**React and Redux**

**React:**

It is library for web and native interface.

A Library for building user interface.

<https://amdocsglobal.udemy.com/course/react-the-complete-guide-incl-redux>

# React-module:

Component driven JS library

**Components**: Reusable building-blocks in your user interface. It is combination of HTML, CSS and JS code.

Why components:

**Reusability**: Don’t Repeat yourself.

**Separation of Concerns:** helps to manage code. Each task do its specific work.

**How is a Component Build:**

It is basically combination of HTML and JS (and CSS) code.

React Uses Declarative approach: Define the desired target state(s) and let React figure out the actual JavaScript DOM instructions.

## JSX:

Stands for JavaScript XML.

It allows to write HTML code inside JS.

## How React Works:

React is all about Component (customer HTML element and JS).

It uses declarative way. Where as JS use imperative way. JS define steps.

## Props:

Passing value to child component.

 <ExpenseItem expense={expenses[0]}></ExpenseItem>

Access it ->

const expenseDate = props.expense.date;

while receiving the component we get only object combined all fields. We can de-structure the object and use the fields or can use as combined object.

## Composition:

It is mostly component only. One Basic example of composition is using shell container. Like card where want use some basic style and html structure in many component but internal structure of those component may be different.

Custom React component does not support existing styles and child content. So React provide special props **children**. It is reserved key-word in react to provide all children component. Also we can add style by adding parent style by **props.className**.

## Working with Events:

All JSX html-element has access to native html dom-events.

React exposes all DOM-events with “on” key-word

onClick:

while defining onClick dethod we only point the method and don’t call it.

Props:

Data passed to child component, normally know as Props. We can pass data, function via props to child component.

Convention to write function name with **on** keyword(not necessarily) like {onCartSubmit: }

# State:

If thee there is no hooks/life-cycle component then react only render the page only once, at the page initialization. It renders through “root.render(<App />”

useState return an array containing two field , first one which contain the value and second is setter function which used to update the value.

*const [expenseTitle, setExpenseTitle] = useState(props.expense.title);*

whenever we call setter method it not only update the variable, it re-evaluate entire component. And render the changes in screen.

If we want to update any state where it uses previous state, don’t use current state field, as react schedule event and during it might update fields indirectly.

Ex: if want use counter which update every click

**Best way: setCounter((prevState) => prevState + 1)**

Instead of : setCounter(counter+ 1)

**eventHandler function:**

in react if we want to use any function in any event, we have use as below:

<button onClick={editExpenseHandler}>Edit Expense</button>

Here we are just mentioning method but calling it, React will call the function when particular event Is triggered.

If want pass any any parameter then we have to use arrow function, as can’t use parenthese to pass parameters. Below is the format:

<button onClick={(event) => editExpenseHandler(‘title’,event.target.value}>Edit Expense</button>

## Lifting State up:

It means to pass the component data to parent component to use there or to pass other child component.

## Derived/Computed State:

If there is any field with which is directly related to some other state, we can call it derived/computed stated. In react no need to use direct state for this. Instead use one variable and set it based on original state in global scope.

## Stateless vs stateful component:

If any react component has its own state, we call it as stateful component. If there is no state directly used in component we call it stateless component. We can have state from parent in stateless component.

## React list:

we can display list/array in jsx using JavaScript map functionality.

Keep in mind, while using map, please provide key to any unique value, otherwise we may fall in bugs. Like State will be updated properly, as react does not with state in updated.

{selectedExpenseList.map(expense =>

                <ExpenseItem

                key ={expense.title}

                expense={expense}></ExpenseItem>

            )}

## Conditional Content:

In React, we can display conditional content **via ternary operator** or simply via setting displayable content in any JavaScript variable like

const content = <button onClick={() => setOpenAddExpenseForm(true)}>Add new expense</button>;

return(

        <div className="new-expense">

            {content}

        </div>

# Styling React Component:

1. Vanila CSS
2. Scoping Styles with CSS Modules
3. CSS in JS styling with Styled component
4. Tailwind CSS
5. Static and Dynamic Styling

## Vanila CSS:

We can import a CSS class to a JS file and can write normal CSS.

CSS will not scoped to file where it injected but it is global to all JS file.

