React Training 22-01-2024

Thursday, May 5, 2022 3:10 PM

React - library - external requirements - 10+years old
Problem with react - no limitations - architecting the solution/structuring the solution - heavy dependecies on 3rd party library
Lesser the dependecy the better the application
Facility to create components
Virtual dom from react

Notes:
- we cannot / shouldnot render all data at once
- When there is a roadblock kill terminal and restart
- VS Code Extensions
  - auto close tag
  - autop rename tag
  - Error lens
  - Path intellisense
  - Paste JSON as code

JS is
- Weakly typed, Single threaded, multi-threading using async and call backs,
- Any long running activity cannot be run on singlethread so it has to be asynchronous

Angular - framework - everything inbuild - opinionated framework

Core : code
Dom : ability to change html at runtime
BOM : print, devtools, etc

Evert callback func is js function
Browser have ability to multithread for JS

Stack - single thread

A separate thread is created when a new web api is called which is managed by browser,

Every call back func is just a waiting for their turns to execute on to the stack, they can go to stack when it is empty. Event loop acts as meditator between stack and queue it tells if the stack is free

- Sync code is always executed first then async
- Sync (run by JS runtime) => Async (tun by Browser)

If thread is blocked entire solution is blocked ex: page is unresponsive

Dom is also a web api
- Virtual Dom is an Abstraction layer, a curtain and also it simplifies rendering
- It is amost powerful tool to
- Virtual DOM on every data change a clone of existing JS dom(3) with uodated(4) dataset was created then a comparision between a clone and JS dom was performed then it identified the diff between 2 DOMS (1change as per above case), once Identified only the diff was patched on as result. The major advantage is there was no more rewriting of entire thing.
- The comparison happens line by line
  - List1: A B C
  - List2: A D B C
- For more efficient comparison, tracking use keys (the key prop has to be unique)
  - List1: A: 101 B:102 C:103
  - List2: A:101 D:104 B:102 C:103
- Irrespective of the item removed the vdom thinks the last one as removed. So we use key.

```
Every time we work with list, key has to be used.
```

- The most expensive thing of JS is Rendering but not logic, the more re-rending the more time it takes
- In react rendering was made easy , allows to build components
- React was never designed to build application they are used to build UI widgets
- From JS perspective rendering is primary concern and Vdom is resolving this.

File Structure
- Public folder - to hold local respources, accessible to public, index.html
- Bundle.js whenever we save everything gets packaged into that
- SRC entire application code resides, anything outside src doesn't allow code packaging,
- Flow of execution => Index.html(vdom gets created) -> index.tsx (where Vdom gets executed)
- Strict mode is development mode in index.tsx
- App component is first compo that gets loaded, that is supported by app.tsx

Components:
- Name of file and component should be same
- One component definition per file
- Naming convention of the component (Pascal case)
- Camel case for predefined pascal for custom component
- Class based convention and Function based convention
- Class

- Function
- React does not offer anything beyond component

Designing components (Design Principle)
- Smallest resuable entity
- We can create our own type.ts if we have more types we can create it at component level or we can create at globally at src
- Single Responsibility Principle

Design Pattern : Smart-Dumb Component
- Smart : Logical : containers
- Dumb : Presentational : components
- Hierarchy
  - Smart: Parent - takes relevant decision and pass to child
    - Dumb: child - take info from parent no questions asked (not from where but how)
  - Note 1 smart component can have more than 1 dumb component

Example
- Product
  - Display the data
  - Fetch the data
  - Add the prod to cart
  - Navigate
  - Wishlist
  - App:
    - ProductList - Smart Component: plist
    - Product - Dumb : data
  - Page 1: add to cart
    - Product -
  - Page 2: add to cart and Navigate
    - Product
  - Page 3:
    - Product: add to cart or Wishlist

- Component Communication : Parent <=> child
- Parent to Child: Props (properties) - right hand side is data
- Child to Parent: Props (events) - right hand side is event
  - The definition gets passed as a prop
  -
  - <component prop="data" prop="function()" />

Conditional rendering in React:
- We have to 2 techniques
  - Ternary Operator
    - EX:

```
 {data.productStock >0 ? (
                    <button onClick={()=>
this.props.btnClick(data.productId)}>Add to Cart</button>
                  ): null}
```

