REACTJS GUIDE

**SECTION 1: GETTING STARTED**

**1. Welcome to the course:**

[**https://subwaymatch.medium.com/disabling-back-button-in-react-with-react-router-v5-34bb316c99d7**](https://subwaymatch.medium.com/disabling-back-button-in-react-with-react-router-v5-34bb316c99d7) **- disabling back button**

**2. What is Reactjs?**

JS library for building user interfaces.

No need to wait for new html response on a request.

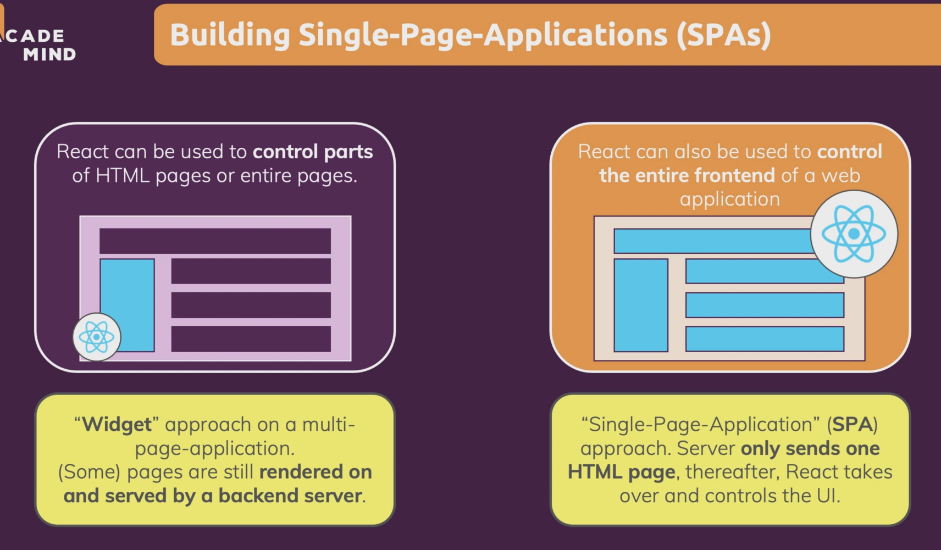
All about building modern, reactive user interfaces for the web.

**3. Why react instead of “just javascript”:**

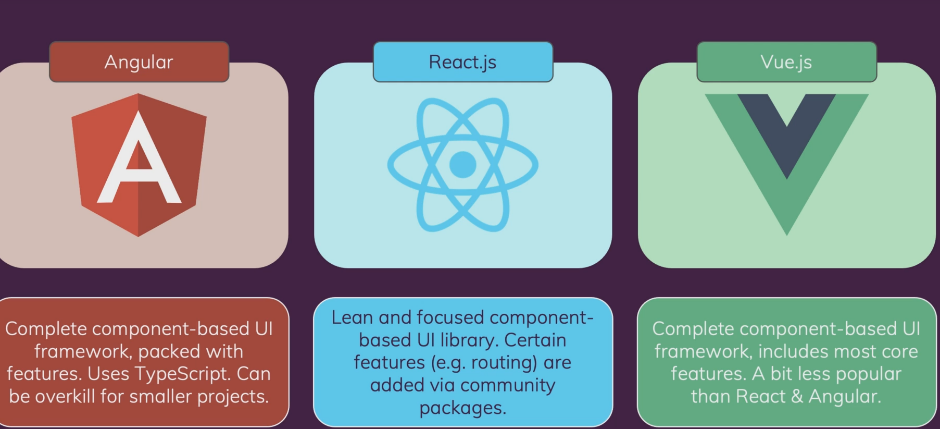
We should repeat many things again and again in html and code complexity will increase even in js.

In react its like we create components which can used as a custom html tags. And this will be easy to maintain and manage. We can reuse many things which will reduce lot codes.

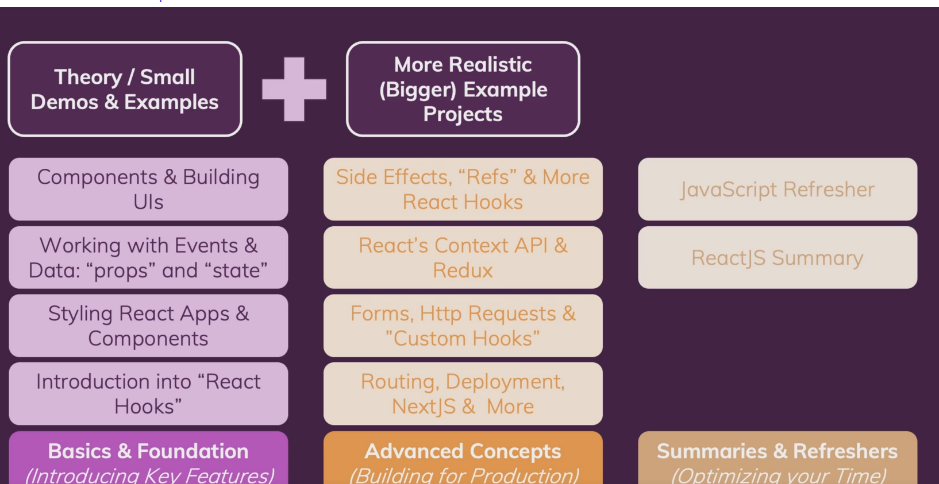
**4. Building single-page applications(SPAs) with text:**



**5. Exploring the reactjs alternatives(Angular/Vue):**



**6. About this course & course offline:**



**SECTION 2: JAVASCRIPT REFRESHER:**

**SECTION 3: React basics & working with components:**

**1. Module introduction:**

Component-driven user interfaces.

**2. What are components? And why is react all about them?**

React makes building complex, interactive and reactive user interfaces simpler.

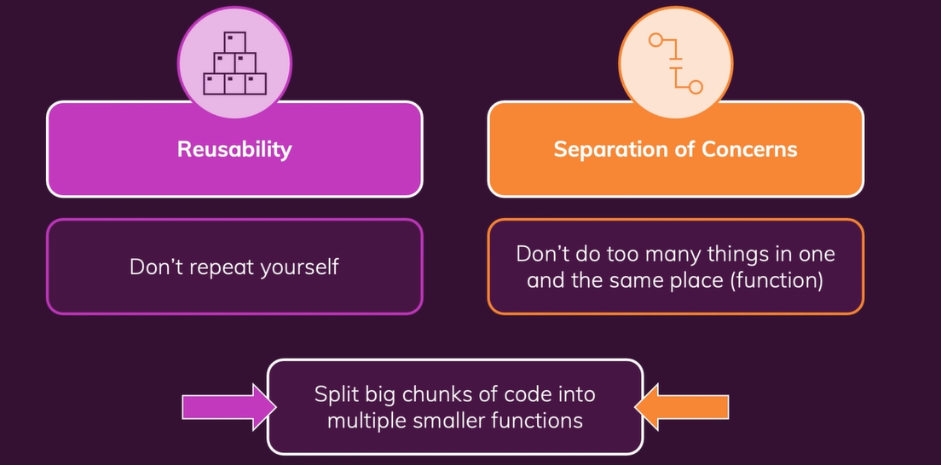
React is all about “components”.

All user interfaces in the end are made up of components.

Reusable building blocks in user interface.

Components are mixed with html, css and javascript.

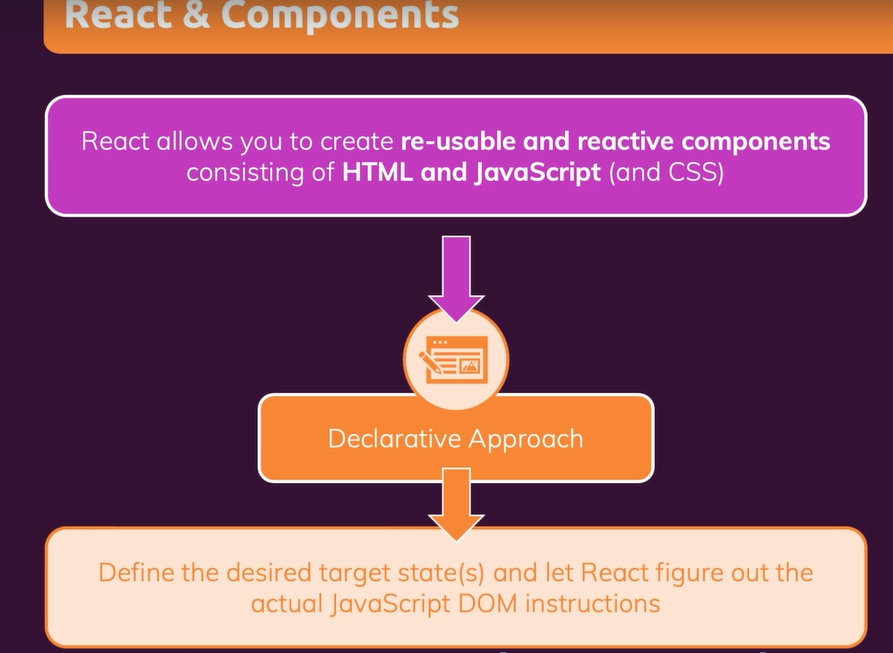
**Why components?**



**3. React code is written in a “declarative way”:**

We built a component combining all html, css(less imp) and javascript.

Build your own html elements.



**4. Creating a new react project:**

Create-react-app is the tool to create react apps.

Package.json will show all the dependencies.

If we are using already created project from external sources,

We should use **npm install**, to install the project details from package.json file and **npm start** can start your project.

**5. Analyzing standard react project:**

We cant import css inside javascript but in reactjs we can do.

Index.js file is the first file to execute.

We need to use export to use the current file into other file using import.

import ReactDOM from 'react-dom';

import './index.css';

import App from './App';

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

In public directory, we can see index.html file which is the only one html file in the whole project.

**6. Introducing JSX:**

JSX – javascript xml

This javascript and jsx code is transformed to browser friendly code.

Go to developer tools -> sources, there we can see static.js folder which consists of some javascript codes. These are the transformed code of the code which we create, this totally convert into basic html and javascript. You can search for App function and check it.

**7. How react works?**

Build your own, custom html elements. We do this by declarative approach(html+javascript).

// in vannila js

  const para = document.createElement('p');

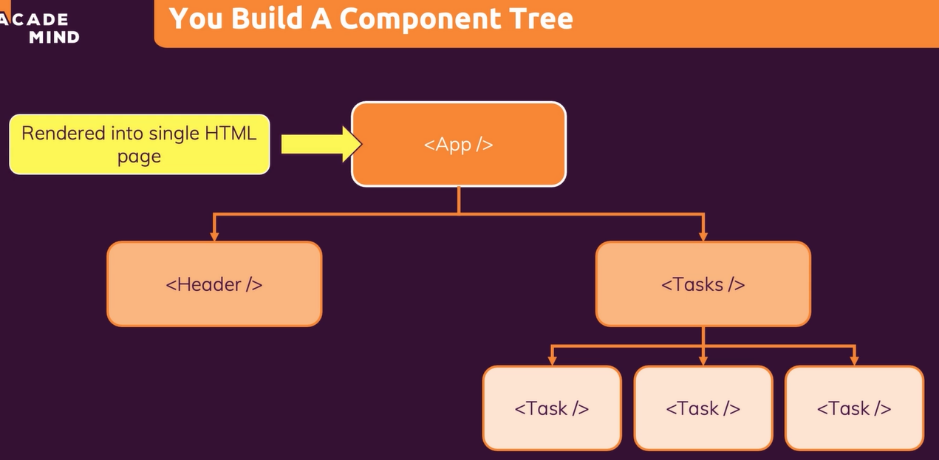
  para.textContent = 'This is also visible!';

  document.getElementById('root').append(para);

this is imperative and it is cumbersome in react, it will appear, disapper.

**8. Building our first custom component:**

One file per component.



function ExpenseItem() {

    return <h2>Expense Item!</h2>

}

export default ExpenseItem;

import ExpenseItem from "./components/ExpenseItem";

function App() {

  // in vannila js

  // const para = document.createElement('p');

  // para.textContent = 'This is also visible!';

  // document.getElementById('root').append(para);

  return (

    <div>

      <h2>Let's get started!</h2>

      <p>This is also visible!</p>

      <ExpenseItem />

    </div>

  );

}

export default App;

<div> - name starts with small letters is html tags.

<ExpenseItem /> - and this is custom html kind of tags which is jsx.

**9. Writing more complex jsx code:**

function ExpenseItem() {

  return (

    <div>

      <div>March 28 2021</div>

      <div>

        <h2>Car Insurance</h2>

        <div>$294.67</div>

      </div>

    </div>

  );

}

export default ExpenseItem;

In return statement, if one html element(start tag + content + end tag) is there then no need to put those brackets else if more html elements like above, use those brackets with high-level div as only one covering elements.

