UI framework packed with features. Uses Typescript. Can be overkill for small projects.

React.js:- Lean & focused component based UI library. Certain features are added via community packages.

Vue.js:- Complete component based UI framework. includes most core features. bit less popular than React & Angular.

Why react instead of Just Javascript?
=> Splitting application into maintable and manageble component.
=> html code inside js file.
=> Custom elements.

About this course & course outline:-

Basics & Foundation:

- Introducing key features.
- Components & Building UI.
- working with Events & Data "props" and "state"
- Styling React Apps & Components.
- Introduction into React Hooks.

Advanced Concepts:- (Building for production).

- Side effects, "Refs" & more React Hooks.
- React's Context API & Redux.
- Forms, Http Requests & "Custom Hooks"
- Routing, Deployment, NextJS & More.

Summaries & Refreshers:- (Optimizing Time).

- Javascript Refresher.
- ReactJS summary.

Next-Gen Javascripts.

=> Highly encouraged to use let & const for variables.

let is new "var"

let => Variable values.

const => Constant value.

Example:

```
let name = 'Naveen'
const name = 'Naveen'
```

Arrow functions:-

Normal function in JS:

```
function myFunc() {
    ...
}
```

Arrow Function:

```
const myFunc = () => {
    ...
}
```

=> No more issue with this keyword.

For one argument we can remove the parenthesis.

```
const myFunc = name => {
    ...
}
```

For empty parameter and multiple arguments we need to use parenthesis.

Below are similar code:

```
const multiply = number => number * 2

const multiply = (number) => {
    return number * 2
}
```

Export & Import: (Modules).

=> Used for writing modular code in multiple files.

person.js:

```
const person = {
    name: 'Max'
}
export default person.
```

utility.js:

```
export const clean = () => {...}
export const baseData = 10;
```

app.js:

```
import person from './person.js'
import prs from './person.js'
import {baseData} from './utility.js'
import {clean} from './utility.js'
import {baseData, clean} from './utility.js'
```

Imports default & only export of the file name export of the file name in the receiving file is up to you.

Default Export:

```
import person from './person.js'
import prs from './person.js'
```

Named exports:

```
import {smith} from './utility.js'
import {smth as Smith} from './utility.js'
import * as bundled from './utility.js'
```

Using Alias.

Understanding class:-

```
class Person {
    name = 'Max'          => Property
    call = () => {...}.   => Method
}.
```

Creating object:

```
const myPerson = new Person()
myPerson.call()
console.log(myPerson.name)
```

Inheritance.

```
class Person extends Master.
```

```
class Human {
    constructor() {
        this.gender = 'male';
    }
    printGender() {
        console.log(this.gender);
    }.
}
class Person extends Human {
    constructor() {
        super(); // mandatory
        this.name = 'Max';
        this.gender = 'female';
    }.
    printMyName() {
        console.log(this.name);
    }.
}
const person = new Person();
person.printMyName();
person.printGender();
```

Classes, Properties & Methods

=> Properties are like variables attached to classes/objects

ES6

```
constructor() {
    this.myProperty = 'value'
}
```

ES7

```
myProperty = 'value'
```

=> Methods are like functions attached to classes/objects.

ES6:

```
myMethod() {...}.
```

ES7

```
myMethod = () => {...}.
```

Example:

```
class Human {
    gender = "male";
    printGender = () => {
        console.log(this.gender);
    }
}

class Person extends Human {
    name = 'Max';
    gender = 'female';
    printMyName = () => {
        console.log(this.name);
    }
}

const person = new Person();
person.printMyName();
person.printGender();
```

Spread & Rest Operators:

→ Spread operator allows an iterable to expand in places where 0+ arguments are expected.

```
var variablename1 = [...value];
```

→ Spread operator used in many cases like, when we want to expand, copy, concat, with math object.

concat:

```
let arr = [1, 2, 3];
arr = let arr2 = [4, 5];
arr = [...arr, ...arr2];
console.log(arr);
// 1, 2, 3, 4, 5
```

Copy:

```
let arr = ['a', 'b', 'c'];
let arr2 = [...arr];
```

Expand:

```
let arr = ['a', 'b'];
let arr2 = [...arr, 'c', 'd'];
console.log(arr2);
```

Math functions:

```
let arr = [1, 2, 3, -1]
console.log(Math.min(...arr));
```

Spread operator with objects:

```
const user1 = {
    name: 'Jen',
    age: 22
};
const clonedUser = {...user1};
console.log(clonedUser);
```

```
const user1 = {
    name: 'Jen',
    age: 22
}

const user2 = {
    name: 'Andrew'
    location: 'philadelphia'
};

const mergedUser = {...user1, ...user2};
console.log(mergedUsers);
```

Output

```
{ name: 'Andrew', age: 22,
  location: 'Philadelphia' }
```

Rest Parameter:

Rest parameter is an improved way to handle function parameter allowing us to more easily handle various input as parameter in a function. The rest parameter syntax allows us to represent an indefinite number of arguments as an array.

