create react app → recommended tool to create react projects

↓

can be installed globally using npm.

or can be used for one time initialization of project using npx

To have the same folder structure

-- scripts-version 1.1.5 : impacts the structure of the project

npm start to run the development server

Folder structure :

i) lock files → locking in the versions of the dependencies we're using.

ii) package.json → Defines the general configuration of the project including dependencies and their versions.

↓

react-scripts : pkg offering all build workflow, development server, next-gen JS support

↓

Scripts are also defined : These can be run by

using npm run followed by the script name.

iii) node-modules: dependencies and sub-dependencies

iv) public: root folder which gets served by the web server in the end.

v) src: script files destination

↓

a) index.js: The JS file with access to root element of the HTML file of the project, and renders our application, using the App element imported from a App.js file

b) App.JS: file where mapority of our would be done

↓

and at start, the only component is present, the code in the div can be replaced.

↓

logo.svg can be discarded.

c) App.css: css for the project. Delete all except .App. index.css

d) registerServiceWorker.js file → precache script files

e) App.test.js

→ Render root component, and in the root component, we can nest all the other components, our app might need. These components can further nest components.

→ One way of creating React Component using classes

```
class App extends Component {
    render() {
        return (
            <div className="App">
                <h1> Hi, I'm a React App </h1>
            </div>
        );
    }
}

export default App;
```

(syntactical sugar)

JSX

on importing the whole file, we import the marked class

→ default export the root component

→ Need to import React, { Component } from 'react',

And then App Component from App.js file

→ render method called to return (React component) renderable html code.

done through createElement ← or using JSX

necessary for any react component

•) understanding JSX

return React.createElement (element / component to be rendered,

configuration (JS object),

children (nested components / elements))

```
return React.createElement('div', null, 'h1',
    'Hi, I'm a React App')
```

↓

in this format both will be regarded as text

↓

need React elements to be passed as children.

```
return React.createElement('div', null,
    React.createElement('h1', null,
        'Hi, I'm a React App'));
```

↓

this does not contain the css applied to it. The class needs to be applied.

↓

passed through configuration argument in Javascript object format, with key being the attribute & value being the value of the particular attribute.

↓ → this is why React is imported even when JSX is used.

```
return React.createElement('div', { className: 'App' },
    React.createElement('h1', null,
        'Hi, I'm a React App'));
```

↓

this is what the JSX code above compiles down to

•) JSX Restrictions : i) class can't be used, as it is a reserved keyword in JS.

↓

className is used.

ii) only one root element per component. (good practice as well)

(loosened with React 16, adjacent elements can be returned)

•) Creating a functional component → another method of creating components

↓

function

↓

In its simplest form component is just a function that returns some JSX.

↓

```
const person = () => {
    return <p> I am Shiv </p>
};

export default person;
```

↓

Component having an uppercase starting letter since lowercase ones are reserved for HTML elements.

↓

Component can be used tag pair or self-closing.

→ Dynamic content in JSX to be written in `{}` using javascript expressions. Simple one line expressions. Can call functions, which then execute complex code

→ Props : making components configurable → basically passing information from parent node to child node using attributes.

↓

done using props argument in the render method for functional components and accessed using this.props for class components.

↓

normal JS object

```
import React from 'react';

const person = (props) => {
    return <p>I'm {props.name} and I am {props.age} old</p>
};

export default person;
```

```
import React from 'react';
import './App.css';
import Person from './Person/Person'


function App() {
  return (
    <div className="App">
      <h1> Hi, I'm a React App</h1>
      <p> This is really working!</p>
      <Person (JSX attribute) name: string
      <Person name="Agrima" age="23"> My hobbies are</Person>
      <Person name="Suresh" age="27"/>
    </div>
  );
}

export default App;
```

The content between the opening and closing tags, called the children of the component doesn't get rendered automatically. Are accessed using

props.children

↓

Need to be explicitly used inside the component it is being passed to.

*) State : method for data mgmt. from inside the component.

Available for components that extend Component class i.e. class based components, but now also available with functional components using hooks.

↓

Any change to state leads to automatic re-rendering of the component, and potentially update the DOM.

→ handling events with methods

onClick and not onclick

↓

We add the onClick attribute to the html button element with a reference to the function which needs to be executed.

↓

There are many different Event Handlers that can be understood from React docs.

In the function, we don't change the state directly, rather use setState function, and assign the new value of the keys in the state.