**10. Adding basic CSS styling:**

We can use className to use the class name. And we should import css file to use those styles.

import "./ExpenseItem.css";

function ExpenseItem() {

return (

<div className="expense-item">

<div>March 28 2021</div>

<div className="expense-item\_\_description">

<h2>Car Insurance</h2>

<div className="expense-item\_\_price">$294.67</div>

</div>

</div>

);

}

export default ExpenseItem;

**11. Outputting dynamic data & working with expressions:**

No need to hardcode the data like below, instead we can receive the data from the user and output it dynamically.

function ExpenseItem() {

return (

<div className="expense-item">

<div>March 28 2021</div>

<div className="expense-item\_\_description">

<h2>Car Insurance</h2>

<div className="expense-item\_\_price">$294.67</div>

</div>

</div>

);

}

We can create dynamic data like the code below:

function ExpenseItem() {

const expenseDate = new Date(2021, 2, 28);

const expenseTitle = "Car Insurance";

const expenseAmount = 294.67;

return (

<div className="expense-item">

<div>{expenseDate.toDateString()}</div>

<div className="expense-item\_\_description">

<h2>{expenseTitle}</h2>

<div className="expense-item\_\_price">${expenseAmount}</div>

</div>

</div>

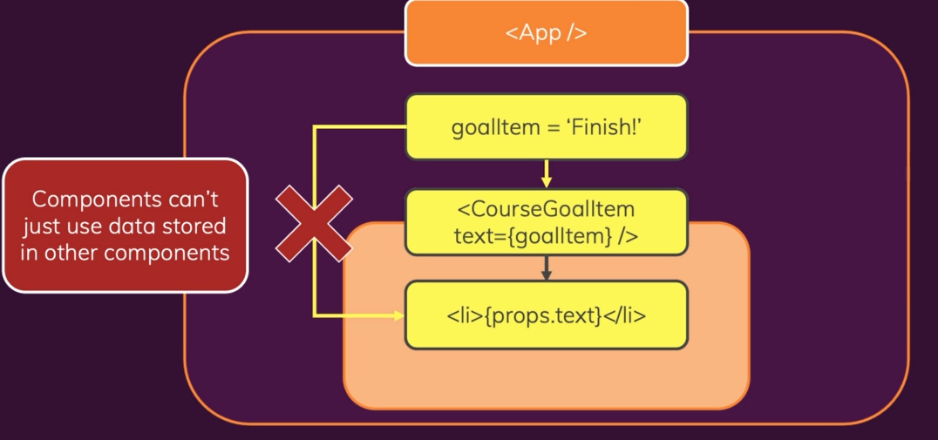
);

}

Here we can use curly brace to use the normal js variable data, we can also do some small js maths work.

Here date is a object, so we cant use that directly to show it as text. So we can change that to string and we can use it.

**12. Passing data via “props”:**



function App() {

  const expenses = [

    {

      id: "e1",

      title: "Toilet Paper",

      amount: 94.12,

      date: new Date(2020, 7, 14),

    },

    { id: "e2", title: "New TV", amount: 799.49, date: new Date(2021, 2, 12) },{

      id: "e3",

      title: "Car Insurance",

      amount: 294.67,

      date: new Date(2021, 2, 28),

    }, {

      id: "e4",

      title: "New Desk (Wooden)",

      amount: 450,

      date: new Date(2021, 5, 12),

    },

  ];

  return (

    <div>

      <h2>Let's get started!</h2>

      <p>This is also visible!</p>

      <ExpenseItem

        title={expenses[0].title}

        amount={expenses[0].amount}

        date={expenses[0].date}

      />

      <ExpenseItem

        title={expenses[1].title}

        amount={expenses[1].amount}

        date={expenses[1].date}

      />

      <ExpenseItem

        title={expenses[2].title}

        amount={expenses[2].amount}

        date={expenses[2].date}

      />

      <ExpenseItem

        title={expenses[3].title}

        amount={expenses[3].amount}

        date={expenses[3].date}

      />

    </div>

  );

}

ExpenseItem.js

function ExpenseItem(props) {

  return (

    <div className="expense-item">

      <div>{props.date.toDateString()}</div>

      <div className="expense-item\_\_description">

        <h2>{props.title}</h2>

        <div className="expense-item\_\_price">${props.amount}</div>

      </div>

    </div>

  );

}

Using props to get data from other components and using that view in this component.

We will pass attributes in the component of custom HTML, only that names should be used in the props to display it.

**13. Adding normal javascript logic to components:**

function ExpenseItem(props) {

  const month = props.date.toLocaleString("en-US", { month: "long" });

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

  const year = props.date.getFullYear();

  return (

    <div className="expense-item">

      <div>

        <div>{month}</div>

        <div>{year}</div>

        <div>{day}</div>

      </div>

      <div className="expense-item\_\_description">

        <h2>{props.title}</h2>

        <div className="expense-item\_\_price">${props.amount}</div>

      </div>

    </div>

  );

}

Here we are using the date functions and using them into the jsx.

**14. Splitting components into multiple components:**

**ExpenseDate.js**

import './ExpenseDate.css';

function ExpenseDate(props) {

  const month = props.date.toLocaleString("en-US", { month: "long" });

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

  const year = props.date.getFullYear();

  return (

    <div className="expense-date">

      <div className="expense-date\_\_month">{month}</div>

      <div className="expense-date\_\_year">{year}</div>

      <div className="expense-date\_\_day">{day}</div>

    </div>

  );

}

export default ExpenseDate;

function ExpenseItem(props) {

  return (

    <div className="expense-item">

      <ExpenseDate date={props.date} />

      <div className="expense-item\_\_description">

        <h2>{props.title}</h2>

        <div className="expense-item\_\_price">${props.amount}</div>

      </div>

    </div>

  );

}

**15. The concept of “Composition” (“children props”):**

Building of user interface from small components to full interface is called composition.

We cant give custom html tags as HOC but we can use it inside the div or HOC(higher order component).

And the tags inside the HOC is composed together with HOC to form the user-interface which is also known as composition.

import './Card.css';

function Card(props) {

    const classes = 'card ' + props.className;

    return <div className={classes}>{props.children}</div>

}

export default Card;

props.children is very special which helps in composition of other components into this component.

It takes all the content between the start and end tags.

function ExpenseItem(props) {

  return (

    <Card className="expense-item">

      <ExpenseDate date={props.date} />

      <div className="expense-item\_\_description">

        <h2>{props.title}</h2>

        <div className="expense-item\_\_price">${props.amount}</div>

      </div>

    </Card>

  );

}

Here Card is used as a wrapper and so the content inside is taken into card component as children and we can access this content using props.children.

**16. Closer look at JSX:**

We will import react whereever jsx is used in before versions.

In modern react this isn’t necessary.

// jsx can be created using like this

return React.createElement(

    "div",

    {},

    React.createElement("h2", {}, "Let's get started!"),

    React.createElement(Expenses, { items: expenses })

  );

------------------------------------------------------------------------------

// normal jsx

  // return (

  //   <div>

  //     <h2>Let's get started!</h2>

  //     <Expenses items={expenses} />

  //   </div>

  // );

**17. An alternative function syntax:**

const Card = (props) => {

  const classes = "card " + props.className;

  return <div className={classes}>{props.children}</div>;

}

Changing a normal function to arrow function.

**SECTION 4: REACT STATE & WORKING WITH EVENTS:**

**1. Module Intro:**

We gonna see user interaction & state; making apps interactive and reactive.

Then we will see handling events, updating the UI & working with “state” and a closer look at components & state.

**2. Listening to events and working with event handlers:**

const clickHandler = () => {

    console.log('clicked!');

  }

<button onClick={clickHandler}>Change Title</button>

We should only point to the function clickHandler and should not give clickHandler(). If we add like this, then immediately the function will run before we click it so only we just only pass a pointer.

**3. How component functions are executed:**

const ExpenseItem = (props) => {

  let title = props.title;

  const clickHandler = () => {

    title = "Updated!";

  }

  return (

    <Card className="expense-item">

      <ExpenseDate date={props.date} />

      <div className="expense-item\_\_description">

        <h2>{title}</h2>

        <div className="expense-item\_\_price">${props.amount}</div>

      </div>

      <button onClick={clickHandler}>Change Title</button>

    </Card>

  );

}

When we try like this to update the title whenever clicked, this wont change the title, because it wont re-render the screen, so to solve this issue, we use the feature called state.

State has a property like, whenever the state changes, the component is re-rendered.

**4. Working with state:**

State is key concept in react.

useState() kind of hooks cannot be outside of the function and it will not be inside any other function as nested function.

They must be called directly inside the function.

const ExpenseItem = (props) => {

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

  const clickHandler = () => {

    setTitle('Updated!');

    console.log(title);

  }

  return (

    <Card className="expense-item">

      <ExpenseDate date={props.date} />

      <div className="expense-item\_\_description">

        <h2>{title}</h2>

        <div className="expense-item\_\_price">${props.amount}</div>

      </div>

      <button onClick={clickHandler}>Change Title</button>

    </Card>

  );

}

Here we use, useState() hook.

The first marked code is deconstructing, title – will have initialized value and setTitle – is the function to update the title. useState(props.title) – props.title is the initial value.

The second marked code will update the title.

Now if we click the button now, the state will re-render the component and compare with the last code if any changes it updates or else it will not.

**5. A closer look at the “useState” hook:**

Why use const?

When we do assign a new value using setTitle(), we are not assigning using equal sign.

By using useState, we tell react to manage some value for us, we never see that variable see that.

When state changes, the component fully re-renders, so const [title, setTitle] = useState(props.title); is also again re-renders and set to const, so no issues. And useState() will not again re-initialize to the initial value, instead react know it has been updated and a new snapshot is created and the new title is returned from react like a array.

**6. State can be updated in many ways:**

Thus far, we update our state **upon user events** (e.g. upon a click).

That's very common but not required for state updates! **You can update states for whatever reason you may have**.

Later in the course, we'll see Http requests that complete (where we then want to update the state based on the Http response we got back) but you could also be updating state because a timer (set with setTimeout()) expired for example.

**7. Adding form inputs:**

const NewExpense = () => {

  return (

    <div className="new-expense">

      <ExpenseForm />

    </div>

  );

};

const ExpenseForm = () => {

  return (

    <form>

      <div className="new-expense\_\_controls">

        <div className="new-expense\_\_control">

          <label>Title</label>

          <input type="text" />

        </div>

        <div className="new-expense\_\_control">

          <label>Amount</label>

          <input type="number" min="0.01" step="0.01" />

        </div>

        <div className="new-expense\_\_control">

          <label>Date</label>

          <input type="date" min="2019-01-01" max="2022-12-31" />

        </div>

      </div>

      <div className="new-expense\_\_actions">

        <button type="submit">Add Expenses</button>

      </div>

    </form>

  );

};

Creating a separate form component and adding it to new expense component.

**8. Listening to user inputs:**

const ExpenseForm = () => {

const titleChangeHandler = (event) => {

console.log(event.target.value);

};

Here through events we can listen to the user inputs and event.target.value will give the value in the text box(user inputs).

**9. Working with multiple states:**

How to manage more than one state, we can simply create more useState() for that particularly.

When one state is updated, the other states are not affected.

**Individual states:**

const ExpenseForm = () => {

const [enteredTitle, setEnteredTitle] = useState("");

const [enteredAmount, setEnteredAmount] = useState("");

const [enteredDate, setEnteredDate] = useState("");

const titleChangeHandler = (event) => {

setEnteredTitle(event.target.value);

};

const amountChangeHandler = (event) => {

setEnteredAmount(event.target.value);

};

const dateChangeHandler = (event) => {

setEnteredDate(event.target.value);

};

return (

<form>

<div className="new-expense\_\_controls">

<div className="new-expense\_\_control">

<label>Title</label>

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

</div>

<div className="new-expense\_\_control">

<label>Amount</label>

<input

type="number"

min="0.01"

step="0.01"

onChange={amountChangeHandler}

/>

</div>

<div className="new-expense\_\_control">

<label>Date</label>

<input

type="date"

min="2019-01-01"

max="2022-12-31"

onChange={dateChangeHandler}

/>

</div>

</div>

<div className="new-expense\_\_actions">

<button type="submit">Add Expenses</button>

</div>

</form>

);

};

**10. Using one state instead (and whats better):**

Instead of using many states we can also use one state.

const [userInput, setUserInput] = useState({

enteredTitle: "",

enteredAmount: "",

enteredDate: ""

});

const titleChangeHandler = (event) => {

setUserInput({

enteredTitle: event.target.value

});

};

When giving the above function, the setUserInput() we use object to update and if we only update the specific thing only then we need to check whether the other data is not lost.

If we only give the new user input, then we dump the other user inputs.

Bcs when we update the new user input state, react will not merge with the old state, it will simply replace the old state with new state.

If the state which is going to update is an object and has only one value, then the other two data’s will be lost.

**Single state example:**

const ExpenseForm = () => {

const [userInput, setUserInput] = useState({

enteredTitle: "",

enteredAmount: "",

enteredDate: ""

});

const titleChangeHandler = (event) => {

setUserInput({

...userInput, // getting old state data

enteredTitle: event.target.value // updating old state data

});

};

const amountChangeHandler = (event) => {

setUserInput({

...userInput,

enteredAmount: event.target.value

});

};

const dateChangeHandler = (event) => {

setUserInput({

...userInput,

enteredDate: event.target.value

});

};

return (

<form>

<div className="new-expense\_\_controls">

<div className="new-expense\_\_control">

<label>Title</label>

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

</div>

<div className="new-expense\_\_control">

<label>Amount</label>

<input

type="number"

min="0.01"

step="0.01"

onChange={amountChangeHandler}

/>

</div>

<div className="new-expense\_\_control">

<label>Date</label>

<input

type="date"

min="2019-01-01"

max="2022-12-31"

onChange={dateChangeHandler}

**Notes:**

Both single and multiple/individual state approaches are fine but suggested approach is individual.