Example:-

```
function (a,b) {
    return a+b;
}
console.log (fun(1,2));
console.log (fun(1,2,3,4,5));
```

↳ No error will be thrown, only two arguments will be considered.

```
function fun(...input){
    let sum=0
    for (let i of input){
        sum+=i;
    }
    return sum;
}
fun(1,2)
fun(1,2,3)
fun(1,2,3,4,5).
```

Note:-

Rest parameter have to be the last argument as it's job to collect all the remaining arguments into an array.

Below is wrong:-

```
function fun(a,...b,c){
    ...
}
```

Right one:-

```
function fun(a,b,...c){
    console.log(`${a} ${b}`);
    log(c);
    log(c[0]);
    log(c.length)
    log(c.indexOf("lions"));
}
fun(1,2,3,4,5)
```

→ Rest.

```
const myfunc = (...args) => {
    return args.filter(e => e===1);
}
console.log (myFunc[1,2,3,4,1,5]));
```

Destructuring:-

Easily extract array elements (or) object properties and store them in variables.

Array Destructuring:-

```
[a,b] = ['Hello','Max']
console.log (a) // Hello
console.log (b) //Max.
```

Object Destructuring:-

```
{name} = {name:'Max',
          age:28}
console.log (name); //Max
console.log (age); //Undefined.
```

Example:-

```
const numbers = [1,2,3];
[num1, , num3] = numbers;
console.log (num1, num3);
```

Reference & Primitive type Refresher:-

```
Const number = 1
Const num2 = number. => Here we are doing the copy of the number.
```

In javascript, number, string, booleans are primitive type.

```
const person = {
    name: 'Max'
};
const secondPerson = person;
person.name = 'Manu';
console.log (secondPerson);
```

Here its copying the address pointer ⇒ Not copying the actual object.

For copying the actual object we can use spread operator:

```
const secondPerson = {
    ...person
};
```

Refreshing Array Function:

```
const numbers = [1, 2, 3]
const doubleNumArray = numbers.map(
    (num) => { return num * 2; });
```

React Basics & Working with Components

=> Component Driven User Interfaces.
=> Building Interactive & Scalable UI's.

Module Content:-

=> React core Syntax & JSX.
=> Working with Components.
=> Working with Data.

What are components & why is React All About them?

=> React is Javascript library for building user interfaces.
=> HTML, CSS & Javascript are about building user interface as well.
=> React makes building complex, interactive & reactive user interfaces simpler.
=> React is all about components.
=> Because all user interfaces in the end are made up of components.

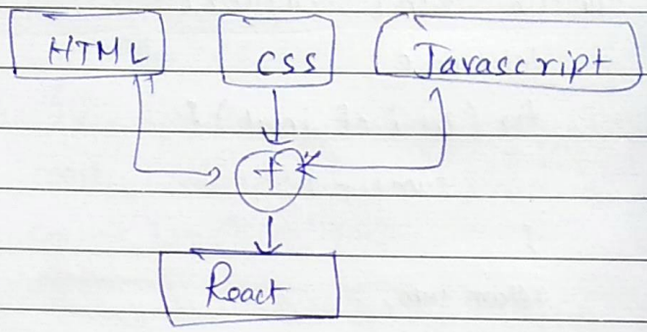
=> Component are reusable. It comprises of HTML, CSS, JS.

Why components?

- Reusability. => Don't Repeat yourself
- Separation of concern. => Don't do too many things in one & same place.

=> Split big chunks into multiple smaller functions.

How is a component Built-?



React allows to create reusable & reactive components consisting of HTML & javascript (& CSS).

↓

Declarative Approach.

