IoIS technology background - Drávai Balázs

Table of contents

- IoIS technology background - Drávai Balázs
    - Table of contents
    - Introduction
    - History of Angular vs Vue vs React
    - Technical dephts
        - Comparison
        - React reconciler
        - React components
        - Lifecycle
        - More hooks
    - Project architecture
        - Redux
        - Redux-observable
        - Routing
    - UI elements
    - API
    - References

Introduction

In 2021, the top 3 most popular frontend frameworks are Angular, Vue and React. However, the last two options are not really frameworks, because they are just javascript frontend libraries to present UI components on the screen. In spite of that for the sake of simplicity I will refer them as frameworks. For my project work I have chosen react and redux, because I have 4 years of experience in React. (https://www.linkedin.com/in/balazs-dravai/)

History of Angular vs Vue vs React

In this next chapter, I will outline the key features of these frameworks and compare them to each other.

AngularJs first stable version released in 2010. However, the current versioning (12) is based on the 2.0, which was completely rewritten in 2016. It is a Typescript based framework maintained by Google. After the second version the term for the framework is changed to **Angular**. It is used by Weather.com, Google, Wix, etc.

React is developed and maintined by Facebook. Its first stable release was in 2013. It is a free and open-source front-end JavaScript for building user interfaces or UI components. Its only consern is the local (component locality) state management and the rendering to the DOM (Document Object Model). It is used by Airbnb, Uber, Netflix, Twitter, Pinterest, Reddit, etc.

Vue is developed by an independent developer, Evan You in 2013. Vue has been learning from the mistakes and successes of React & Angular. So in some cases it has similarities with them but it solve some tasks faster. It is used by Nintendo, GitLab, Alibaba, etc.

Technical dephts

Comparison

One of the most important thing that both **Vue* and **React** utilizes is the Virtual DOM. It is important due to the fact that the most important motivation behind these modern Javascript frameworks are the high cost of manipulating the DOM. Virtual DOM is a programming concept where a virtual representation of the UI stored and synched with the DOM. This mechanism is called "reconcilation". In the case of **React**, the library, responsible for this is called ReactDOM.

Angular does not have a virtual DOM, it uses their own change detection algorithm to going through from its root to their children.

Angular uses a html based template syntax and React uses jsx (see later.) but Vue can use either a html based template or a render function with or without jsx.

Angular is considered a complete solution because it provides strong opinions as to how your application should be structured and also has more functionality out of the box. However, React and Vue has more flexibility and their libraries can be paired to all kind of packages.

React uses a one-way binding, it first changes the model and then updates the elements, but Angular uses two-way data binding, which is cleaner to code but the React way is easier to debug because the data flow is unidirectional.

React reconciler

React elements are stored in a tree-like structure and when a `render` function of a component is called first a new node is created. After that, when an update happens the `render` function returns a different tree. To do this in an efficient way, the react reconciler has a diffing algorithm. First, it compares the two root elements. If these elements have different types, then it will tear down the old tree and build a new one. To indicate the deletion of the old elements, it calls their `componentWillUnmount` lifecycle method and after mounts the newly created elements it calls the corresponding `componentDidMount` method. (After `v16` these lifecycle methods can be replaced an equivalent mechanism with `hooks`)

For example:

```
<ComponentBefore>
    <ChildrenComponent/>
</ComponentBefore>

<ComponentAfter>
    <ChildrenComponent/>
</ComponentAfter>
```

If the two compared elements has the same type, react compares and updates their attributes.

```
<Component attr1="before">
...
</Component>

<Component attr1="after">
...
</Component>
```

After handling these elements it recursively go through their children and when there's a difference it generates a mutation.

For components rendering array of elements can suffer a huge performance loss, because if we for example add a new elements to the list, or update one. React does not know which elements it should keep or change, so it recreates the whole subtree.

```
<ul>
    {data.map(item => <li>{item.name}</li>)}
</ul>
```

This issue can be solved with a unique (if it is not unique it will be unstable and results an inconsistent state or unnecessary updates) `key` property, which helps to identify the elements.

```
<ul>
    {data.map(item => <li key={item.id}>{item.name}</li>)}
</ul>
```

React components

React components can be defined in two way. Either it can be a class based component or a functional component. Before `v16` a functional component represented a stateless component and a class component represented a component with internal state. However, it changed with the introducing of the hooks.

Though, I will present some examples with Typescript, **React** is not bound to ts like Angular.