  - When readability is lost we can create another render
    - EX:

```
renderStock(stock: number, id:number){
        if (stock>0){
            return <button onClick={() =>
this.props.btnClick(id)}>Add to cart</button>
        }
        return <p>out of stock</p>
    }
```

- To pass array of objects we use array.map() to iterate over the list
  - EX:

```
plist.map((item)=><Product pdata={item}
                  btnClick={(id) => console.log(id, ": add
item")}/>)
```

  - Map func allows us to generate the components Dynamically and use
  - Without Key prop the, the VDOM renders multiple times using key prop reduces so much effort or rendering

```
{
                  plist.map((item)=><Product
                  key={item.productId}
                  pdata={item}
                  btnClick={(id) => console.log(id, ": add
item")}/>)
                }
```

- JS: Fetch API
- 3rd party library - axios (http client for node js)

- Use only well maintained library

Component Lifecycle Phases:

- Mounting
  - Constructor
  - Render: first render
  - ComponentDidMount: onload
- Updating
  - shouldComponentUpdate
  - Render: Future render (when there is an update)
- Unmounting

Components

- Props: Data exchange
- State: UI Updates
  - State should always be initialized
  - State should always be immutable
    - To achieve immutability Use setState : to update the state (shallow updates) , which keeps the state immutable internally and calls the render function automatically.
    - For Deep updates : use immer js

A class based component is a stateful component and lifecycle
A function based component is stateless, no state, no lifecycle and all functional components are pure components

In world of JS there is no value types, keep your data immutable as we cannot always unecessarily mess with the heap

Component && Pure Component(base class) :

- Address one of the major problem which is unnecessary rendering
  - A => B rerender
  - B => A rerender
  - A=>A no rerender
- Pure-component there is a built-in logic which checks for any data change then allows rerendering
  - EX:

```
class Demo extends React.PureComponent{
    state= { location: "A" };
    //every component must have a render method
    render() {
        console.log("Render", this.state);
        //every render must return something
        //each render should retuyrn single html
        //{} are called jsx expression which can be used to
call any const, variable
        const name = "varun";
        return (
        <div>
        <h1>Demo Component</h1>
        <p> some more content</p>
        <p>Hello from <h2>{name.toUpperCase()}</h2></p>
        <h2>{9+10}</h2>
        <button onClick={() => this.setState({ location:
"A" })}>A</button>
        <button onClick={() => this.setState({ location:
"B" })}>B</button>

        </div>
```

React Training 23-01-2024

Monday, January 22, 2024 10:54 AM

FORMS:

- Controlled - State => Vdom
  - Drawback of this is it rerenders the component

```
class Checkout extends React.Component{
    state = { name: "" };
    saveData(e: SyntheticEvent){
        e.preventDefault();
        console.log("checkout Submitted", this.state);
    }
    render() {
        return(
            <form onSubmit={(ev)=>this.saveData(ev)}>
                {/* Controlled Behaviour */}
                <input type="text"
                placeholder="Name"
                onChange={(e)=> this.setState({name:e.target.value})}
                ></input>
                <input type="text" placeholder="Email"></input>
                <button type="submit">Submit</button>
            </form>
        )
    }
}
```

- Uncontrolled we work with ref tells Vdom we need js dom object and internally it fetches the object. Basic object of this is to get DOM object.
  - As soon as the component loads we focus on the object using focus()
    - EX:

```
componentDidMount():void{
        if (this.emailRef){
            this.emailRef.focus();
        }
    }
```

  - This causes a small issue during validation as we are directly dealing withJ JS DOM and react cannot deal with it

```
class Checkout extends React.Component{
    state = { name: "" };
    emailRef: any = null;
    saveData(e: SyntheticEvent){
        e.preventDefault();
        console.log("checkout Submitted", this.state);
    }
    render() {
        return(
            <form onSubmit={(ev)=>this.saveData(ev)}>
                {/* Controlled Behaviour */}
                <input type="text"
                placeholder="Name"
                onChange={(e)=> this.setState({name:e.target.value})}
                ></input>
                {/* UnControlled Behaviour */}
                <input type="text"
                placeholder="Email"
                ref={(r)=>(this.emailRef = r)}></input>
                <button type="submit">Submit</button>
            </form>
        )
    }
```