*import “./index.css”*

### Inline Styling:

In React JSX code we can use inline style as html code with slight modification. Here Style has to object and inside curly braces. It looks like double curly-brace but its not any special syntax. Also we need to use camelCase for multi-word css style name like (background-color will become backgroundColor) or alternatively use quote ‘background-color’ as key.

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

        <input style={{

          borderColor: !isValid ? 'red' : 'black',

          background: !isValid ? 'salmon' : 'transparent'

        }}

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

### Dynamic CSS class:

In React we can dynamically add CSS class.

 <div className={"form-control " + (isValid ? "" : "invalid")}>

## Scoping Styles – CSS Modules:

Make sure CSS styles are scoped to file itself where it imported not the global like vanilla CSS.

It needs to be configured in project setup. In CRA setup it is already included.

It helps us to create scoped css syles which only limited to required React component. For this we have to change css file name to **styleName.module.css** and ad update import as below. While using the css className, give name like style.className.

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

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

React while processing change className to unique value and apply to the element.

For dynamic classes we use string literal or string concatenation along with logic

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

## Styled Component:

npm install styled-components

In React we add styled-components package and use there. It normal JS feature;

tagged template.  
 below component can be used to have the CSS styles( label is child component with some css)

const ControlledContainer = styled.div`

display: flex;

  flex-direction: column;

  gap: 0.5rem;

  margin-bottom: 1.5rem;

  & label {

display: block;

}

Here can add extra class in the jsx element or simply pass the value in styled component dynamically update. We also add media-queries as simple css.

Hover and media-query properties can be written like this:

*&:hover*

*@media (min-width: 768px) {*

*margin-bottom: 4rem;*

*& h1 {*

*font-size: 2.25rem;*

*}*

*}*

**Tailwind CSS:**

CSS framework

npm create vite@latest my-project -- --template react

npm install -D tailwindcss postcss autoprefixer

npx tailwindcss init -p

content: [

"./index.html",

"./src/\*\*/\*.{js,ts,jsx,tsx}",

],

This is more like adding class in html file and respective css will be be added via tailwind,

Sample classname are:

className='flex flex-col items-center mt-8 mb-8'

className='text-4xl font-semibold tracking-widest text-center uppercase text-amber-800 font-title'

we can add custom font as well

hover: -> no hover

md: -> media queries

# React Fragments:

# React Portal:

# React Reference / Ref:

It allows to control other element and through hooks. Via this we can read an element value and manipulate the element. But it is recommended not to manipulate the element.

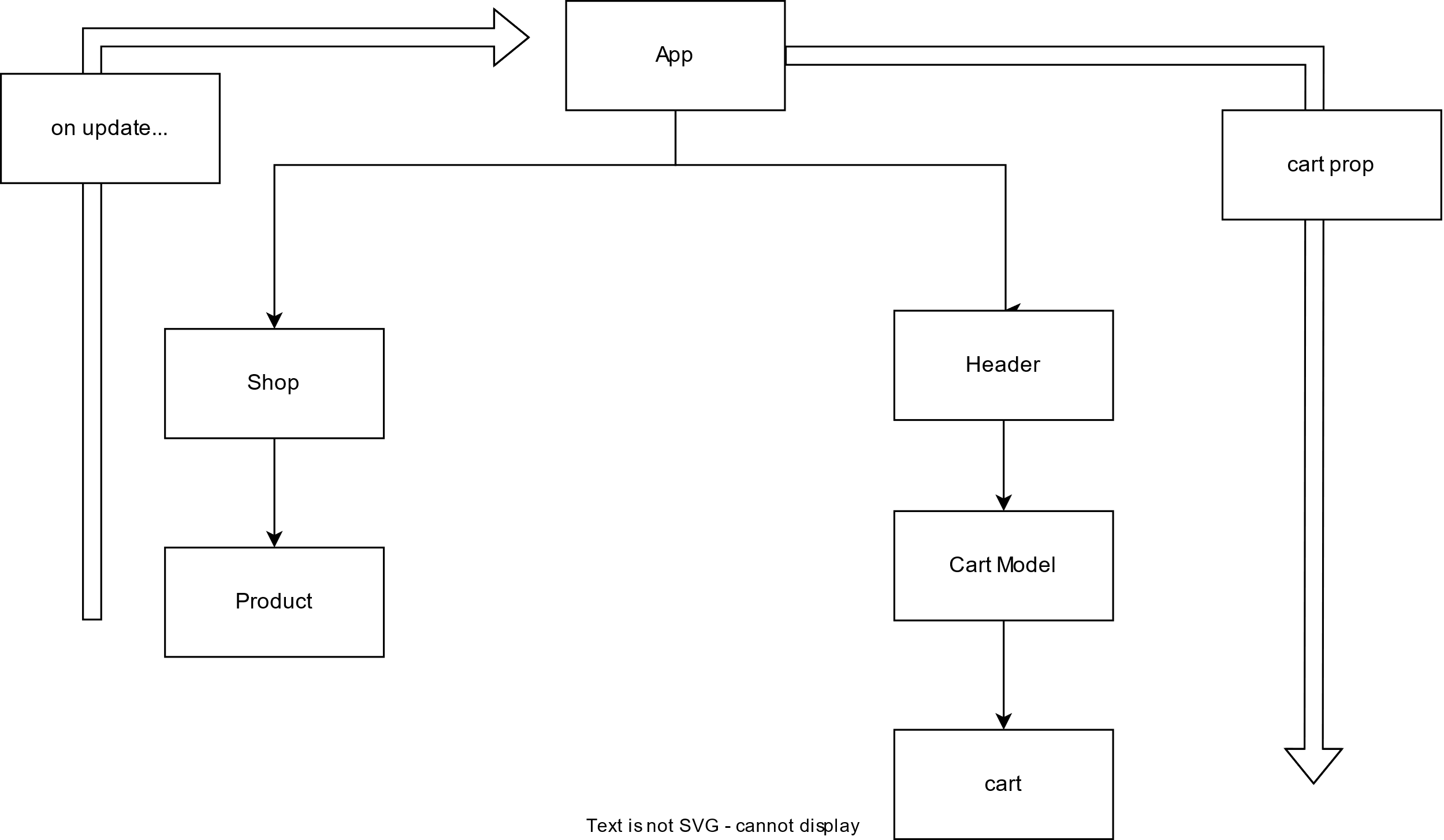
Controlled element vs controlled element:

# React Context API and useReducer:

1. Prop Drilling problem.
2. Embracing Component Composition
3. Sharing State with Context
4. Managing complex state with reducer

## Understand prop-drill:

Passing shared data through multiple components layer just to pass data. Unless then don’t have any requirement of those data.



## Component Composition:

Instead of passing data to child component, make it as a wrapper and main logic write it in main component.

## Context API:

It helps to share data across component. Context will wrap component and all component can access state values directly. It removes props drilling.

Usually context(shopping-cart-context) stays in store folder. createContext helps to create react-component .

export const CartContext = createContext({ items: [] });

export default CartContextProvider({children}) {

//states

//functions

//ctxValues with states and function (items, addItemToCart, updateItemQuantity)

return <CartContext.Provider value = {ctxValues}>

{children}

</CartContext.Provider>

}

use this context :

<CartContext.Provider values={shoppingCartState}> .. </CartContext.Provider>

Consuming Context value:

useContext hook and CartContext to be import

*const cartCtx = useContext(CartContext);*

another way to consume:

*<CartContext.consumer>*

*{(cartCtx) => {*

*return ()*

*}}*

default value set in context creating only used when component is not wrapped by the Provider component and helps in auto-complete.

## useReducer:

a function that reduces one or more complex values to a simple one is called reducer.

useReducer is state management system, it returns two element state and a dispatch function.

shoppingCartReducer will accept latest state and action.

*const [shoppingCartState, shoppingCartDispatch] = useReducer (shoppingCartReducer, [])*

we can call dispatcher function to add/update state with an identifier and optional payload or other fields.

*shoppingCartDispatch ({type: ‘ADD\_ITEM’, payload: id});*

*function shoppingCartReducer (state, action) {*

*if ( action.type == ‘ADD\_ITEM’) {*

*updatedItems = …*

*return {… state, items: updatedItems}*

*}*

# Handling Side-effects:

React mostly is for rendering the content, evaluate fields and re-render again. But there is something which is not directly connected for rendering the content like sending http request/ time-intervals etc. we can’t/ don’t want to use the directly in react components as most of the time, we don’t want to impact these fields whenever component evaluate.

useEffect( () => { … } , [dependencies])

function gets called if any of the dependencies gets changed.

Debounce effect:

We can delay useEffect function call by setTimeout and debounce it. It will call only after certain time passed.