**11. Updating state that depends on the previous state:**

setUserInput({

...userInput,

enteredTitle: event.target.value

});

Updating state like this is not entirely correct. It would technically work but in niche cases it might not work like sometime not updated data will be there and its not a good practice.

If we depend on the previous state, then do like the below:

setUserInput((prevState) => {

return {

...prevState,

enteredTitle: event.target.value

};

});

Use state like this is a good practice and it has a anonymous function which has a prevState parameter, which contains state data of the previous state.

**12. Handling form submission:**

When form is submitted it immediately reloads and sends data to server. But here we no need this, we need only a js function to combine all the form data into one object and add it to the expense items.

event.preventDefault();

this will make the function to not send the request so no reload will happen.

const submitHanlder = (event) => {

event.preventDefault(); *// this will prevent from reloading*

const expenseData = {

title: enteredTitle,

amount: enteredAmount,

date: new Date(enteredDate) *// this will convert the string to date object*

};

console.log(expenseData);

};

return (

<form onSubmit={submitHanlder}>

<div className="new-expense\_\_controls">

<div className="new-expense\_\_control">

<label>Title</label>

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

</div>

<div className="new-expense\_\_control">

<label>Amount</label>

<input

type="number"

min="0.01"

step="0.01"

onChange={amountChangeHandler}

/>

</div>

<div className="new-expense\_\_control">

<label>Date</label>

<input

type="date"

min="2019-01-01"

max="2022-12-31"

onChange={dateChangeHandler}

/>

</div>

</div>

<div className="new-expense\_\_actions">

<button type="submit">Add Expenses</button>

</div>

</form>

);

**13. Adding two-way binding:**

const submitHanlder = (event) => {

event.preventDefault(); *// this will prevent from reloading*

const expenseData = {

title: enteredTitle,

amount: enteredAmount,

date: new Date(enteredDate) *// this will convert the string to date object*

};

console.log(expenseData);

setEnteredTitle(""); // setting back to empty

setEnteredAmount("");

setEnteredDate("");

};

After submitting everything, setting back to default that is empty string.

<form onSubmit={submitHanlder}>

<div className="new-expense\_\_controls">

<div className="new-expense\_\_control">

<label>Title</label>

<input

type="text"

value={enteredTitle}

onChange={titleChangeHandler}

/>

</div>

In input we should add value and set it to the state data, so the value again will be set to empty this will change and show it empty in the inputs.

**14. Child-to-Parent component communication (bottom-up):**

From the child component, the data will be sent to parent component.

Parent to child -> props

Child to parent -> create func in parent and pass it as prop

-> then use this prop in child component and call it

**Example:**

**App.js**

const addExpenseHandler = (expense) => {

console.log("In app.js");

console.log(expense);

};

return (

<div>

<NewExpense onAddExpense={addExpenseHandler} />

<Expenses items={expenses} />

</div>

);

In parent component App.js, created a function with parameter to add that to the array but now just logging it in the console. And adding that function to NewExpense component as a reference, so that we can call this method from child component and pass the data from child to parent.

**NewExpense.js**

const saveExpenseDataHandler = (enteredExpenseData) => {

const expenseData = {

...enteredExpenseData,

id: Math.random().toString()

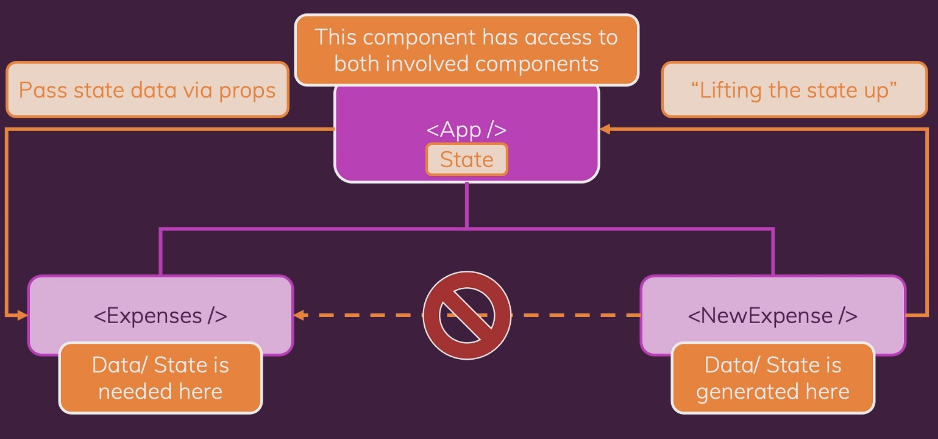
};

props.onAddExpense(expenseData);

};

In child component, we can see the props that calling the parent function and passing the data as argument, this data will be passed to the parent function as parameter and we achieved passing the data from child to parent. So now we can do anything with the data that we shared.

**15. Lifting the state up:**



Here the data is really needed in expenses component to display the expense items.

But the data generated in new expense cannot be passed from newExpense to expense component bcs it has no connection between them.

But the App component has access to both the components, so the solution will be first the data from new expense component can be passed to the App component, then that data can be easily passed via props to the expenses component, this is the idea.

**16. Controlled vs Uncontrolled components & stateless vs stateful components:**

Whenever we use two-way binding, we control that component.

Here we pass on the value input from child to parent and again it comes from parent to child.

These are not really part of this component(ExpensesFilter).

ExpensesFilter is just a component that presents the UI, drop down and couple of listeners and real logic is in the parent component.

And because of this, expense filter is called as controlled component.

So Expenses component controls the ExpensesFilter component.

**Stateless vs stateful or dumb vs intelligent or presentation or non-presentational components:**

Stateful – is where we manage the state

Stateless – is we do not manage the state

Mostly we manage state in some or couple of components and pass the state values to other components to make use of that.

**SECTION 5: RENDERING LISTS & CONDITIONAL CONTENT:**

**1. Module Introduction:**

We can see working with really dynamic content, rendering lists and conditional content.

**2. Rendering lists of data:**

<Card className="expenses">

<ExpensesFilter

selected={filteredYear}

onChangeFilter={filterChangeHandler}

/>

{props.items.map((expense, i) => {

return (

<ExpenseItem

key={i}

title={expense.title}

amount={expense.amount}

date={expense.date}

/>

);

})}

</Card>

here we can see we use map to render the list of array items. Key is used to give a unique kind of id to each data of that list to render properly without any error.

**3. Using stateful lists:**

In app.js, we will add the expense data to the expenses array, but it won’t re-render and display the data with the new data’s. So we should change that to state to make it work properly.

const addExpenseHandler = (expense) => {

setExpenses([expense, ...expenses]);

};

Adding state for an array, this would be not exactly correct.

const DUMMY\_EXPENSES = [

{

id: "e1",

title: "Toilet Paper",

amount: 94.12,

date: new Date(2020, 7, 14)

}

const [expenses, setExpenses] = useState(DUMMY\_EXPENSES);

const addExpenseHandler = (expense) => {

setExpenses((prevExpenses) => {

return [expense, ...prevExpenses];

});

};

This will add the data dynamically to the state to re-render the display.

**4. Understanding “keys”:**

{props.items.map((expense, i) => {

return (

<ExpenseItem

key={i}

title={expense.title}

amount={expense.amount}

date={expense.date}

/>

);

If we not give key attribute when using map(), then it will throw an error.

Performance & bug may be there if we do not use key.

For key we can use, index and separate unique id. But separate unique id’s is better than the index.

Its required for react to correctly identify and update the list elements if needed.

**5. Outputting conditional content:**

{filteredExpenses.length === 0 ? (

<p>No Expenses Found.</p>

) : (

filteredExpenses.map((expense) => {

return (

<ExpenseItem

key={expense.id}

title={expense.title}

amount={expense.amount}

date={expense.date}

/>

);

})

)}

Using ternary operator.

{filteredExpenses.length === 0 && <p>No Expenses Found.</p>}

{filteredExpenses.length > 0 &&

filteredExpenses.map((expense) => {

return (

<ExpenseItem

key={expense.id}

title={expense.title}

amount={expense.amount}

date={expense.date}

/>

);

})}

Here we can see, without ternary operator we only use && operator to achieve the same result.

Also we can use the below code and its most suggestable because of the clean jsx code.

let expensesContent = <p>No expenses found.</p>;

if (filteredExpenses.length > 0) {

expensesContent = filteredExpenses.map((expense) => {

return (

<ExpenseItem

key={expense.id}

title={expense.title}

amount={expense.amount}

date={expense.date}

/>

);

});

}

return (

<div>

<Card className="expenses">

<ExpensesFilter

selected={filteredYear}

onChangeFilter={filterChangeHandler}

/>

{expensesContent}

</Card>

</div>

);

};

**6. Adding conditional return statements:**

Here we can see, that we use conditional return statements.

First if length === 0, it will return no expenses found. And down there another return is there to return the list of expenses.

const ExpensesList = (props) => {

*// to print the expense data, if len 0, no expense found.*

if (props.items.length === 0) {

return <h2 className="expenses-list\_\_fallback">Found no expenses.</h2>;

}

return (

<ul className="expenses-list">

{props.items.map((expense) => {

return (

<ExpenseItem

key={expense.id}

title={expense.title}

amount={expense.amount}

date={expense.date}

/>

);

})}

</ul>

);

};

**7. Demo: Adding a Chart:**

const Chart = (props) => {

return (

<div className="chart">

{props.datapoints.map((datapoint) => (

<ChartBar

key={datapoint.label}

value={datapoint.value}

maxValue={null}

label={datapoint.label}

/>

))}

</div>

);

Like we expect some datapoints array of object which has data like value, label etc. And using the chart bar we can visualize it.

**8. Adding dynamic styles:**

const ChartBar = (props) => {

let barFillHeight = "0%";

if (props.maxValue > 0) {

barFillHeight = Math.round((props.value / props.maxValue) \* 100) + "%";

}

return (

<div className="chart-bar">

<div className="chart-bar\_\_inner">

<div

className="chart-bar\_\_fill"

style={{ height: barFillHeight }}

></div>

</div>

<div className="chart-bar\_\_label">{props.label}</div>

</div>

);

};

Here we dynamically change the styles by dynamically giving the bar height based on values.

**SECTION 6: STYLING REACT COMPONENTS:**

**1. Module Introduction:**

Conditional & dynamic styles

Styled components

CSS modules

**2. Setting dynamic inline styles:**

Till now the stylesheets we are using is global and not scoped.

So we will see how to scope the stylesheets to particular components only.

const CourseInput = (props) => {

const [enteredValue, setEnteredValue] = useState("");

const [isValid, setIsValid] = useState(true);

const goalInputChangeHandler = (event) => {

if (event.target.value.trim().length > 0) {

setIsValid(true);

} // length greater than 0, then ok

setEnteredValue(event.target.value);

};

const formSubmitHandler = (event) => {

event.preventDefault();

if (enteredValue.trim().length === 0) {

setIsValid(false);

return;

} // length is 0, then its wrong

props.onAddGoal(enteredValue);

};

return (

<form onSubmit={formSubmitHandler}>

<div className="form-control">

<label style={{ color: !isValid ? "red" : "black" }}>Course Goal</label>

<input

style={{

borderColor: !isValid ? "red" : "#ccc",

background: !isValid ? "salmon" : "transparent"

}}

type="text"

onChange={goalInputChangeHandler}

/>

</div>

<Button type="submit">Add Goal</Button>

</form>

);

};

Here we use inline styling with conditional rendering.

**3. Setting CSS classes dynamically:**

<form onSubmit={formSubmitHandler}>

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

<label>Course Goal</label>

<input type="text" onChange={goalInputChangeHandler} />

</div>

<Button type="submit">Add Goal</Button>

</form>

Using backticks, we can add dynamic classes or to an array we can add the values and can join it through space and can put that into the className to give it the style.

**4. Introducing styled components:**

<https://styled-components.com/>

npm install --save styled-components

import styled from 'styled-components';

const Button = styled.button``;

we will take the class names and instead of pseudo-selectors we will use “&”.

No need to put react, bcs we will not use react here.

const Button = styled.button`

font: inherit;

padding: 0.5rem 1.5rem;

border: 1px solid #8b005d;

color: white;

background: #8b005d;

box-shadow: 0 0 4px rgba(0, 0, 0, 0.26);

cursor: pointer;

&:focus {

outline: none;

}

&:hover,

&:active {

background: #ac0e77;

border-color: #ac0e77;

box-shadow: 0 0 8px rgba(0, 0, 0, 0.26);

}

`;

**5. Styled components & dynamic props:**

We can create more than 1 component in a file but keeping it to 1 component is good.

.form-control label {

font-weight: bold;

display: block;

margin-bottom: 0.5rem;

}

// the above code should be replaced like the below.