↓

React pays attention mainly to two types of concepts to update DOM, i.e. state and props.

```
class App extends Component {

  state = {
    persons : [
      {name:'Shiv',age:22},
      {name:'Agrima',age:23},
      {name:'Suresh',age:27}
    ]
  }

  switchNameHandler = () => {
      this.setState({
        persons : [
          {name:'Shivam',age:22},
          {name:'Agrima',age:23},
          {name:'Suresh',age:28}
        ]
      })
  }

  render() {
    return (
      <div className="App">
        <h1> Hi, I'm a React App</h1>
        <p> This is really working!</p>
        <button onClick={this.switchNameHandler}>Switch Name</button>
        <Person name={this.state.persons[0].name} age={this.state.persons[0].age}/>
        <Person name={this.state.persons[1].name} age={this.state.persons[1].age}> My hobby is Racing</Person>
        <Person name={this.state.persons[2].name} age={this.state.persons[2].age}/>
      </div>
    );

  }
}
```

*) React hooks → collection of functions exposed to us by React, which can be used in functional components

↓

need to be imported separately.

↓

example hook : useState (imported from 'react');

pass initial state to usestate, it then returns

an array with two elements, always

↓

first element : current state

second element : fn. that allows us to

update the current state

```
const App = props =>{

  const [personsState, setPersonsState] = useState({persons : [
          {name:'Shiv',age:22},
          {name:'Agrima',age:23},
          {name:'Suresh',age:27}
        ]
  })

  const switchNameHandler = () => {
          setPersonsState({
            persons : [
              {name:'Shivam',age:22},
              {name:'Agrima',age:23},
              {name:'Suresh',age:28}
            ]
          })
      }


  return (
          <div className="App">
            <h1> Hi, I'm a React App</h1>
            <p> This is really working!</p>
            <button onClick={switchNameHandler}>Switch Name</button>
            <Person name={personsState.persons[0].name} age={personsState.persons[0]
            <Person name={personsState.persons[1].name} age={personsState.persons[1]
            <Person name={personsState.persons[2].name} age={personsState.persons[2]
          </div>
        );
}
```

The onclick event handler method can again be defined inside the component function

An important difference b/w state mgmt - using class based components & components using hooks,

is that hooks don't automatically other properties in the state object, rather straightaway sets the elements. ∴ a different variable should be used for each state property, and use its specific state manager

*) Stateful and Stateless components

↓ manages state

↓ does not manage state

*) function references can also be passed using properties.

↓

passing click handlers to children to modify parent state → common pattern

passing function arguments is the main challenge for current state mgmt, and for use by children as well, since we can't call the function.

↓

This is done in two ways: a) using bind.

this.switchName.handler.bind(this, "maximilian");

↑ to specify context

A way to associate variables to function references.

list of arguments passed to the function.

b) Alternative view could be to pass an anonymous function that calls this eventhandler method, passing the arguments.

↓

(*) This is inefficient

→ should be single line, arrow fn. adds implicit return hitz so just need to pass function call with arguments.

```
render() {
  return (
    <div className="App">
      <h1> Hi, I'm a React App</h1>
      <p> This is really working!</p>
      <button onClick={this.switchNameHandler.bind(this,"Shiva")}>Switch Name</button>
      <Person
      name={this.state.persons[0].name}
      age={this.state.persons[0].age}/>
      <Person
      name={this.state.persons[1].name}
      age={this.state.persons[1].age}
      click={() => this.switchNameHandler("Shivam")}>
      My hobby is Racing
      </Person>
      <Person
      name={this.state.persons[2].name}
      age={this.state.persons[2].age}/>
    </div>
  );
```

→ 1st method

→ 2nd method

*) Two way binding and handling input fields.

To respond to change in input field, we can define another Event handler in the parent component. This will change the state of the component based on the input, in the input field.

The eventhandler function will take as argument event which triggered it, and through its target's value update the required value.

```
render() {
  return (
    <div className="App">
      <h1> Hi, I'm a React App</h1>
      <p> This is really working!</p>
      <button onClick={this.switchNameHandler.bind(this,"Shiva")}>Switch Name</button>
      <Person
      name={this.state.persons[0].name}
      age={this.state.persons[0].age}/>
      <Person
      name={this.state.persons[1].name}
      age={this.state.persons[1].age}
      click={() => this.switchNameHandler("Shivam")}>
      My hobby is Racing
      </Person>
      <Person
      name={this.state.persons[2].name}
      age={this.state.persons[2].age}/>
    </div>
  );
```

globally scoped

*) Adding css → needs to be explicitly imported into the JS file. and accordingly defined CSS needs to be used

*) working with inline styles

define a style object in the render fn., and pass it to the style attribute of the element.