↓

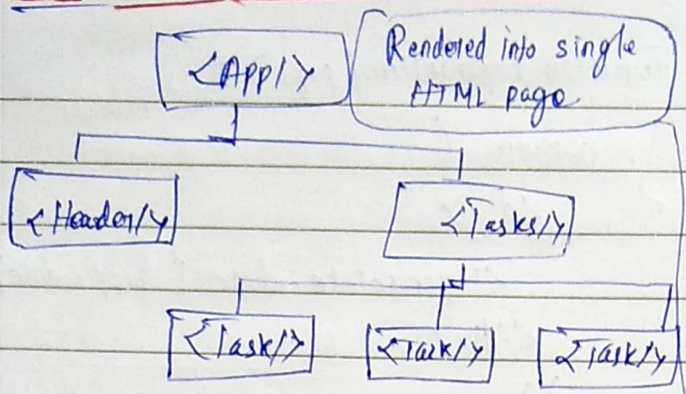
Define the desired target states & let react figure out the actual Javascript DOM instructions.

Build your own, custom HTML elements.

Creating react Project:-

=> npx create-react-app react-pet-app.

You Build a component Tree:-



→ Here we are building component tree.
→ Only top most component is rendered into HTML page. This top most component is multiple sub components.
→ with the help of that ReactDom render instruction.
→ Component is just a java script function.

Creating custom component:-

ExpenseItem.js:

```
function ExpenseItem() {
    return <h2> Expence Items !! </h2>
}.
export default ExpenseItem;
```

App.js:

```
import ExpenseItem from "./components/
                        ExpenseItem";
function App() {
    return(
        <div>
            <h2>---</h2>
            <ExpenseItem></ExpenseItem>
        </div>
    );
}
export default App;
```

CSS

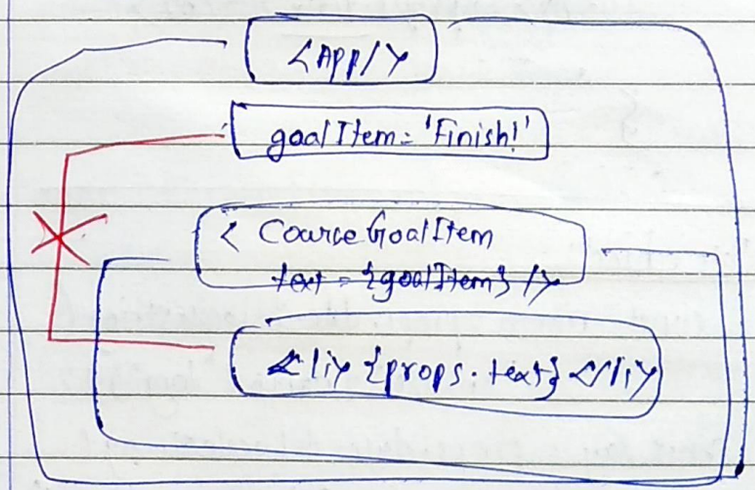
```
<div className="expense-item">
    ---
</div>
```

Here className is react specified. The whole code looklike (HTML). But it is not, It belongs to React, react will interpret and convert accordingly.

Outputing Dynamic Data & working with Expression in JSX :-

```
import './ExpenseItem.css'
function ExpenseItem() {
    const expenceDate = new Date();
    return (
        <div>
            <div>{expenseDate.toISOString()}</div>
        </div>
    );
}
```

Passing Data via "Props"



Component can't just used data stored in other components.

```
function App() {
  const expenses = [
    {
      id: "e1"
      title: "Toilet Paper",
      amount: 94.92
      date: new Date (7
    },
    {...
    },
  ];
  return ( <div <ExpenseItem
      title = {expenses[0].title}.
      amount= {expenses[0].amount}
      date ={expenses[0].date}
    ></ExpenseItem>);
}
```

Here we can do Hard coding also

ExpenseItem.js:

```
function ExpenseItem(props) {
  <div> {props.date.toISOString()}>
  <h2> {props.title} </h2>
}
```

Date object:

```
const month =props.date.toLocaleString(
      "en-US",{month: "long"});
const day = props.date.toLocaleString(
      "en-US", {day: "2-digit"});
const year = props.date.getFullYear();

<div>{month}</div>
<div>{day}</div>
<div>{year}</div>
```

Splitting components:

```
function ExpenseItem(props) {
  return (
    <div> ...
      <ExpenseDate date={props.date}/>
    </div>
  );
}
```

The concept of composition:-

* The approach of building the user interface from smaller building block is called composition.