& label {

font-weight: bold;

display: block;

margin-bottom: 0.5rem;

}

Second component in same file,

const FormControl = styled.div`

margin: 0.5rem 0;

& label {

font-weight: bold;

display: block;

margin-bottom: 0.5rem;

}

& input {

display: block;

width: 100%;

border: 1px solid #ccc;

font: inherit;

line-height: 1.5rem;

padding: 0 0.25rem;

}

& input:focus {

outline: none;

background: #fad0ec;

border-color: #8b005d;

}

&.invalid input {

border-color: red;

background: #ffd7d7;

}

&.invalid label {

color: red;

}

`;

<FormControl className={!isValid && 'invalid'}>

<label>Course Goal</label>

<input type="text" onChange={goalInputChangeHandler} />

</FormControl>

We can add that above one for adding invalid class.

const FormControl = styled.div`

margin: 0.5rem 0;

& label {

font-weight: bold;

display: block;

margin-bottom: 0.5rem;

color: ${(props) => (props.invalid ? "red" : "black")};

}

& input {

display: block;

width: 100%;

border: 1px solid ${(props) => (props.invalid ? "red" : "#ccc")};

background: ${(props) => (props.invalid ? "#ffd7d7" : "transparent")};

font: inherit;

line-height: 1.5rem;

padding: 0 0.25rem;

}

& input:focus {

outline: none;

background: #fad0ec;

border-color: #8b005d;

}

`;

<FormControl invalid={!isValid}>

<label>Course Goal</label>

<input type="text" onChange={goalInputChangeHandler} />

</FormControl>

Compare this with the last page, we can find difference here we use props true or false, to work out.

**6. Styled components & media queries:**

const Button = styled.button`

width: 100%;

font: inherit;

padding: 0.5rem 1.5rem;

border: 1px solid #8b005d;

color: white;

background: #8b005d;

box-shadow: 0 0 4px rgba(0, 0, 0, 0.26);

cursor: pointer;

@media (min-width: 768px) {

width: auto;

}

&:focus {

outline: none;

}

&:hover,

&:active {

background: #ac0e77;

border-color: #ac0e77;

box-shadow: 0 0 8px rgba(0, 0, 0, 0.26);

}

`;

Media query is used as normal as in css.

**7. Using CSS Modules:**

Css in separate files will be always nice and good.

We can make the file name as Course.module.css to use the module scopes. It is scoped now.

import styles from "./Button.module.css";

<button type={props.type} className={styles.button} xonClick={props.onClick}>

{props.children}

</button>

**8. Dynamic styles with CSS modules:**

<div className={styles['form-control']}>

If different css classes, has dashes in it, we can use it like the above. Else we can use dot notation.

<form onSubmit={formSubmitHandler}>

<div

className={`${styles["form-control"]} ${!isValid && styles.invalid}`}

>

<label>Course Goal</label>

<input type="text" onChange={goalInputChangeHandler} />

</div>

<Button type="submit">Add Goal</Button>

</form>

We can styles dynamically.

**SECTION 7: DEBUGGING REACT APPS:**

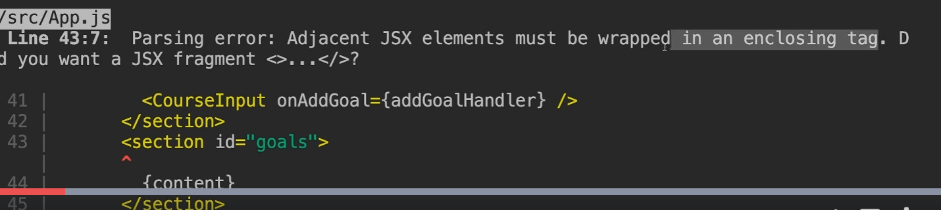
**1. Module introduction:**

Finding & fixing errors.



**2. Understanding React error messages:**

Mostly here no code, just watching and writing the solutions & feedback.



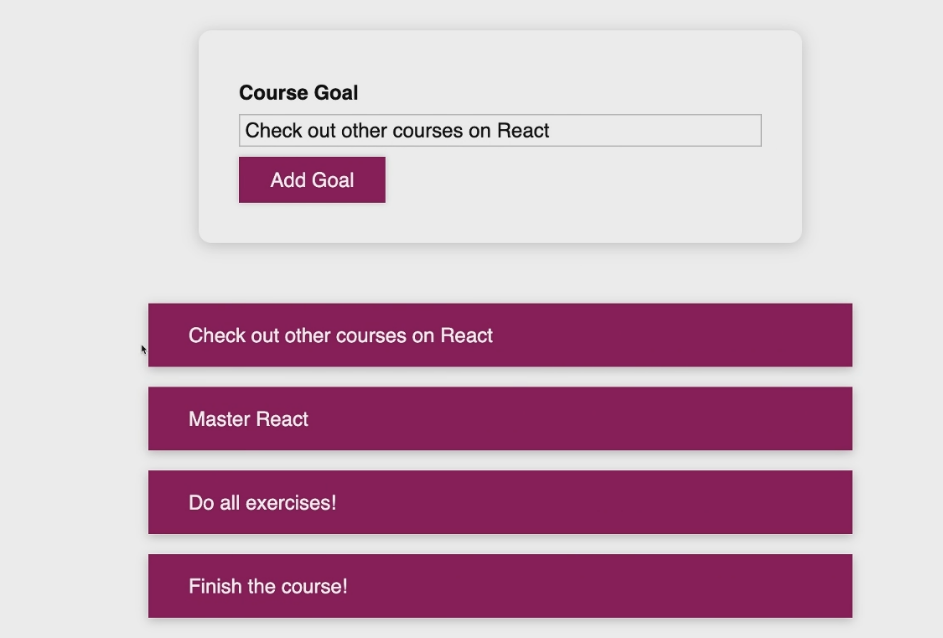
Understanding error msgs will give lot of details to rectify it.

Look at the error msgs and connect the error msg to rectify it.

**3. Analyzing code flow & warnings:**

Sometimes the error won’t lead to compilation error and will not display too.

Foe example,



Here if I click on check out other courses on react, it deleted the master react. But it should be deleted itself. Which causing this error and how we gonna rectify it?

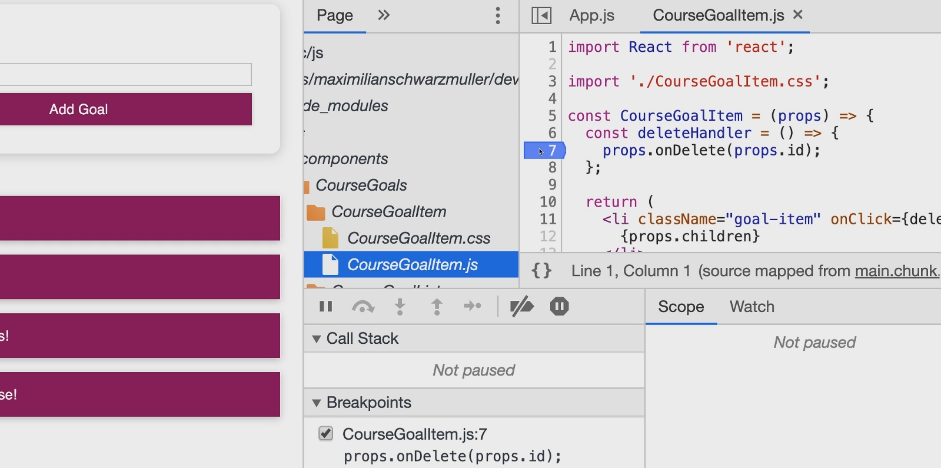
First we should find delete logic or the place where we got error. And we should go there. But here the logic is fine, so may be the id which we give is wrong. Now we should check on that. Here there is a logical error, the id is same when we add, which is hardcoded. We can check through chain of components to get the issues. It analyzing the codes.

**4. Working with breakpoints:**

If we can’t find the errors, we can try using breakpoints.

In developer tools -> click sources -> users/username -> src -> react code.

If not found there, we can find it in the webpack.



If not found there, we can find it in the webpack.

When we start deleting, code execution stops at line 7.

**5. Using the react dev tools:**

Dedicated react developer tool. After installing, we can see components and profiler.

With the help of components, we can which is rendered by which component.

It shows the props, hooks used in that particular component and more.

**SECTION 8: TIME TO PRACTICE: A COMPLETE PRACTICE PROJECT:**

**1. Module Introduction:**

**2. Adding a “User” component:**

const AddUser = (props) => {

const addUserHandler = (event) => {

event.preventDefault();

};

return (

<form onSubmit={addUserHandler}>

<label htmlFor="username">Username</label>

<input id="username" type="text" />

<label htmlFor="age">Age (Years)</label>

<input id="age" type="number" />

<button type="submit">Add User</button>

</form>

);

};

import AddUser from "./Components/Users/AddUser";

function App() {

return (

<div>

<AddUser />

</div>

);

}

**3. Adding a re-usable card component:**

const Card = (props) => {

return (

<div className={`${classes.card} ${props.className}`}>{props.children}</div>

);

};

**4. Adding a reusable “Button” component:**

const Button = (props) => {

return (

<button

className={classes.button}

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

onClick={props.onClick}

>

{props.children}

</button>

);

};

**5. Managing the user input state:**

const AddUser = (props) => {

const [enteredUsername, setEnteredUsername] = useState("");

const [enteredAge, setEnteredAge] = useState("");

const addUserHandler = (event) => {

event.preventDefault();

console.log(enteredUsername, enteredAge);

};

const usernameChangeHandler = (event) => {

setEnteredUsername(event.target.value);

};

const ageChangeHandler = (event) => {

setEnteredAge(event.target.value);

};

return (

<Card className={classes.input}>

<form onSubmit={addUserHandler}>

<label htmlFor="username">Username</label>

<input id="username" type="text" onChange={usernameChangeHandler} />

<label htmlFor="age">Age (Years)</label>

<input id="age" type="number" onChange={ageChangeHandler} />

<Button type="submit">Add User</Button>

</form>

</Card>

);

};

**6. Adding validation & resetting logic:**

const AddUser = (props) => {

const [enteredUsername, setEnteredUsername] = useState("");

const [enteredAge, setEnteredAge] = useState("");

const addUserHandler = (event) => {

event.preventDefault();

if (enteredUsername.trim().length === 0 || enteredAge.trim().length === 0) {

return;

}

if (+enteredAge < 1) { *// making enteredAge as number*

return;

}

console.log(enteredUsername, enteredAge);

setEnteredUsername("");

setEnteredAge("");

};

const usernameChangeHandler = (event) => {

setEnteredUsername(event.target.value);

};

const ageChangeHandler = (event) => {

setEnteredAge(event.target.value);

};

return (

<Card className={classes.input}>

<form onSubmit={addUserHandler}>

<label htmlFor="username">Username</label>

<input

id="username"

type="text"

value={enteredUsername}

onChange={usernameChangeHandler}

/>

<label htmlFor="age">Age (Years)</label>

<input

id="age"

type="number"

value={enteredAge}

onChange={ageChangeHandler}

/>

<Button type="submit">Add User</Button>

</form>

</Card>

);

};

**7. Adding users list component:**

const UsersList = (props) => {

return (

<Card className={classes.users}>

<ul>

{props.users.map((user) => (

<li>

{user.name} ({user.age} years old)

</li>

))}

</ul>

</Card>

);

};

**8. Managing a list of users via state:**

function App() {

const [usersList, setUsersList] = useState([]);

const addUserHandler = (uName, uAge) => {

setUsersList((prevState) => {

return [...prevState, { name: uName, age: uAge }];

});

};

return (

<div>

<AddUser onAddUser={addUserHandler} />

<UsersList users={usersList} />

</div>

);

}

const addUserHandler = (event) => {

event.preventDefault();

if (enteredUsername.trim().length === 0 || enteredAge.trim().length === 0) {

return;

}

if (+enteredAge < 1) {

*// making enteredAge as number*

return;

}

props.onAddUser(enteredUsername, enteredAge);

setEnteredUsername("");

setEnteredAge("");

};

**9. Adding the “errorModal” component:**

const ErrorModal = (props) => {

return (

<div>

<div className={classes.backdrop} />

<Card className={classes.modal}>

<header className={classes.header}>

<h2>{props.title}</h2>

</header>

<div className={classes.content}>

<p>{props.message}</p>

</div>

<footer className={classes.actions}>

<Button>Okay</Button>

</footer>

</Card>

</div>

);

};