→ locally scoped

↓

all values will be in string format

*) Rendering content conditionally

JSX → Javascript (alias for the createReactElement, where for children, we could take a function, which needs to

return a React Component)

↓

this can be one line code which returns a react component or a function call.

↓

same is the case with JSX. Looks like HTML but inside the {}, the single line rule will work.

↓

This can be used to employ a ternary condition, and accordingly render elements.

↓

(*) The {} should return a JSX element, and inside the {} should be single line.

↓

Not necessary to be limited by one line, can call a function, or can take a variable which can store result of the applied conditional logic.

*) Outputting lists → React is able to render a list of JSX elements. So we need to map the list of elements we want to put on page to their corresponding react components.

{ [{...}, {...}, {...}] }

this.state.persons.map(person => {
return (<Person
name = {person.name},
age = {person.age}
/>)
})

```
return (
  <div>
    {
      [<Person
      name={this.state.persons[0].name}
      age={this.state.persons[0].age}/>,
      <Person
      name={this.state.persons[1].name}
      age={this.state.pe
      click={() => this.switchNameHandler("Shivam")}
      changed={this.nameChangeHandler}>
      My hobby is Racing
      </Person>,
      <Person
      name={this.state.persons[2].name}
      age={this.state.persons[2].age}/>]
      }
  </div>
);
} else {
  return null;
}
```

(property) App.switchNameHandler: (newName: any) => void

*) We could write a deletePersonHandler method and pass it on to all persons. In the method, we should operate on our deep copy rather the shallow copy got via the state. This can be done by calling slice w/o arguments on the state prop. or using the spread operator

(more modern and a better approach)

*) key is important kind of data that we should take care of when dealing with rendering of lists of data, to efficiently track elements that changed from the ones that didn't. w/o key property, it will just render the whole list, and that can be super inefficient w.r.t long and complex lists. ∴ key property should be defined on each individual component in a list.

↓

an id or a unique identifier

*) To work with the nameChange Handler for the entire list, we need a way to identify the particular id. This needs to be sent to function. This is done by sending the event argument and the id (which needs to be bound to the reference).

↓

can be done by using anonymous fn. It will receive event as argument and that can be straight sent to the fn. along with other parameters.

*) Styling React Components → as mentioned before, two ways, either CSS or inline styles.

↓

Setting styles dynamically: the style object passed for inline styling can be modified, and

then that allows for conditional rendering of the element

↓

setting classNames dynamically

```css
.red {
  color: red;
}

.bold {
  font-weight: bold;
}
```

passing a string (which can be dynamically manipulated) to the className prop.

```
const classes = [];
if(this.state.persons.length<=2)classes.push('red');
if(this.state.persons.length<=1)classes.push('bold');

return (
  <div className="App">
    <h1> Hi, I'm a React App</h1>
    <p className = {classes.join(' ')}> This is really working!</p>
    <button
    onClick={this.togglePersonHandler}
    style={style}>
    Toggle
    </button>
    {this.showPersons(style)}

  </div>
);
```

•) using pseudo-selectors in inline styling → Radium

↓

pseudo-selectors and media queries in inline styling

importing Radium into the file we want to use the above mentioned functionality.

*) higher order component → wrapping component over component injecting functionality

•) adding pseudoselectors using Radium :

```
":hover": {
    backgroundColor: "lightgreen",
    Color: "black"
}
```

the pseudo selector inside quotes, and accordingly defined.

↳ same to be followed for all pseudoselectors.

•) media queries : Different style rules based on the capability of the viewing device.

@ media mediatype and | not | only (expressions)

```
{
    css-code;
}
```

```
const style = {
    '@media (min-width: 500px)': {
        width: '450px'
    }
```

To use media queries need to wrap the entire application in a special component. called StyleRoot

↓

in the main render method.

```
return (
  <StyleRoot>
  <div className="App">
    <h1> Hi, I'm a React App</h1>
    <p className = {classes.join(' ')}> This is really working!</p>
    <button
    onClick={this.togglePersonHandler}
    style={style}>
    Toggle
    </button>
    {this.showPersons(style)}

  </div>
  </StyleRoot>
);
```

↓

another 3rd party pkg that can be used is

styled components.

(get rid of all Radium code)

*) important to look at `` method of passing arguments to methods. To add styles using styled-components, we need to import the styled object from styled-components, and use that to create styled components.

↓

This is done using → styled.div``

can also include media queries and pseudoselectors.

← textual argument (css code w/o selectors)

working with pseudo selectors

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

↓

to be applied for the Button on the main component

-) js logic can be inserted in text literal using

${} → should return string here

*) CSS - modules