Card:

This for accepting the class name in Card element

```
function Card(props) {
  const classes = "card " + props.className
  return (<div className={classes}>
      {props.children}
    </div>);
}
export default Card;
```

For accepting the elements inside the <Card> element

```
<card className="expenses">
  <div>
    :
  </div>
</card>
```

For JSX code:

```
return (
  <div>
    <h2>Let's get started!</h2>
    <Expenses items={expenses}/>
  </div>
);
```

we can write like:-

```
import React from 'react';
. . .
. . .

return React.createElement('div', {},
  React.createElement('h2', {},
    'Let's get started!'),
  React.createElement(Expenses, {items:
    expenses}, ' '));
```

order the need

React library is still used when we write JSX code.

Organizing components:-

```
import Card from "../UI/Card";
```

Go up one level

Components.

- Expenses.
- UI.

Every UI in the end up is made up of multiple building blocks (=components) hence it makes sense to think about User interface as "combination of components"

Here Declarative means:

You define the desired out come. (a target UI) and let the library figure out the steps.

Pass Data b/w components:-

we build our own HTML element in the end, hence you can also define your own attributes (called props in React world).

Output Dynamic data in React component:

You can output any result of any JS expression by using this spacial feature.

React States & Working with Events

Module Content:

- ⇒ Handling Events.
- ⇒ Updating the UI & working with State.
- ⇒ A closer look at Component and state.

Listening to Events & Working with Event Handlers:

```
Const ExpenseItem = (props) => {
```

Event listener name end up with Handler

```
  const clickHandler = () => {
    console.log('Clicked!');
  }.
  return (.....
    <button onClick={clickEvent}>
      Change Title </button>
  );
```

How Component Functions are Executed:-

We need to implement a scenario where when button click, we need to change the title. So cee implemented like below.

```
const ExpenseItem = (props) => {
    let title = props.title;
    const clickHandler = () => {
        title = 'updated!';
    };
    return (
        ....
        <h2>{title}</h2>
        ....);
};
    <button onClick={clickHandler}>
        Change Title </button>
```

* Here <h2> won't get updated when clicking button. Becoz, ExpenseItem function executed only once. All the component function will be executed only once when page is loaded then. When we want to change the element value.. like this scenario, we need to use state..
* When the variable changes react does not care about that as per this current code.
* This can be handled by "State" in React

Working with State:-

useState()

This function allows to define values as state where changes to these values should reflect in the component function being called again.

This the key difference for the regular variable.

=> should be directly inside the component function.

=> useState is a so called React Hook.

=> All hooks are starting with "use" in their name.

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

Importing useState function from react library ({}) By using this, we can extract specific variable or function.

```
const [title, setTitle] = useState(props.title);
```

we are assigning initial value to special variable.

* Returns array with two value, one is variable, another is updating function.
* When calling setTitle function, not only variable getting updated, it calls the actual component function when useState is defined.

Example:

```
import React, {useState} from 'react';
...
const ExpenseItem = (props) => {
    const [title, setTitle] = useState(props.title);
    const clickHandler = () => {
        setTitle('updated!');
    };
    return (
        ....
        <h2>{title}</h2>
    );
};
```

Tells the React that useState registered, component is re evaluated.

* UseState() function register some state to component intance from which it been called.
* Component is re evaluated when Update function called.
* State is sperated on a per component intance basis
* When re evaluate the component function, useState won't get reinitialized, it make use of the previously initialized state.
* State do the reactivity to the component, without state our user interface wont change.

State can be updated in many way:
* Update the state based on Http response we got back.
* Based Http request that complete. Because a time expired.

Listening to user input:

```
const ExpenseForm = () => {
  const titleChangeHandler = (event) => {
    console.log(event.target.value);
  };
  return (
    ....
    <input type='text'
      onChange={titleChangeHandler} />
  );
```

Working with multiple state:-

```
const ExpenseForm = () => {
  const [title, setTitle] = useState('');
  const [amount, setAmount] = useState('');
  const [date, setDate] = useState('');
  const titleChangeHandler = event => {
    setTitle(event.target.value);
  };
  ...
}
```

These all states are independent to each other

Using one state intead:-

```
const ExpenseForm = () => {
  const [userInput, setUserInput] =
    useState({
      title: '',
      amount: '',
      date: ''});
  const titleChangeHandler = (even) => {
    setUserInput({
      ...userInput,
      title: event.target.value,
    });
  };
```

This will override the title field in userInput object

Here, this solution will work. But it is not a best practice. Bcoz, whenever we update the state, when are depending on the previous value.
=> And we are copying the other two values.