**10. Managing the error state:**

const [error, setError] = useState();

return (

<div>

{error && (

<ErrorModal

title={error.title}

message={error.message}

onConfirm={errorHandler}

/>

)}

**SECTION 9: DIVING DEEPER: WORKING WITH FRAGMENTS, PORTALS AND REFS:**

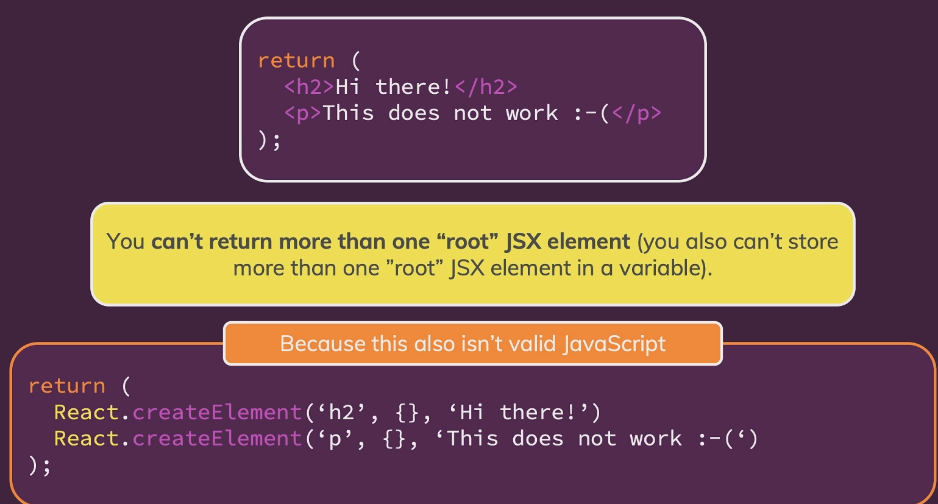
**1. Module Introduction:**

Jsx limitation & fragments

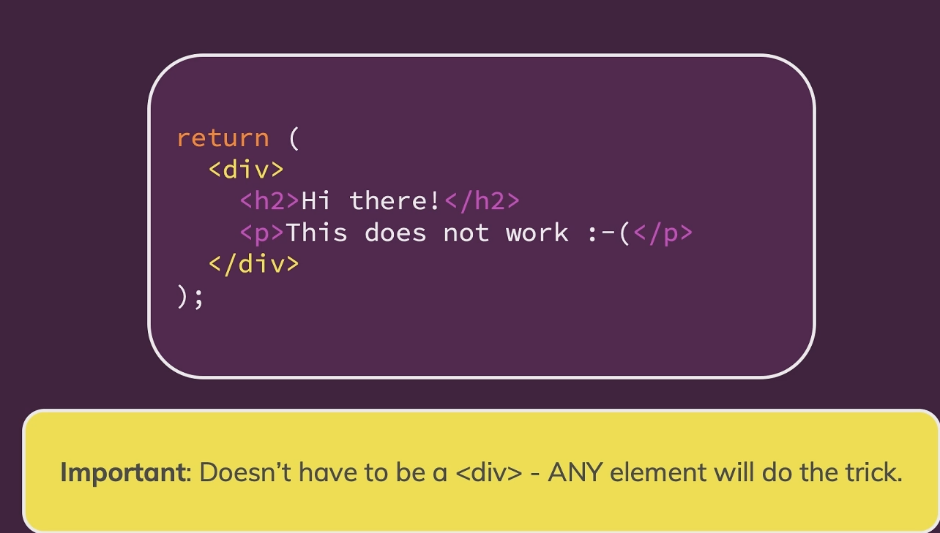
Getting a cleaner DOM with portals

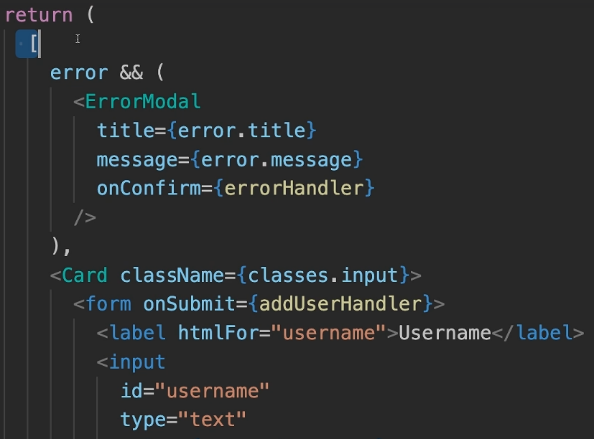
Working with Refs

**2. JSX Limitations & Workarounds:**

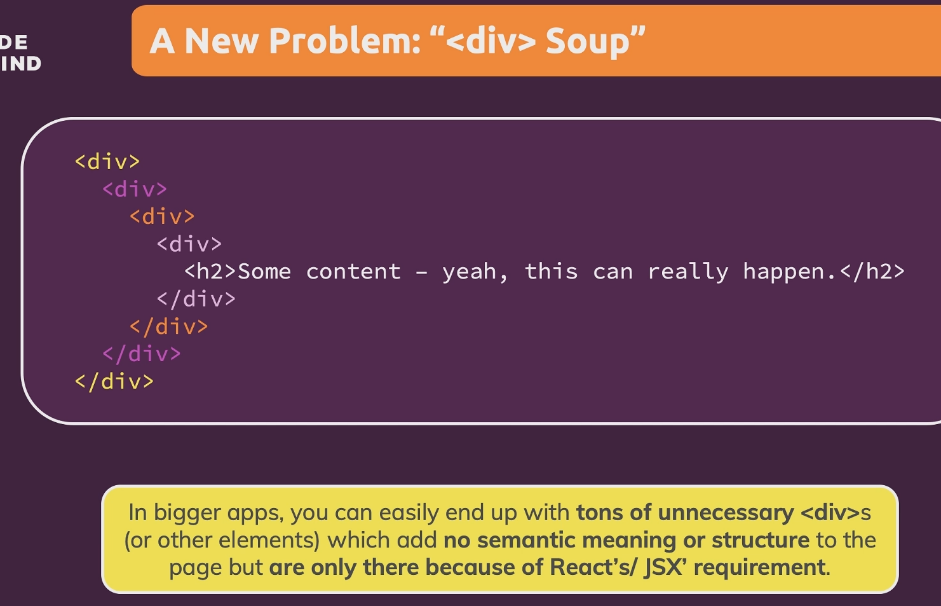


We can also return array but not two array.





We can also use array instead of div with comma-seperated. And should add key or else an error will be thrown.



It may break styling and not a good practice. And this may make the application slower bcs it has render many html things.

**3. Creating a wrapper component:**

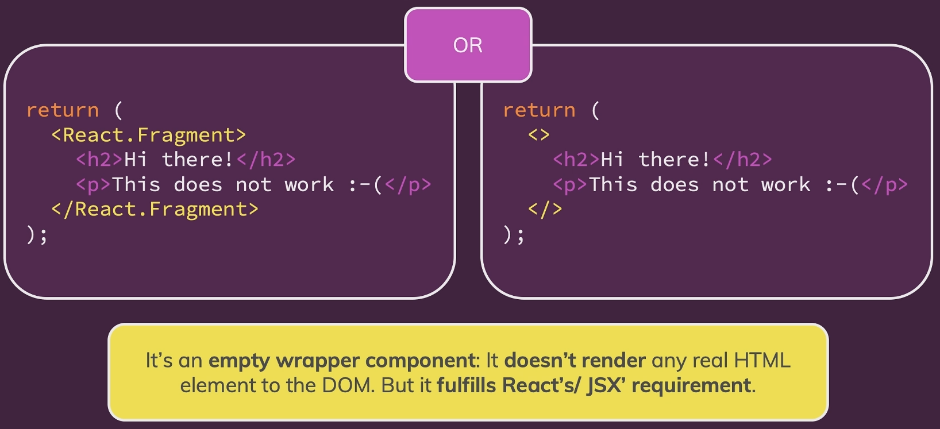
const Wrapper = (props) => {

return props.children;

};

export default Wrapper;

**4. React Fragments:**



return (

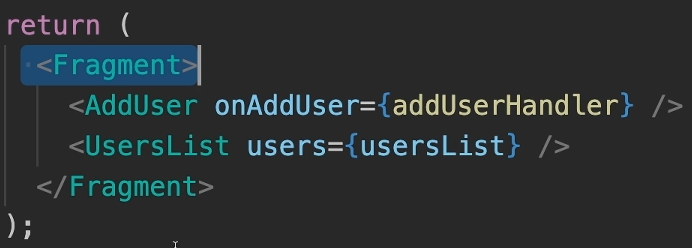
<>

<AddUser onAddUser={addUserHandler} />

<UsersList users={usersList} />

</>

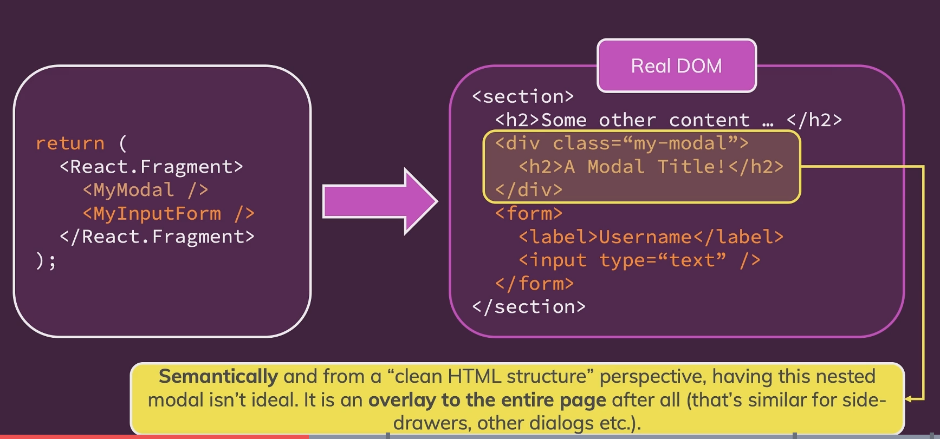
);

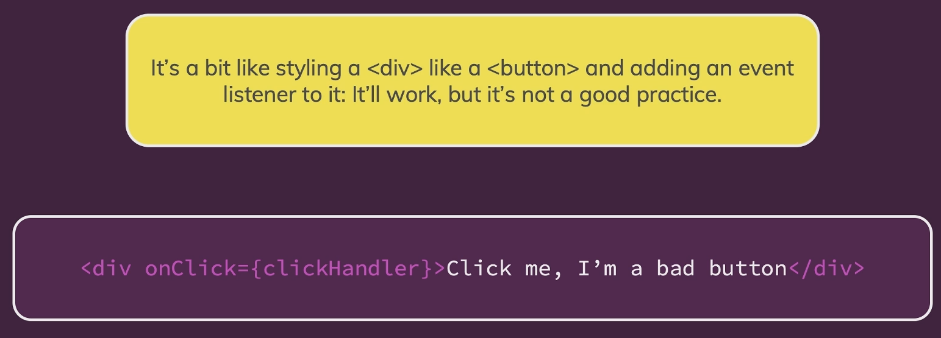


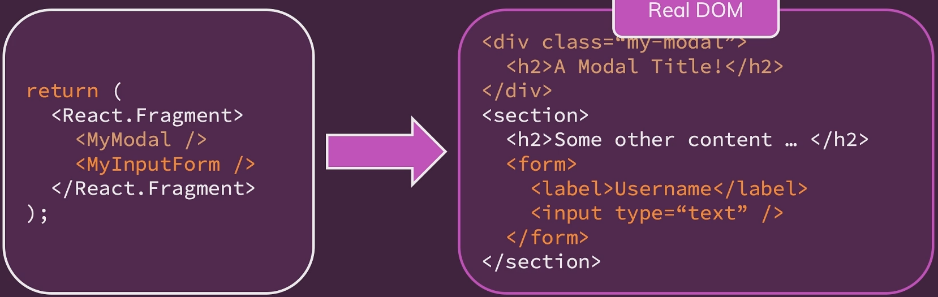
**5. Introducing react portals:**

Fragments allow us to write cleaner code to end up with less unnecessary elements on final page.

React Portals also like fragments. Also help us write cleaner code.







**6. Working with portals:**

const ErrorModal = (props) => {

return (

<React.Fragment>

<div className={classes.backdrop} onClick={props.onConfirm} />

<Card className={classes.modal}>

<header className={classes.header}>

<h2>{props.title}</h2>

</header>

<div className={classes.content}>

<p>{props.message}</p>

</div>

<footer className={classes.actions}>

<Button onClick={props.onConfirm}>Okay</Button>

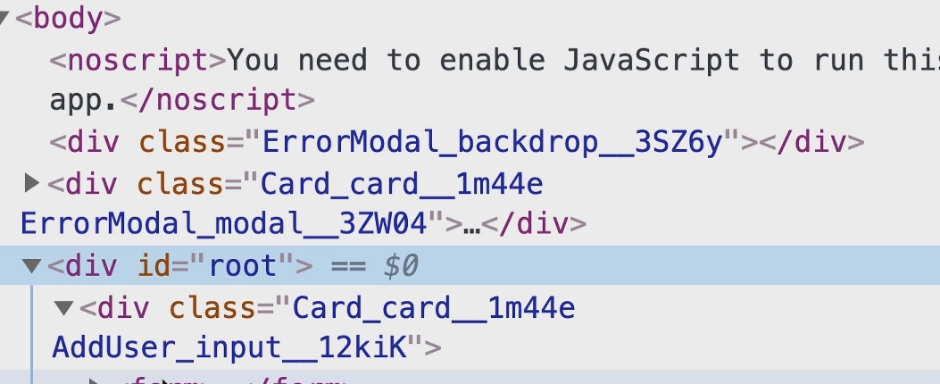
</footer>

</Card>

</React.Fragment>

);

};



The div element, above the root div, we can achieve through react portals.

Overlay-root: like modal, side drawer etc.