Basically In form when click on submit the page reloads and log is displayed which causes logic not to be executed effectively

```
class Checkout extends React.Component{
    saveData(){
        console.log("checkout Submitted")
    }
    render() {
        return(
            <form onSubmit={()=>this.saveData()}>
                <input type="text" placeholder="Name"></input>
                <input type="text" placeholder="Email"></input>
                <button type="submit">Submit</button>
            </form>
```

To avoid this default behavior when submission of the reloading the page we follow a inbuilt event called syntheticEvent which helps logic execute effectively.

```
class Checkout extends React.Component{
    saveData(e: SyntheticEvent){
        e.preventDefault();
        console.log("checkout Submitted")
    }
    render() {
        return(
            <form onSubmit={(ev)=>this.saveData(ev)}>
                <input type="text" placeholder="Name"></input>
                <input type="text" placeholder="Email"></input>
                <button type="submit">Submit</button>
            </form>
        )
    }
}
```

In reallife we have a Library called formik and Yup Validator

Wrapper Component: UI

- HTML gives us 2 types of tags
  - <tag />- Empty tag
  - <tag>content</ tag> - Non empty tag

- UI components this allows us to

- Passing Data between Components
  - Props: Parent-Child
  - Context API: V16 introduced - It is a provider scope
    - There are 2 people
      - Provider: Single candidate
      - Consumer : multiple set of people
      - Seperation of concern is major issue
    - Only the descendants of the provider can be a consumer
      - EX: Here B can be direct provider for D as D is descendant of B
        - A
          - B: data : provider
            - C
              - D:use: consumer
          - E
            - F
              - G
          - H
            - I
          - J

      - If G is consumer A has to be provider as there is no direct dependency between B and G

        - A:provider
          - B: data
            - C
              - D
          - E

- F
  - G:use: consumer
- H
  - I
- J

- Redux: Application level state management / session management. It is a global scope
  - Ex: To handle multiple things which means if the nesting is too high we have to use this
- If complexity is high use context or redux

- React Hooks: they allow us to add a functionality to a functional component which in general is available with class components
- It is more like plug and play, they are meant only for functional components,
- We have like built in and custom hooks

  - Components : Reusable HTML
  - Hooks: Reusable Logic
- Always perform side-effects within lifecycle hook only
  - Ajax
  - Dom
  - Timer functions

- Use useEffect for side effect

```
useEffect(()=>{
        console.log("effect called");
        document.body.className= "bg-"+theme;
    },[
        //dependecies
        //emprty array: didMount
        theme
    ]);
```

  - useEffect takes 2 arguments
    - Logic
    - Dependency array

Router - it is 3rd party library

- SPA - npm install react-router-dom
  - APP Component
    - Static - Header and Footer
    - Dynamic - Footer
- Create a AppRouter file at src level and define routes in that file
- We can use navigation link - Link Component and buttons - UseNavigation Hook
  - For buttons we have to use useNavigate(a function from react router dom)
    - EX:

```
const navigate = useNavigate();

<button onClick={()=> navigate("/cart")}>Cart</button>
```

- Private/ Protected routes
  - We have to wrap the page which with the private router to make it private
    - EX: Here Checkout is a private route

```
<PrivateRoute>
                        <Checkout/>
 </PrivateRoute>
```

    - If we have an admin page not to be accessed by user mention Privateroute with role
      - If the role is user then never route to that page
- Route / URL Params
  - Path Params (required) (ex: Page number, product ID)
  - Query Params (optional) (ex: filters)

- Datasets:
  - Smaller : 20-30 records is fine

- Medium
- Larger > 50 to not load all of info together
    - Pagination
    - Virtual Scrolling - removes older content when new content is appended
    - Infinite Scrolling - does not remove older content but keeps on adding new content

State Persistence:

- How to retain the data(ex on reload of a page I have to have black theme), what is the rentention time ?
    - Short time: URL
    - Long term : Browser Storage, session Storage

React Training 24-01-2024

Wednesday, January 24, 2024 9:38 AM

Optimizing the Code - not just the logic but optimizing the code is also imp

- Lazy and Suspense : Code Splitting
  - Products: 100kb
  - Orders:50kb
  - Cart: 50kb
    - Bundle js: 200kb: 4sec page load time increase as bundle js size increases
    - Load any page only on demand
  - Use React.lazy which is called as dynamic import4

```
const LazyProductList = React.lazy(()=>
import("./Containers/ProductList"));
```