Clean-up function:

We can return a function in useEffect hook. It call the function every time before useEffect calls (except first time).

## UseReducer:

Some-times, if we have complex state – for example multiple states, multiple way of changing it or dependencies to other states. useReducer can be replacement for useState(), if we need “more powerful state management”.

Try to avoid using other two different kind of state during state-setter function.

We can combine those related state or can use useReducer.

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

**dispatchFn**: A function that can be used to dispatch new action(i.e. trigger an update of the state)

**reducerFn**: A function that is triggered automatically once an action is dispatched (via dispatchFn()) – It receives latest snapshot and should return the new, updated state.

**initFn**: in optional function, can be used to initialize the state in in-case it is complex.

## useState vs useReducer:

state used mostly for main state management tool, few kind of updated and independent field.

useReducer user for more complex, inter-related states. Many different kind of state updating ways.

## useContext:

component-wide “behind-the-scenes” state storage system.

It helps us store states in single place (mostly in App component) and we can directly fetch the required stated in any component without passing via props-chain. Basically it remove prop chaining problem.

Firstly crate store via below:

const AuthContext = React.createContext({

isLoggedIn: false

});

AuthContext.Provider used to provide data to each child(and sub-childs) component. Without Provided it will get default data from store(where createContext defined) but it does have capability to update data. So we have to use provider and also it’s default data.

<AuthContext.Provider value={{

      isLoggedIn: false

}}>...</AuthContext.Provider>

AuthContext.Consumer used to consume data passed from provider/store.

<AuthContext.Consumer>

      {(ctx) => {

        return (  ...  )

      }}

</AuthContext.Consumer>

We can use contextHooks instead of Consumer it is more compact/elegant way to consume states.

const ctx = useContext(AuthContext);

We can use props and context-hooks simultaneously . When we use data directly in child component, we can use props (technically we can use context-hooks here as well).

We can also maintain all authentication related details in auth-context store and remove details from App component. Just use wrap App component with newly created AuthContextProvider component.

## Limitation of context:

React context is not optimized for high frequency changes.

It should not replace all component communications and porps. Just use where need to shorten props-chaining.

## Rules of Hooks:

1. Only call React Hooks in React Function( React Component Functions and Custom Hooks).
2. Only call React Hooks at the Top Level (should not be used in blocks/ nested function).
3. In useEffect, always add everything that refers inside as a dependency.

**Forward Ref:**

Sometimes we want to use ref of component from parent component. (i.e. we want to control/read some child component element from parent). This can we achieved via forwardRef.

First define child component inside forwardRef and pass all useRef(of child) via useImperativeHandle. Everything in child-component create as normal useRef to control/read element/s.

const Input = React.forwardRef((props, ref) => {

    const inputRef = useRef();

    const activate = () => {

        inputRef.current.focus();

    }

    useImperativeHandle( ref, () => {

        return {

            focus: activate

        }

    })

...

...

<input

            ref={inputRef}

...

/>

...

In Parent component, we can define Ref and can be passed to child-component. And from parent ref, we can control child component element.

const emailInputRef = useRef();

...

emailInputRef.current.focus();

...

<Input

          ref={emailInputRef}

...

/>

...

React behind Scenes:

useMemo- component re-evaluation

useCallBack

# Class based React Component:

class Product extends Component {

Render() {

Return …;

}

}

Component is exported by ‘react’ library . It has few properties like props.

For class based component. We can use state via initializing and updating when required.

Constructor function called in class-React component is created/first loaded. Ehere we can initialize State like this.state = {…}. Naming is fixed and there is only one possible state. To update any state we can call setState method. Here unlike useState, it will only update field which are passed.It accept function to state updating.

## Component lifecycle:

componentDidMount() -> called once component mounted (was evaluated and rendered) .. like useEffect with no dependency

componentDidUpdate() -> called once component updated (was evaluated and rendered) .. like useEffect with some dependency. It accept two argument (prev Props and prev State)

componentWillUnmount() -> called right before component is unmounted (removed from DOM) -> it is equivalent to clean-up function in useEffect. But it will only call if component is removed from DOM (is useEffect, will call every time state dependencies are changed).

## Context:

We define and provide context like functional component. For consuming, we can use Context.Consumer or.

Define “static contextType = userContext”

By this way we can access only one context.

# Error Boundaries:

We can through error via class based React component from any function/ lifecycle method:

Throw new Error(“msg”)