Portals will be in react-dom.

const ErrorModal = (props) => {

return (

<React.Fragment>

{ReactDOM.createPortal(

<Backdrop onConfirm={props.onConfirm} />,

document.getElementById("backdrop-root")

)}

</React.Fragment>

);

};

createPortal(JSX component, index.html->div backdrop root)

second parameter is we will create a div outside/above the root div and we should point that to the backdrop-root div.

const ErrorModal = (props) => {

return (

<React.Fragment>

{ReactDOM.createPortal(

<Backdrop onConfirm={props.onConfirm} />,

document.getElementById("backdrop-root")

)}

{ReactDOM.createPortal(

<ModalOverlay

title={props.title}

message={props.message}

onConfirm={props.onConfirm}

/>,

document.getElementById("overlay-root")

)}

</React.Fragment>

);

};

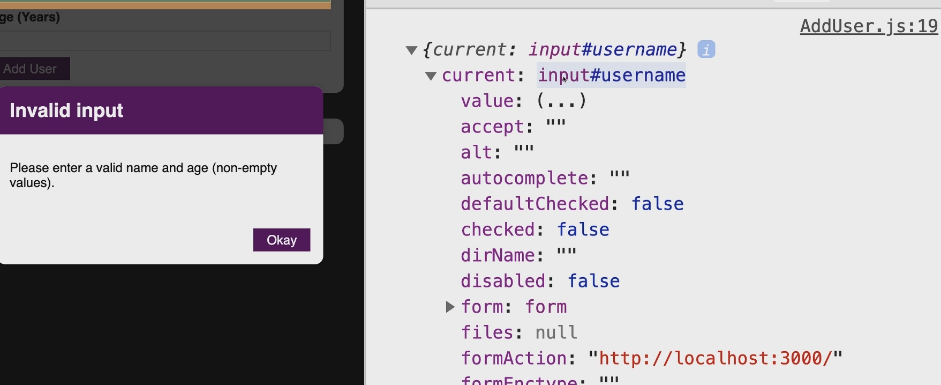
**7. Working with “refs”:**

Ref’s are quite powerful, they allow us to access with other DOM elements to work with them.

In AddUser, the text, state will change to each keystroke. That’s the scenario where refs can help us.

When we set a value in useRef(), when rendering, this initial value will be set to input value.

The current prop holds the actual value that ref is connected with.



So when we print the value, we can the value what we type when we submit.

So there is no need of state and no onChange can be used for all keystrokes, so the rendering will become less and performance will become high.

So need to reset the value to null, to make it empty atlast. And we can clear the state stuffs. And the changeHandlers can be removed.

And can also remove value and onChange property in input tags.

But resetting of input values will be lost it will not reset after submit.

nameInputRef.current.value = "";

ageInputRef.current.value = "";

this is not a good practice, but only to reset we can use but not anywhere else.

Rarely use ref’s to manipulate the DOM. We are not adding new element or css class. Its only changing the user entered input.

We can use ref as well as state.

When if we need only to read the value and not to change, we can use ref there.

Read a value – ref’s are better

Otherwise it will be our own wish to use any of this.

Ref will make code less than the state.

const AddUser = (props) => {

const nameInputRef = useRef();

const ageInputRef = useRef();

const [error, setError] = useState();

const addUserHandler = (event) => {

event.preventDefault();

const enteredName = nameInputRef.current.value;

const enteredUserAge = ageInputRef.current.value;

if (enteredName.trim().length === 0 || enteredUserAge.trim().length === 0) {

setError({

title: "Invalid Input",

message: "please enter a valid name and age (non-empty values)."

});

return;

}

if (+enteredUserAge < 1) {

*// making enteredAge as number*

setError({

title: "Invalid Input",

message: "Please enter a valid age (> 0)."

});

return;

}

props.onAddUser(enteredName, enteredUserAge);

nameInputRef.current.value = "";

ageInputRef.current.value = ""; // to reset the value

};

**8. Controlled vs Uncontrolled components:**

If we access with the ref then it will be uncontrolled component.

We must control the state of the component for controlled component.

**SECTION 10: ADVANCED: HANDLING SIDE EFFECTS, USING REDUCERS & CONTEXT API:**

**1. Module Introduction:**

Effects, Reducers & Context.



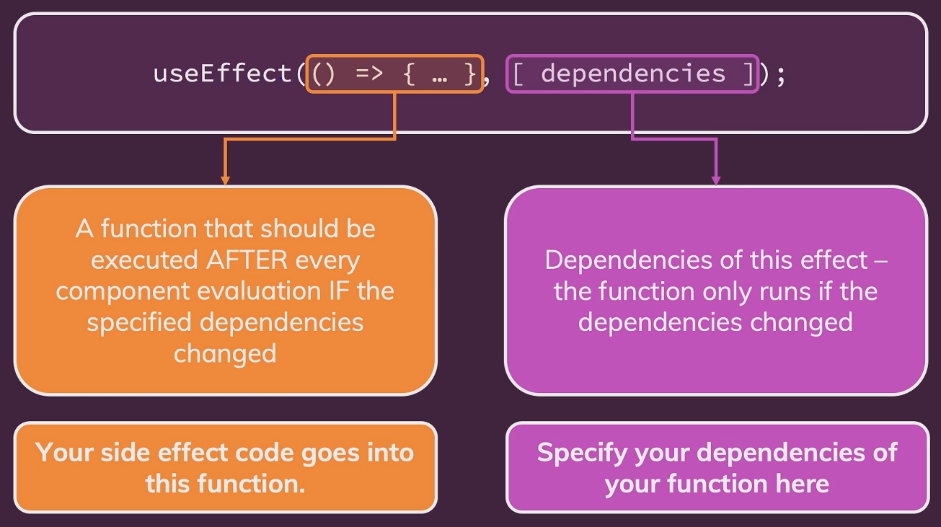
**2. What are “Side Effects” & Introducing useEffect:**

Effect or side effect.

If eventually we change the state, it will lead to infinite loop.

Side effects cannot go directly into this component function bcs that would create bugs, infinite loops or simply send too many http requests.

useEffect() – built in hook



**3. Using the useEffect() hook:**

We can send http request to backend server to check the username & password, we can use useEffect() there.

But in this app, if we login and reload it, the login status will be lost and it will logout, why this happens is, we use state to store login status and if refresh it, the before state will be lost and now this login state will be false again, and y its showing the login page.

So we should persist the data or check whether the data is persisted. And no data will be lost. So to make it happen we use useEffect().

const [isLoggedIn, setIsLoggedIn] = useState(false);

const storedUserLoggedInInformation = localStorage.getItem("isLoggedIn");

if (storedUserLoggedInInformation === "1") {

setIsLoggedIn(true);

}

We can do this to check the state and if already logged in, the state is updated to true.

But major con is, it will remain in infinite loop of many re-renders. So we need useEffect.

useEffect(anonymous func, array of dependencies)

After the app component runs everything, atlast the useEffect function will run. And it will not update all time, only the array of dependencies are changed it will update.

If we reload the app now, it will not go to the login again, it will stay in the same page.

We can use useEffect(), when a func that needs to be run only when its dependencies changes.

**4. useEffect & Dependencies:**

const emailChangeHandler = (event) => {

setEnteredEmail(event.target.value);

setFormIsValid(

event.target.value.includes('@') && enteredPassword.trim().length > 6

);

};

const passwordChangeHandler = (event) => {

setEnteredPassword(event.target.value);

setFormIsValid(

event.target.value.trim().length > 6 && enteredEmail.includes('@')

);

};

Here we can see, we will check form is valid or not, with same lines in two functions. So we can check this at last once, so for that to happen we can use useEffect().

useEffect(() => {

setFormIsValid(

enteredEmail.value.includes("@") && enteredPassword.trim().length > 6

);

}, []);

And if we done like this, then the login button will not be enabled, bcs setFormIsValid only once when the component is rendered for the first time without any dependencies.

If we removed the empty array from useEffect, then it will run into infinite loop.

useEffect(() => {

setFormIsValid(

enteredEmail.value.includes("@") && enteredPassword.trim().length > 6

);

}, [enteredEmail, enteredPassword]);

So now in above, we gave dependencies like enteredEmail & enteredPassword and only it changes from last snapshot, then the function will run.

We use useEffect() and helps us make sure that we have one code in one place, instead of, as before in multiple places which re-runs, whenever one of the dependencies changed.

**Note:**

If only one time a function should run & when the component first re-rendered, then there should be empty array in useEffect.

If we want to run more than one time, but only based on dependencie changes, then we can also use useEffect.

useEffect is a super important hook that helps you deal with code that should be executed in response to something. And something could be the component being loaded. It could be email address being updated. It could be anything, whenever you have an action that should be executed in response to some other action, that is a side effect.

**5. Want to add & Not to add as dependencies:**

In the previous lecture, we explored useEffect() dependencies.

You learned, that you should add "everything" you use in the effect function as a dependency - i.e. all state variables and functions you use in there.

That is correct, but there are a **few exceptions** you should be aware of:

* You **DON'T need to add state updating functions** (as we did in the last lecture with setFormIsValid): React guarantees that those functions never change, hence you don't need to add them as dependencies (you could though)
* You also **DON'T need to add "built-in" APIs or functions** like fetch(), localStorage etc (functions and features built-into the browser and hence available globally): These browser APIs / global functions are not related to the React component render cycle and they also never change
* You also **DON'T need to add variables or functions** you might've **defined OUTSIDE of your components** (e.g. if you create a new helper function in a separate file): Such functions or variables also are not created inside of a component function and hence changing them won't affect your components (components won't be re-evaluated if such variables or functions change and vice-versa)

So long story short: You must add all "things" you use in your effect function **if those "things" could change because your component (or some parent component) re-rendered.** That's why variables or state defined in component functions, props or functions defined in component functions have to be added as dependencies!

Here's a made-up dummy example to further clarify the above-mentioned scenarios:

1. import { useEffect, useState } from 'react';
3. let myTimer;
5. const MyComponent = (props) => {
6. const [timerIsActive, setTimerIsActive] = useState(false);
8. const { timerDuration } = props; // using destructuring to pull out specific props values
10. useEffect(() => {
11. if (!timerIsActive) {
12. setTimerIsActive(true);
13. myTimer = setTimeout(() => {
14. setTimerIsActive(false);
15. }, timerDuration);
16. }
17. }, [timerIsActive, timerDuration]);
18. };

In this example:

* timerIsActive is **added as a dependency** because it's component state that may change when the component changes (e.g. because the state was updated)
* timerDuration is **added as a dependency** because it's a prop value of that component - so it may change if a parent component changes that value (causing this MyComponent component to re-render as well)
* setTimerIsActive is **NOT added as a dependency** because it's that **exception**: State updating functions could be added but don't have to be added since React guarantees that the functions themselves never change
* myTimer is **NOT added as a dependency** because it's **not a component-internal variable** (i.e. not some state or a prop value) - it's defined outside of the component and changing it (no matter where) **wouldn't cause the component to be re-evaluated**
* setTimeout is **NOT added as a dependency** because it's **a built-in API** (built-into the browser) - it's independent from React and your components, it doesn't change

**6. Using the useEffect cleanup function:**

Sometimes we can use this useEffect to do some clean up work.

When we type in the input text field, for each keystroke, the useEffect will re-render bcs of state.

useEffect(() => {

console.log('checking from validity');

setFormIsValid(

enteredEmail.includes("@") && enteredPassword.trim().length > 6

);

}, [enteredEmail, enteredPassword]);

So to avoid useEffect to run for every keystroke, we can make like, when the user passes, that time only we can update the value, so here network traffic is low.

useEffect(() => {

setTimeout(() => {

console.log("checking from validity");

setFormIsValid(

enteredEmail.includes("@") && enteredPassword.trim().length > 6

);

}, 500);

return () => {

console.log("CLEANUP");

}; *// cleanup process*

}, [enteredEmail, enteredPassword]);

The cleanup process for the first time it wont run, but after that it will run to clean up whatever runs in the useEffect.

Like it will clean up the timer thing. So no more than 1 timer will be there.

When running the above code, cleanup will not print for the first time.

useEffect(() => {

const identifier = setTimeout(() => {

console.log("checking from validity");

setFormIsValid(

enteredEmail.includes("@") && enteredPassword.trim().length > 6

);

}, 500);

return () => {

console.log("CLEANUP");

clearTimeout(identifier);

}; *// cleanup process*

}, [enteredEmail, enteredPassword]);

**7. useEffect Summary:**

**i) without parameter:**

useEffect(() => {

console.log("EFFECT RUNNING");

});

When give without parameter, it runs for everything like every keystroke, component mount etc.

**ii) with empty parameter:**

useEffect(() => {

console.log("EFFECT RUNNING");

}, []);

This with empty paramtere, it will run only for the first time when the component render.

And not for any subsequenet re-render cycle.

**iii) with value in parameter:**

useEffect(() => {

console.log("EFFECT RUNNING");

}, [enteredPassword]);

Adding a dependency to a useEffect parameter.

Now the func re-runs whenever the component was re-evaluated and the enteredpassword state(dependency) change this func will run.

So we can for keystroke for state changing in enteredEmail will not run this func. But in enteredpassword it will run.

**iv) with cleanup func and dependency:**

useEffect(() => {

console.log("EFFECT RUNNING");

return () => { // cleanup func

console.log("EFFECT CLEANUP");

};

}, [enteredPassword]);

This cleanup func will run before this useEffect function as a whole runs but not the first time it runs.



First cleanup runs then only the useEffect func will run and displays the Effect running.

**v) cleanup func with no dependency:**

useEffect(() => {

console.log("EFFECT RUNNING");

return () => { // cleanup func

console.log("EFFECT CLEANUP");

};

}, []);

This cleanup func will run only when the component is unmounted. Like componentDidUnmount.

**8. Introducing useReducer & Reducers in general:**

useReducer is another built-in hook and it helps us with state management.

So its bit like useState but with more capabilities. And especially for more complex state.

const emailChangeHandler = (event) => {

setEnteredEmail(event.target.value);

setFormIsValid(

event.target.value.includes("@") && enteredPassword.trim().length > 6

);

};

This should not be done but to understand we use this. Sometime when the first condition works and second state may not be updated before the first condition.

const [enteredEmail, setEnteredEmail] = useState("");

const [emailIsValid, setEmailIsValid] = useState();

When we have states that belong together like enteredValue and validity of the value.

or if u have state updates that depand on other state.

const validateEmailHandler = () => {

setEmailIsValid(enteredEmail.includes("@"));

};

enteredEmail is different state and we are using setEmailIsVaild which is another state.