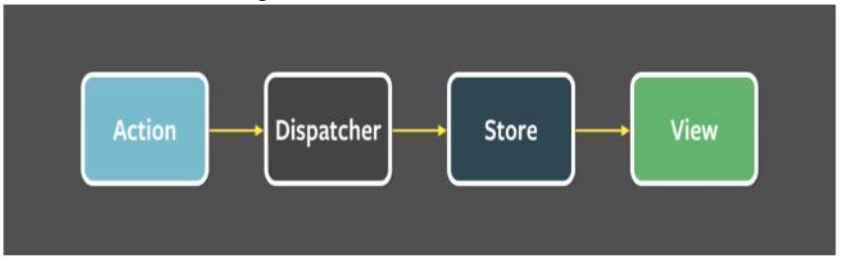
  - Whenever we create a lazy loading we need create a suspense use React.suspense

```
<React.Suspense fallback={<div>Loading ...</div>}>
```

  - If everything is lazy loaded it increases the load time. It is developers decision to check which one to lazily load
- Eager Loading
- Lazy loading

FLUX & REDUX

- Flux- Concept - it talks about unidirectional data flow technique, officially by FB along with react.
- Redux backlog: Redux doesnot support Async actions
  - Now it is no more a problem because of hooks
- Redux - Library that implements the above concept. : global
  - EX:
    - Store: Data Repo (tv : data)
    - Person(executive): Component - component cannot change data
    - Flux Diagram:




```
    if(theme === "light"){
        return <button onClick={()=>{setTheme("dark");
props.changeTheme('dark')}}>Dark</button>
    }
    return <button onClick={()=>{setTheme("light");
props.changeTheme('light')}}>Light</button>
}


function App() {
  let [theme, setTheme]= useState("light")
  return (
```

```
<BrowserRouter>
  {/* <Demo></Demo> */}
  <ThemeSwitch changeTheme={(t)=> setTheme(t)}/>
  <ThemeContext.Provider value={theme}>
    {/* <ProductList /> */}
    <Menu />
    <AppRouter/>
  </ThemeContext.Provider>
```

- Here onClick - action
- Dispatcher - props
- Store - useState
- View - Component <ThemeContext.Provider value={theme}>

➢ RTX: Redux Took Kit - a modernized/improved version of redux
- Debugging is not directly possible in redux

➢ First step towards Redux is :
- **Create Actions** - they are related to the data changes to be done in the store.
- Actions are functions which return a dispatcher object
- Every time you do an action it returns an dispatcher
  - Ex:
    - Theme
    - Cart
    - Login/logout - login & logout actions

- **Create a reducer**: Reducers are functions which analyze te type of dispatcher and return the updated data to the store. For ex: It is like a software used inside a store
  - They will hold all the logic related to the data store
  - The data we return from the reduced has to be **immutable**

    - Using setstate it handles the immutability, but redux we have to take care of the immutability

- **Create Store :**Once/app
  - Store is created by combining the data returned from the reducers.
    - Store belongs to the entire application

- **Provide the store to the app:** Done only once
- **Connecting Component to the store:** Who ever want to work with store has to come to the store.
- Bydefault store is not available.
- Every data we keep in the store is called slice, which means in the store we have slices of data
- Everytime you want to keep data in store make slice

➢ When ever we talk about action, we cannot directly call an action like regular function, every action must be dispatched

Redux FLOW

➢ In case of Redux on every action dispatched all the reducers are called/ invoked
- Internally all the reducers will try to analyze the type of dispatcher.
  - Once reducer finds the type reducer will execute the corresponding logic for that action and it returns updated data to the store.
    - Store publishes a notification to all the

connected components
- All the connected components run the mapping.
  - When ever the prop values changes.
    - Rerendering takes place

- In redux for functional component we use hooks but for class based components we use connect()()
  - Connect(how to connect)(what to connect)

- Class, connect = > Connect function is available for both
  - Function: useDispatch to read data from store

- **Redux Middleware :**
  - Managing the Storage
  - Async actions
    - Redux Thunk
    - Redux Saga

- Redux Persist - https://www.npmjs.com/package/redux-persist (refer for docs)

- Custom Hooks:

- React:
  - UI Widgets