Updating State that Depends on the Previous State:

React schedule a state update does not perform them instantly. If multiple updates happen on state, previous approach does not gurantee the we are working lates state.

For this problem we can use arrow function with previous state as argument.

```
setUserInput ((previousState) => {
  return { ...previousState, title: event.target.value };
});
```

Handling Form Submission.

```
<form onSubmit={submitHandler}>
</form>

const submitHandler = (even) => {
  event.preventDefault();
  const expenseData = {
    title: title,
    amount: amount
  };
  console.log(expenseDate);
};
```

prevent page from loading again.

Two way Binding.

```
<input type="text"
  value={enteredTitle}
  onChange={titleChangeHandler} />
```

Here, we are not just listening to the changes to the input, we also feed back the changes to the input. This called twoway binding.

Child to Parent Component Communication: (Bottom up)

We learned to pass the data b/w component using "props". This is the parent to child communication.

Now we have to look at child to parent component communication.

Example: NewExpense.js:

```
const NewExpense = (props) => {
  const saveExpenseDataHandler =
      (enteredExpenseData) => {
    const expenseData = {
      ...enteredExpenseData,
      id: Math.random.toString(),
    };
    props.onAddExpense(expenseData);
  };
  return (
    <div className="...">
      <ExpenseForm onSaveExpenseData=
        {saveExpenseDataHandler} />
    </div>
  );
};
```

App.js:

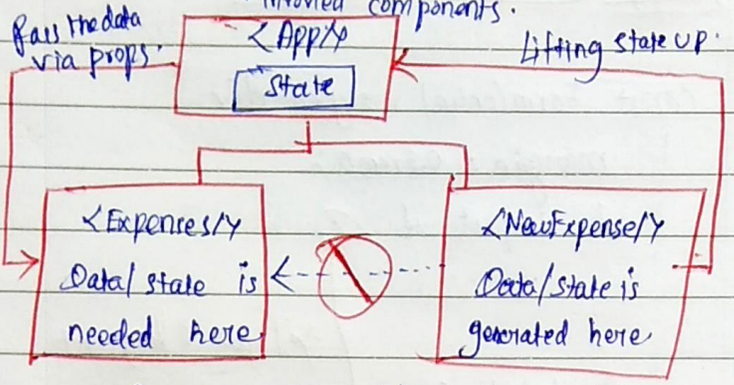
```
const addExpenseHandler = expense => {
  console.log(expense);
};

return (
  <div>
    <NewExpense onAddExpense={addExpenseHandler} />
  </div>
);
```

* The idea, here is to defining the method in parent.
* And pass the method reference to to child via "props".
* In the child, calling the passed method reference wherever applicable.

Lifting the state up:-

This component have access to both invoked components.



* Communicating with siblings is not possible here. So we are lifting the state up, & communicate with App component.

Rendering List & Conditional content.

Module content:-

=> Outputting dynamic list of content
=> Rendering content under certain condition.

Rendering List of Data.

```
{ props.items.map (expense) => {
    <ExpenseItem
        title = {expense.title}
        amount = {expense.amount}
        date = {expense.date}
    />
})}
```

Outputting conditional Element:

```
{ filteredExpenses.length === 0 ? (
    <p> No expense found </p> ) :
    ( ... ) }.
```

or

```
{ filteredExpenses.length === 0 &&
    <p> No expenses found </p> }.
{ filteredExpense.length > 0 &&
    ... }.
```

We can do the alternative for this, we can store Jsx content to variable.

```
let expenseContent = <p> No expenses found. </p>;
```

We should add the spelial "key" prop to list Jsx elements.
It's required for React to correctly identify and update (if needed) the list elements.

Styling Component.

Module Content:

- Conditional & Dynamic Styles.
- Styled Components.
- Css Modules.

Inline Style

Default html element accept style as attribute. In react, values for a style attribute will set as object.

```
<div
    className = "chart-bar__fill"
    style = {{ height: barFillHeight }} />
</div>
```

Example:-

```
<label style={{ color: !isValid ? 'red' : 'black' }}>
```

Course Goal </label>.

This the is object for style.