Updating once state using the other state, we should not do. It may work in most cases, but in some cases it may no work bcs may be some state update for enteredEmail wasn’t processed in time. And then we are trying to update other state that depand on enteredEmail which has some outdated data. So here useReducer is good choice here.

If u update a state which depands on other state, then merging this into one state could be a good idea. And we can also do this without useReducer as well.

In case like state becomes bigger and combines multiple related state we can use useReducer which would be worth a closer look.

**9. Using the useReducer() hook:**

Const [state, dispatchFn] = useReducer(reducerFn, initialState, initFn);

State – the state snapshot used in the component re-render/re-evaluation cycle.

dispatchFn – a func that can be used to dispatch a new action(i.e. trigger an update of the state)

And the action that is dispatched from the above fn will be consumed by the first argument that we pass to useReducer a so called reducer function.

reducerFn – (prevState, action) => newState

A func that is triggered automatically once an action is dispatched(via dispatchFn()) – it receives the latest state snashot and should return the new, updated state.

initialState – the initial state

initialFunc – a func to set the initial state programatically

const [emailState, dispatchEmail] = useReducer(emailReducer);

and above the outside component,

import Button from "../UI/Button/Button";

const emailReducer = (state, action) => {

return { value: "", isValid: false };

};

const Login = (props) => {

We can put the reducer func outside the component, bcs it doesn’t need to interact with anythings that defined inside of the component func. All the data that need inside this reducer func will be passed into this func when its executed by react automatically, so that why we can define it outside the component func here.

This reducer func receives two parameters, our last state snapshot and the action that was dispatched.

const [enteredEmail, setEnteredEmail] = useState("");

const [emailIsValid, setEmailIsValid] = useState();

combine this two into one state using reducer.

const [emailState, dispatchEmail] = useReducer(emailReducer, {

value: "",

isValid: false

});

Replacing the enteredEmail with emailState.value and emailIsValid with emailState.isValid.

dispatchEmail('NEW\_EMAIL\_VALUE');

dispatchEmail(12);

reducer dispatchFN it can have anything, it could be a string, it could be a number but often it would be an object.

dispatchEmail({ type: "USER\_INPUT", val: event.target.value });

in this object we can have, type and payload named val with the latest email value.

Action structure should be same for all dispatch func.

import React, { useState, useEffect, useReducer } from "react";

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

import classes from "./Login.module.css";

import Button from "../UI/Button/Button";

const emailReducer = (state, action) => {

if (action.type === "USER\_INPUT") {

return { value: action.val, isValid: action.val.includes("@") };

}

if (action.type === "INPUT\_BLUR") {

return { value: state.value, isValid: state.value.includes("@") };

}

return { value: "", isValid: false };

};

const passwordReducer = (state, action) => {

if (action.type === "USER\_INPUT") {

return { value: action.val, isValid: action.val.trim().length > 6 };

}

if (action.type === "INPUT\_BLUR") {

return { value: state.value, isValid: state.value.trim().length > 6 };

}

return { value: "", isValid: false };

};

const Login = (props) => {

*// const [enteredEmail, setEnteredEmail] = useState(""); // no use bcs of reducer*

*// const [emailIsValid, setEmailIsValid] = useState();*

*// const [enteredPassword, setEnteredPassword] = useState("");*

*// const [passwordIsValid, setPasswordIsValid] = useState();*

const [formIsValid, setFormIsValid] = useState(false);

const [emailState, dispatchEmail] = useReducer(emailReducer, {

value: "",

isValid: null

});

const [passwordState, dispatchPassword] = useReducer(passwordReducer, {

value: "",

isValid: null

});

useEffect(() => {

console.log("EFFECT RUNNING");

return () => {

console.log("EFFECT CLEANUP");

};

}, []);

*// useEffect(() => {*

*// const identifier = setTimeout(() => {*

*// console.log("checking from validity");*

*// setFormIsValid(*

*// enteredEmail.includes("@") && enteredPassword.trim().length > 6*

*// );*

*// }, 500);*

*// return () => {*

*// console.log("CLEANUP");*

*// clearTimeout(identifier);*

*// }; // cleanup process*

*// }, [enteredEmail, enteredPassword]);*

const emailChangeHandler = (event) => {

*// setEnteredEmail(event.target.value);*

dispatchEmail({ type: "USER\_INPUT", val: event.target.value });

setFormIsValid(event.target.value.includes("@") && passwordState.isValid);

};

const passwordChangeHandler = (event) => {

*// setEnteredPassword(event.target.value);*

dispatchPassword({ type: "USER\_INPUT", val: event.target.value });

setFormIsValid(emailState.isValid && event.target.value.trim().length > 6);

};

const validateEmailHandler = () => {

*// setEmailIsValid(emailState.isValid);*

dispatchEmail({ type: "INPUT\_BLUR" }); *// input lost focus - input blur*

};

const validatePasswordHandler = () => {

*// setPasswordIsValid(enteredPassword.trim().length > 6);*

dispatchPassword({ type: "INPUT\_BLUR" });

};

const submitHandler = (event) => {

event.preventDefault();

props.onLogin(emailState.value, passwordState.value); //

};

return (

<Card className={classes.login}>

<form onSubmit={submitHandler}>

<div

className={`${classes.control} ${

emailState.isValid === false ? classes.invalid : ""

}`}

>

<label htmlFor="email">E-Mail</label>

<input

type="email"

id="email"

value={emailState.value}

onChange={emailChangeHandler}

onBlur={validateEmailHandler}

/>

</div>

<div

className={`${classes.control} ${

passwordState.isValid === false ? classes.invalid : ""

}`}

>

<label htmlFor="password">Password</label>

<input

type="password"

id="password"

value={passwordState.value}

onChange={passwordChangeHandler}

onBlur={validatePasswordHandler}

/>

</div>

<div className={classes.actions}>

<Button type="submit" className={classes.btn} disabled={!formIsValid}>

Login

</Button>

</div>

</form>

</Card>

);

};

export default Login;

Above we can see, how the useReducer is used to set and use the states.

But doing like this is no0t optimal bcs we have two reducer with same func and formValid state depands on other state and that state may be old when its validating.

useEffect(() => {

const identifier = setTimeout(() => {

console.log("checking from validity");

setFormIsValid(emailState.isValid && passwordState.isValid);

}, 500);

return () => {

console.log("CLEANUP");

clearTimeout(identifier);

}; *// cleanup process*

}, [emailState, passwordState]);

So using form validation in useEffect is a ok way bcs at that time, the state we received is latest bcs useEffect will run only after state updates.

One problem here is, if we gave the valid password, and if again entered a char in the password again the useEffect runs and becomes valid and again it repeats like that.

To solve that, we can use object destructing and using it in the useEffect parameter.

const { isValid: emailIsValid } = emailState;

const { isValid: passwordIsValid } = passwordState;

emailIsValid & passwordIsValid is alias for isValid.

useEffect(() => {

const identifier = setTimeout(() => {

console.log("checking from validity");

setFormIsValid(emailIsValid && passwordIsValid);

}, 500);

return () => {

console.log("CLEANUP");

clearTimeout(identifier);

}; *// cleanup process*

}, [emailIsValid, passwordIsValid]);

After using like this, when we add extra char to the password it wont run the useEffect again.

But if we make the password short, useEffect will run because the validity will change.

This is optimization and to avoid unnecessary useEffect execution.

**10. Adding nested properties as dependencies to useEffect:**

In the previous lecture, we used object destructuring to add object properties as dependencies to useEffect().

1. const { someProperty } = someObject;
2. useEffect(() => {
3. // code that only uses someProperty ...
4. }, [someProperty]);

This is a **very common pattern and approach**, which is why I typically use it and why I show it here (I will keep on using it throughout the course).

I just want to point out, that they **key thing is NOT that we use destructuring** but that we **pass specific properties instead of the entire object** as a dependency.

We could also write this code and it would **work in the same way**.

1. useEffect(() => {
2. // code that only uses someProperty ...
3. }, [someObject.someProperty]);

This works just fine as well!

But you should **avoid** this code:

1. useEffect(() => {
2. // code that only uses someProperty ...
3. }, [someObject]);

Why?

Because now the **effect function would re-run whenever ANY property** of someObject changes - not just the one property (someProperty in the above example) our effect might depend on.

**11. useReducer() vs useState() for state mgt:**

Generally, you’ll know when you need useReducer().

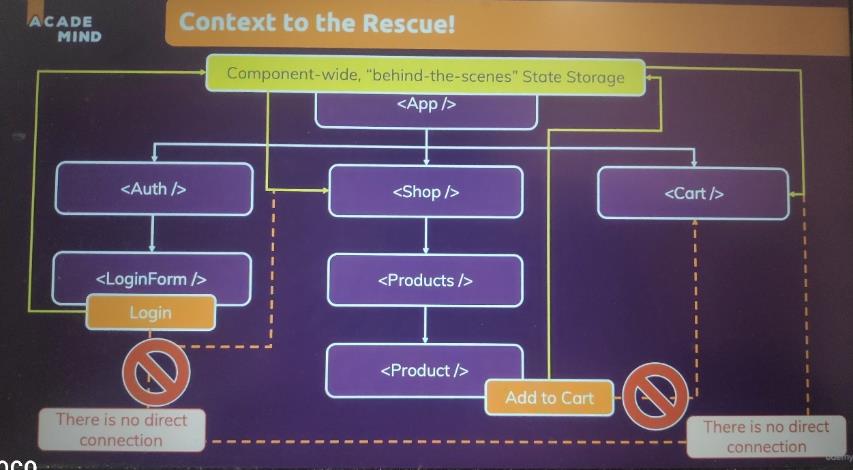
* + When using useState() becomes cumbersome or you’re getting a lot of bugs/unintended behaviours.

|  |  |
| --- | --- |
| useState | useReducer |
| main state mgt tool. | Great if you need more power. |
| Great for independent pieces of state/data. | Should be considered if u have related pieces of state/data. |
| Great if state updates are easy and limited to a few kinds of update. | Can be helpful if u have more complex state updates. |

**12. Introducing React Context:**

**Diagram

Description automatically generated**



So we can put the state in the app component and get the data wherever it needed across the component.

**13. Using the react Context API:**

The React Context which allows to manage state kind of behind the scenes in react, such that we actually able to directly change it from any component in our app and directly pass it to any component in our app without building such a prop chain.

React.createContext({});

Inside the parameter we can give default context, it may be string, array or object etc. Often it would be object. This createContext will return a component.

const AuthContext = React.createContext({

isLoggedIn: false

});

export default AuthContext;

AuthContext is a object that contains the component. And will export it to use in other context.

And to use context in our app, we need to do two things.

We need to provide it, which basically tells react, ‘hey, here’s my context’. All components that are wrapped by it should have access to it.

And besides providing, we then need to consume it.

1. Provider

App.js

import AuthContext from "./store/auth-context";

return (

<AuthContext.Provider>

<MainHeader isAuthenticated={isLoggedIn} onLogout={logoutHandler} />

<main>

{!isLoggedIn && <Login onLogin={loginHandler} />}

{isLoggedIn && <Home onLogout={logoutHandler} />}

</main>

</AuthContext.Provider>

);

Now the data in authcontext can be accessed in those child components.

1. Consumer:

Listening/consuming can be done in two ways, authContext.Consumer or react hooks.

<AuthContext.Consumer>

{(ctx) => {

return (

<nav className={classes.nav}>

<ul>

{ctx.isLoggedIn && (

<li>

<a href="/">Users</a>

</li>

)}

Consumer works bit different, like the consumer takes the child which actually would be a func. And as a argument we will get the context data.

But doing like the above will give an error(undefined variable), that bcs even we do have a default value in context but this default value will actually only be used If we would consume in child without having the provider.

Technically, provider doesn’t needed. If we have a default value, we dnt need this provider.

But in reality, you will use context to have a value which can change and that will be possible only with a provider.(to update u need a provider, for default value which will not change we don’t need provider).

To overcome the issues, we use value prop in provider.

<AuthContext.Provider

value={{

isLoggedIn: false

}}

>

<MainHeader isAuthenticated={isLoggedIn} onLogout={logoutHandler} />

<main>

{!isLoggedIn && <Login onLogin={loginHandler} />}

{isLoggedIn && <Home onLogout={logoutHandler} />}

</main>

</AuthContext.Provider>

Now by doing like this, we can able to change the object, for example, through state and the app component. And whenever it changes, the new value will be passed down to all consuming component.

<AuthContext.Provider

value={{

isLoggedIn: isLoggedIn

}}

>

By mapping it the isLoggedIn state to this value, whenever the isLoggedIn changed then the value in provider will also change.

FullCode:

App.js

<AuthContext.Provider

value={{

isLoggedIn: isLoggedIn

}}

>

<MainHeader onLogout={logoutHandler} />

<main>

{!isLoggedIn && <Login onLogin={loginHandler} />}

{isLoggedIn && <Home onLogout={logoutHandler} />}

</main>

</AuthContext.Provider>

Navigation.js

<AuthContext.Consumer>

{(ctx) => {

return (

<nav className={classes.nav}>

<ul>

{ctx.isLoggedIn && (

<li>

<a href="/">Users</a>

</li>

)}

{ctx.isLoggedIn && (

<li>

<a href="/">Admin</a>

</li>

)}

{ctx.isLoggedIn && (

<li>

<button onClick={props.onLogout}>Logout</button>

</li>

)}

</ul>

</nav>

);

}}

</AuthContext.Consumer>

**14. Tapping into context with the useContext hook:**

import AuthContext from "../../store/auth-context";

const Navigation = (props) => {

const ctx = useContext(AuthContext);

return (

<nav className={classes.nav}>

<ul>

{ctx.isLoggedIn && (

<li>

<a href="/">Users</a>

</li>

)}

It is simple and easier.

**15. Making context dynamic:**

We can notice but we still forward the onLogout prop through the child components still and that’s not be still ideal. We should not pass any prop butwe should make use of this logouHanlder to update the state.

<MainHeader />

{*/\* <MainHeader onLogout={logoutHandler} /> \*/*}

Now I just took logout handler, so the logout functionality will not work if we login.

And the good thing is, we can use dynamic context, that means we don’t just pass data, we can also pass a function.

onLogout – is pointing to a func that will update the state.

App.js

<AuthContext.Provider

value={{

isLoggedIn: isLoggedIn,

onLogout: logoutHandler

}}

>

<MainHeader />

{*/\* <MainHeader onLogout={logoutHandler} /> \*/*}

<main>

{!isLoggedIn && <Login onLogin={loginHandler} />}

{isLoggedIn && <Home onLogout={logoutHandler} />}

</main>

</AuthContext.Provider>

Navigation.js

{ctx.isLoggedIn && (

<li>

<button onClick={ctx.onLogout}>Logout</button>

</li>

)}

Making use of the onLogout func that will change the state.

In most case, we will use props to pass data to the components, bcs props are your mechanism to configure components and to make them re-usable.

Only if you have something which would forward to lot of components and you’re forwarding it to a component that does something very specific. Like for example, the navigation where this button will always log the user out. In such cases, we wanna consider context.

**16. Building & using a custom context provider component:**

const AuthContext = React.createContext({

isLoggedIn: false,

onLogout: () => {}

});

We do add this dummy onLogout for IDE auto-completion, if onLogout is not there then in ide it wont show.

Auth-context.js

import React, { useState, useEffect } from "react";

const AuthContext = React.createContext({

isLoggedIn: false,

onLogout: () => {},

onLogin: (email, password) => {}

});

export const AuthContextProvider = (props) => {

const [isLoggedIn, setIsLoggedIn] = useState(false);

useEffect(() => {

const storedUserLoggedInInformation = localStorage.getItem("isLoggedIn");

if (storedUserLoggedInInformation === "1") {

setIsLoggedIn(true);

}

}, []);

const logoutHandler = () => {

localStorage.removeItem("isLoggedin");

setIsLoggedIn(false);

};

const loginHandler = () => {

localStorage.setItem("isLoggedIn", "1");

setIsLoggedIn(true);

};

return (

<AuthContext.Provider

value={{

isLoggedIn: isLoggedIn,

onLogout: logoutHandler,

onLogin: loginHandler

}}

>

{props.children}

</AuthContext.Provider>

);

};

export default AuthContext;

Giving every details from App to auth-context and making it as a component and wrapping it in Index.js will make this more cleaner and good.

Index.js

ReactDOM.render(

<AuthContextProvider>

<App />

</AuthContextProvider>,

document.getElementById("root")

);

App.js

function App() {

const ctx = useContext(AuthContext);

return (

<React.Fragment>

<MainHeader />

{*/\* <MainHeader onLogout={logoutHandler} /> \*/*}

<main>

{!ctx.isLoggedIn && <Login />}

{ctx.isLoggedIn && <Home />}

</main>

</React.Fragment>

);

}

Now we can see this is very clean. One central context and one central state management.

**17. React context limitations:**

It can be great for app wide or component wide state. So essentially state that affects multiple components. But its not a replacement for component configuration.

const Button = (props) => {

return (

<button

type={props.type || 'button'}

className={`${classes.button} ${props.className}`}

onClick={props.onClick}

disabled={props.disabled}

>

{props.children}

</button>

);

};

In Button component, onClick may be used for different purpose and its reusable and not only for logout and sometimes there may not be onCLick itself. In home u can see onCLick but in login component there is no onClick we use submit type. So here we should use only props.

const Home = (props) => {

const authCtx = useContext(AuthContext);

return (

<Card className={classes.home}>

<h1>Welcome back!</h1>

<Button onClick={authCtx.onLogout}>Logout</Button>

</Card>

);

};

Login.js

<div className={classes.actions}>

<Button type="submit" className={classes.btn} disabled={!formIsValid}>

Login

</Button>

</div>

But in Home component, button here is for logout and only for logout, so we can use context here.

Props for configuration and context for state management across components or possibly across the entire app.

- React context is NOT Optimized for high frequency changes! So that means for ex, if you have state changes every second or multiple times per second, this is clearly not the case where we should use context.

**Note:**

What if I have a app-wide or component-wide state that changes often and?

In that cases we use **redux.**

**-** React Context also shouldn’t be used to replace allcomponent communications and props. Props are important and vital.

- Component should be still configurable via props and short “prop chains” might not need any replacement.

- if long prop chains, then it is worth to use.

**18. Learning the “Rules of Hooks”:**

**Rule 1:** Only call react hooks in react functions.(React Component functions & custom hooks)

**Rule 2:** Only call react hooks at the top-level in the component func or in custom hook func.

- don’t call them in nested functions

- don’t call them in any block statements

**Rule 3:** unofficial rule for useEffect(): Always add everything you refer to inside of useEffect() as a dependency.But the state updating functions of useReducer or useState are guaranteed by react to never change. And so we don’t need to add that as dependency.Even we could add that no issues.

useEffect(() => {

const identifier = setTimeout(() => {

console.log("checking from validity");

setFormIsValid(emailIsValid && passwordIsValid);

}, 500);

return () => {

console.log("CLEANUP");

clearTimeout(identifier);

}; *// cleanup process*

}, [emailIsValid, passwordIsValid, setFormIsValid]);

All the value we use should be in dependency.

If the data is from outside like from webservice call then no need to be added as dependency.

**19. Refactoring an input component:**

const Input = (props) => {

return (

<div

className={`${classes.control} ${

props.isValid === false ? classes.invalid : ""

}`}

>

<label htmlFor={props.id}>{props.label}</label>

<input

type={props.type}

id={props.id}

value={props.value}

onChange={props.onChange}

onBlur={props.onBlur}

/>

</div>

);

};

Making it reusable.

Login.js

<Input

type="password"

id="password"

label="Password"

value={passwordState.value}

isValid={passwordIsValid}

onChange={passwordChangeHandler}

onBlur={validatePasswordHandler}

/>

**20. Diving into “forward refs”:**

The hook which we gonna see allows us to interact with the input component imperatively which means not by parsing some state to it that then changes somethings in the component, but by calling a function inside of a component.

We should not use it often bcs its not typical react pattern you want but sometimes it is helpful.

const submitHandler = (event) => {

event.preventDefault();

if (formIsValid) {

authCtx.onLogin(emailState.value, passwordState.value);

} else if (!emailIsValid) {

emailInputRef.current.activate();

} else {

passwordInputRef.current.activate();

}

};

<Input

ref={emailInputRef}

id="email"

label="E-Mail"

type="email"

isValid={emailIsValid}

value={emailState.value}

onChange={emailChangeHandler}

onBlur={validateEmailHandler}

/>

This will result in error bcs func component cannot be given refs.

import React, { useRef, useImperativeHandle } from "react";

a hook which should be used rarely. Which exposes the object inside to the parent component.

Basically a translation object between internal functionalities and the outside world, so the parent component.

forwardRef is basically a func which we execute, a method which we execute, to which we pass our component func.

For focusing inputs,scrolling and triggering something programatiicaly this can be very useful.

With the help of useImperitiveHandle and React.forwardRef(), we can expose functionailities from a react component to its parent component to then use your component in the parent component through refsand trigger certain functionalities.

**SECTION 20: BUILDING A MULTI-PAGE SPA WITH REACT ROUTER:**

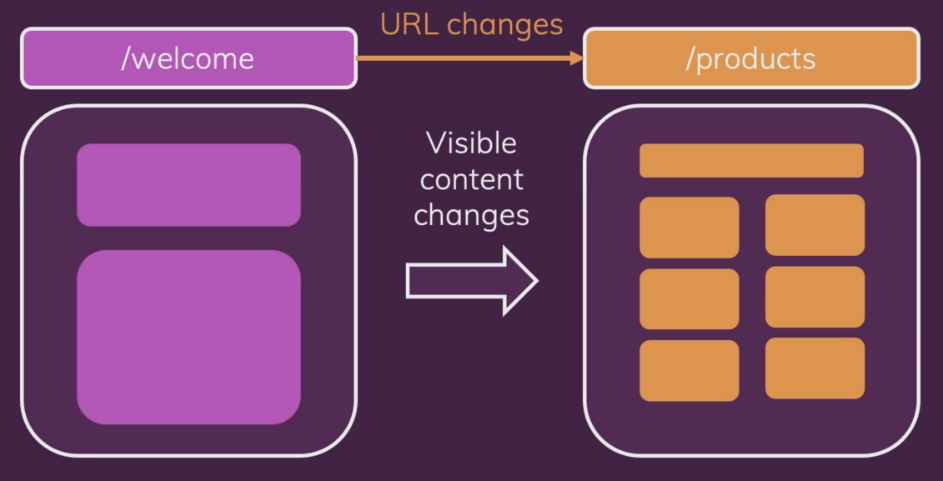
**1. Module Intro:**

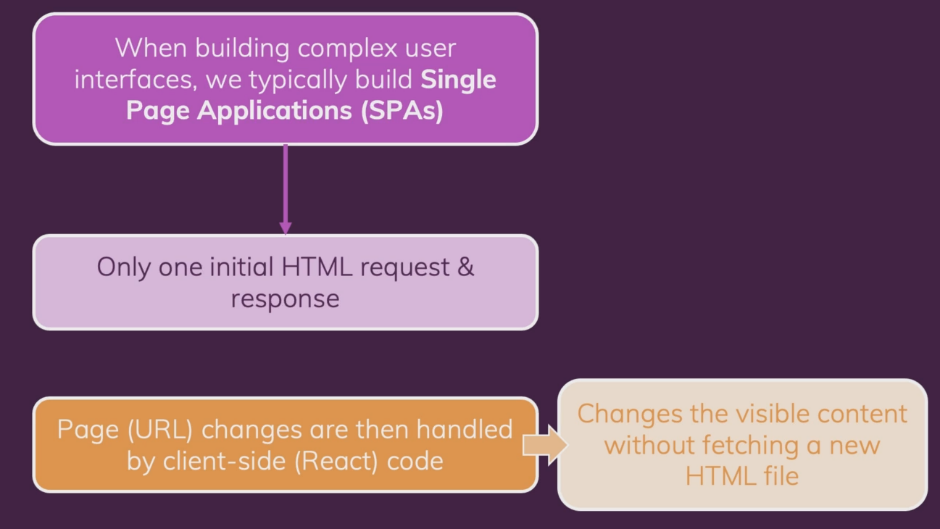
Client-side routing

React-router

**2. What is routing & why?**

Different paths load different pages.





**3. Installing React-Router:**

npm install react-router-dom

**4. Defining and using routes:**

import { Route } from "react-router-dom";

we register a route component in the App.js. Like below.

export default function App() {

return (

<div>

<Route path="/welcome">

<Welcome />

</Route>

<Route path="/products">

<Products />

</Route>

</div>

);

}

But to make it to work we need to do something else to the route.

const rootElement = document.getElementById("root");

ReactDOM.render(

<BrowserRouter>

<App />

</BrowserRouter>,

rootElement

);

Adding this BrowserRouter to the index page, unlocks the router features and Route in app.js will work now.

Basic routing setup is finished with this.

**5. Working with Links:**

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

const MainHeader = () => {

return (

<header>

<nav>

<ul>

<li>

<Link to="/welcome">Welcome</Link>

</li>

<li>

<Link to="/products">Products</Link>

</li>

</ul>

</nav>

</header>

);

};

export default MainHeader;

import { Route } from "react-router-dom";

import Welcome from "./pages/Welcome";

import Products from "./pages/Products";

import MainHeader from "./Component/MainHeader";

import "./styles.css";

export default function App() {

return (

<div>

<MainHeader />

<main>

<Route path="/welcome">

<Welcome />

</Route>

<Route path="/products">

<Products />

</Route>

</main>

</div>

);

}

Links and routes are given.

**6. Using NavLinks:**

Navlinks basically work like std Link. It still creates the anchor tag, captures the click, prevents the browser default but it does more than that.

NavLink also set css on active link. We only need to tell which class to link.

<li>

<NavLink activeClassName={classes.active} to="/welcome">

Welcome

</NavLink>

</li>

<li>

<NavLink activeClassName={classes.active} to="/products">

Products

</NavLink>

</li>

**7. Adding dynamic routes with param:**

<div>

<MainHeader />

<main>

<Route path="/welcome">

<Welcome />

</Route>

<Route path="/products">

<Products />

</Route>

<Route path="/product-detail/:productId">

<ProductDetails />

</Route>

</main>

</div>

Dynamic id is set using the identifier, now to get the param.

**8. Extracting Route params:**