Example style objects:-

```
{ borderColor: !isValid ? 'red' : 'black',
  backgroundColor: !isValid ? '...' : '...' }
```

Setting css classes Dynamically:

In css file:-

```
.form-control.invalid input {
  border-color : red.
  background : rgb(...);
}
.form-control.invalid label {
  color : red ;
}.
```

In component function

```
<div className={`form-control ${!isValid ? 'invalid' : ''}`}>
```

Template literal backtick

${ } => accept js expression.

```
  <label> ...
  <input> ...
</div>.
```

Styled Components:

```
import styled from 'styled-components';
const Button = styled.button`
  font : inherit ;
  padding : 0.5rem 1.5rem;
  color : white;
```

For pseudo selectors we can use '&'

```
& :focus {
  outline : none ;
}
```

Styled component make sure css not affecting another component.

All the props set on the buttons work exactly the same way.

Styled component Dynamic:-

```
const FormControl = styled.div`
  margin : 0.5rem 0;
  & input {
    ...
  }
  & label {
    ...
  }
`;
```

=> Input Element

Here we are having two component in single file.

```
const CourseInput = props => {
  ...
  return (
    ...
    <FormControl className={!isValid && 'invalid'}>
      ...
    </FormControl>
  );
```

Make use of props in styled component:-

```
const FormControl = styled.div`
  & label {
    color: ${(props) => (props.invalid ? 'red' : 'black')};
  }
```

```
const CourseInput = ... {
  ...
  <FormControl invalid={!isValid}>
  ...
  </FormControl>
```

Media Queries in Css For different devices :-

```
const Button = styled.button`
  ...
  width: 100%;
  ...
  @media (min-width: 768px) {
    width: auto;
  }
`;
```

Use Css Modules-

* It takes the CSS classes and a CSS file, basically changes those class names to be unique.
* It converts all the classes in CSS file to be unique
* Css Modules ensure that css files are scoped to the component it imported into.

Example:

Convert the file name end with Button.module.css.

```
import styles from './Button.module.css';

const Button = props => {
  return (...
    <button className={styles.button}>
  );
};
```

Css Module Dynamic Css:-

```
<div
  className={`${styles['form-control']}
  ${!isValid && styles.invalid}`}>
  ...
</div>
```

MediaQueries in Css Module:

Button.module.css

```
.button {
  width: 100%;
  ...
}
@media (min-width: 768px) {
  .button {
    width: auto;
  }
}
```

Component Function:

```
<button className={styles.button}>
```

Debugging React App

Finding & Fixing Errors.

Module content:

- Understanding Error Messages
- Debugging & Analyzing React App
- Using the React Dev Tools.

Understanding React Error Messages:

1. JSX elements must be wrapped in an enclosing tag.
2. 'addGoalsHandler' is not defined =>
3. This is sometime due to typo.

Warning Encountered two children with the same key. Keys should be unique so that component maintain their identity across updates.

Custom wrapper component & CSS:

```
import styles from './Card.module.css';
const Card = (props) => {
  const classes = `${styles.card} ${props.
                  className}`;
  return <div className={classes}>
          {props.children}
         </div>;
};
export default Card;
```

```
<button
  ...
  type={props.type || "button"}.
```

`+` ensures age is a number.

```
if (+age < 1) {
  return;
}
```

Fragments, Portals & Refs.

Module Content:

- JSX Limitation & Fragments.
- Getting a cleaner DOM with Portals.
- Working with Refs.

JSX Limitations & Workaround.

- Cannot return more than one "root" JSX element.

```
return (
  <h2>Hi there!</h2>
  <p>This does not work :-(</p>
);
```

Simple solution is, wrapping JSX elements with (like any elements which wraps).

```
return (
  <div>
    <h2>Hi there!</h2>
    <p>This is works</p>
  </div>
);
```

If array of JSX elements we need a key in elements.

Other alternative is using array.

```
return [
  errorA && (<ErrorModal key ... />),
  <Card key ... />];
```

Using Wrapper component

```
const Wrapper = props => {
  return props.children;
};
export default Wrapper;
```

Introducing Fragments:-

```
return (
  <React.Fragment>
    <h2> Hi there </h2>
    <p> .... </p>
  </React.Fragment>
);
```

(or)

```
return (
  <>
    <h2> .... </h2>
    <p> .... </p>
  </>
);
```

It's an empty wrapper component. It does not render any real HTML element to the DOM. But it fulfills React's JSX Requirement.

Using Fragment:-

```
import React, { useState, Fragment } from 'react';

. . . .

return (
  <Fragment>
    <AddUser ... />
    <UserList ... />
  </Fragment>
);
```

Understanding React Portals:

```
return (
  <React.Fragment>
    <MyModel />
    <MyInputForm />
  </React.Fragment>
);
```

Using react portal:

```
<section>
  <h2> Some other content </h2>
  <div class Name="my-model">
    <h2> A Model Title! </h2>
  </div>
  <form>
    <label> Username </label>
    <input type="text" />
  </form>
</section>
```

Semantically and form a "clean HTML structure" perspective having this nested model is not ideal. It's an overlay to the entire page after all (that's similar for side-drawers, other dialogs etc).

It's a bit like styling a like <button> and adding an event listener to it. It will work. but it's not a good practice.

```
<div onClick={clickHandler}>
  Bad Button </div>.
```

By using React portal, we can control React to html conversation. (JSX)

On which order element has to ~~be~~ created.

Portal needs two things :-

⇒ Place you wanna port the component to.
⇒ Let the component know that it should have a portal to that place.

index.js:

```
<body>
  <div id="backgroup-root"></div>
  <div id="overlay-root"></div>
  ....
</body>.
```

Component.js:

```
const Backdrop = props => {
  return <div className = {classes.backdrop}
  ...
}

const ModelOverlay = (props) => {
  return (
    ...);
};

const ErrorModel = (props) => {
  return (
    <React.Fragment>
      {ReactDOM.createPortal (
        <Backdrop .../>,
        document.getElementById('backgrp-root');
      )}
      {ReactDOM.createPortal (
        <ModelOverlay
          .../>,
        document.getElementById('...')
      )}
    </React.Fragment>
  );
};
```

Working with "ref"s :-

Ref ⇒ Reference.
⇒ Allow us to get access to other DOM elements & work with them.
⇒ Reading the value from html elem.

Example:-

```
import React, {useState, useRef}
    from 'react';
const AddUser = props => {
  const nameInputRef = useRef();

  Handlerfanction() {
    const username = nameInputRef.
      current.value;
  }.
```

Html Element:

```
<input ref={nameInputRef} />
```

Module

Advanced : Handling Side Effects; fine Using Reducers & Using Context API.

Module Content:

* Working with side effects.
* Managing more complex state with Reducers.
* Managing App-Wide (or) Component-Wide state with Context.

What are "Side Effects" & Introducing useEffect:-

Main Job:

* Render UI & React to user input.
  Evaluate & Render JSX Manage state & Props React to (user) Events & Input.
* Reevaluate component upon state & prop changes.
* This all is baked into "React via tools" and features covered in this course. (ie. useState(), Hook, props etc).

Side Effects: Anything Else.

* Stored Data in Browser storage.
* Send Http requests to Backend server.
* Set & Manage Timers.

⇒ These are not related to bringing something to the UI.

These tasks must happen outside of the normal component evaluation and render cycle - especially since they might block(or) Delay rendering. (Http requests).

(No side effects in component function)

Using the useEffect Hook:-

```
function App() {
  const [isLoggedIn, setIsLoggedIn] = useState(false);
  const isUserLoggedIn = localStorage.
                         getItem('isLoggedIn');
  if (isUserLoggen === '1') {
    setIsLoggenIn(true);
  }
}
```

Error: Too many rerenders.

```
const loginHandler = (...) => {
  setIsLoggenIn(true);
}
```

It will make a infite loop.

The thing is if user ~~already~~ logg. ive need to go to main page.

Since the current code producing the error, we can focus on to useEffect function.

Here we can use useEffect:

```
useEffect( () => {
  const isUserLoggedIn = localStorage.getItem
                         ('isLoggedIn');
  if (isUserLoggedIn === '1')
    setIsLoggedIn(true);
}, []);
```

Since no dependency is placed the effect run only once.

This useEffect will execute after the after the component function executed.

Handling side effect with useEffect() Hook:

```
useEffect(() => {...}, [dependencies]);
```

A function that should be executed AFTER Every component evaluation If the specified dependencies changed.

Dependencies of this effect - Function only runs if the dependencies changed.

useEffect with Dependencies:

```
useEffect( () => {
  setFormIsValid (
    ....
  );
}, [enteredEmail, enteredPassword]);
```

whenever, email or password changes this func will execute