Class component

```
class MyComponent extends React.Component<MyComponentProps,
MyComponentState> {
    state = {
        myState: 'initial'
    }

    render() {
        return (
            <>
```

```
                <h1>My component</h1>
                <div className="state-container">
                    {this.state.myState}
                </div>
            </>
        )
    }
}
```

React utilizes jsx for rendering elements. It is an optional prerpocessor for a html-like language which embeds html and custom elements into javascript functions. Props are passed to these elements as html attributes and children props is passed as the children of the html element. React can render a `ReactNode`, which is a `ReactElement` or `string` or `null` or an array composed one of these types. (e.g. `string[]`)

Functional component (stateless)

```
const MyComponent:React.FC<MyComponentProps> = () => {
    return (
        <>
            <h1>My component</h1>
            <div className="stateless-example">
                Hello world
            </div>
        </>
    );
}
```

A component also can be defined as a function. The rendered elements are defined by the return value of the function and for every update, the whole function will be evaluated.

With the `useState` react hook, it can be transformed into a stated component easily.

```
const MyComponent:React.FC<MyComponentProps> = () => {
    const [myState, setMyState] = useState('initial');

   return (
        <>
            <h1>My component</h1>
            <div className="stateless-example">
                {myState}
            </div>
        </>
    );
}
```

There is a key diffence between a class component's states and a functional one's. The class component stores its internal state in an object and provides a private method called `setState` to update the whole or the part of that state object. Although, the `useState` hook only represents one state instance (However, it can be an object to similarize with the class) and its corresponfing state transition function, like the `setMyState` function in the example above.

It is important that ttate updates not happen in a synchronized way, instead whenever a state change is initiated, it is pushed into a queue, and it will be executed sometime. If multiple state transition function called after each other, therefore for every state update a new rerender will happen until `v18`. The new version introduces automatic batching feature to help this.

Lifecycle

In this paragraph, i will show the lifecycles of a component with both class based and hook based examples.

```
class A extends React.Component {
    componentDidMount() {
        /*
            This method is called when
            a component is first rendered.
        */
    }

    componentWillUnmount() {
        /*
            This is called when a
            component will be destroyed by the reconciler.
        */
    }

    componentShouldUpdate(prevProps, nextProps) {
        /*
            This should returns a boolean value to determine
            from the
            incoming props wether it
            should be rerendered or not.
        */
    }

    componentDidCatchError(error) {
        /*
            Error boundary for the component
        */
    }
}
```

The `useEffect` hook is responsible fo handling both the `mount` and the `unmount` states. There is a `useLayoutEffect`, which not only waits for the render but also waits for the `window.onload` event to happen.

```
const B = () => {
    useEffect(() => {
        /* Equivalent of component did mount */

        return () => {
            /*  Equivalent of component will unmount*/
        }
    }, []);
}
```

For performance purposes, we can also limit the cases when a component should update for functional components also. This functionality is not provided by a hook but a built-in higher order component called `memo`. It takes two argument. The first is a react component and the second is a comparison function which is similar to the class component's `componentShouldUpdate` method.

More hooks

Because functional components are evaluated in every update, it is paramount to memoize the "methods" and the values that have high cost to compute.

```
const C = () => {
    useCallback(() => {
        /* Memoized method */
    }, []);

    useMemo(() => {
        /* Memoized value */
    }, []);
}
```

Project architecture

Now that we have a picture about react, we can move on to the other building blocks of the project. React has a pretty decent state management architecture, and a unidirectional data flow pattern, which helps the developers to maintain the states consistent. However, in real world application there are many cases where multiple components have to share the same state. (Or even modify). If a state is used by a lot of components in the component tree, it has to be defined in their root component and should be pass down level by level for every component that uses or one of their uses it. It is a highly unrecommended practice, called prop drilling.

Instead **React** provides a context api that helps us share these type of states between our components. In the context of the consumer components it is considered a "global" state, but it not. It is defined in one of their root elements which is the provider.

Redux

Redux utilizes the previously mentioned context api to provide an object of "globalized" state. This state is immutable (thus, cannot modified directly) and can only be modified through reducers and actions. (To reduce the boiler plate that is needed for redux, I used redux-toolkit.)

State updates happens in the next way: A component dispatches an action. This action has a type and optionally, a payload. It flows through the registered reducers, that recognize or ignore the action. If an action is known by the reducer it updates that part of the state. It is important that the state update will not be mutated, instead a new copy with the updated values will be created and notify the component to initiate a rerender. This pattern helps to keep the state consistent, since the store acts as a singel source of truth.

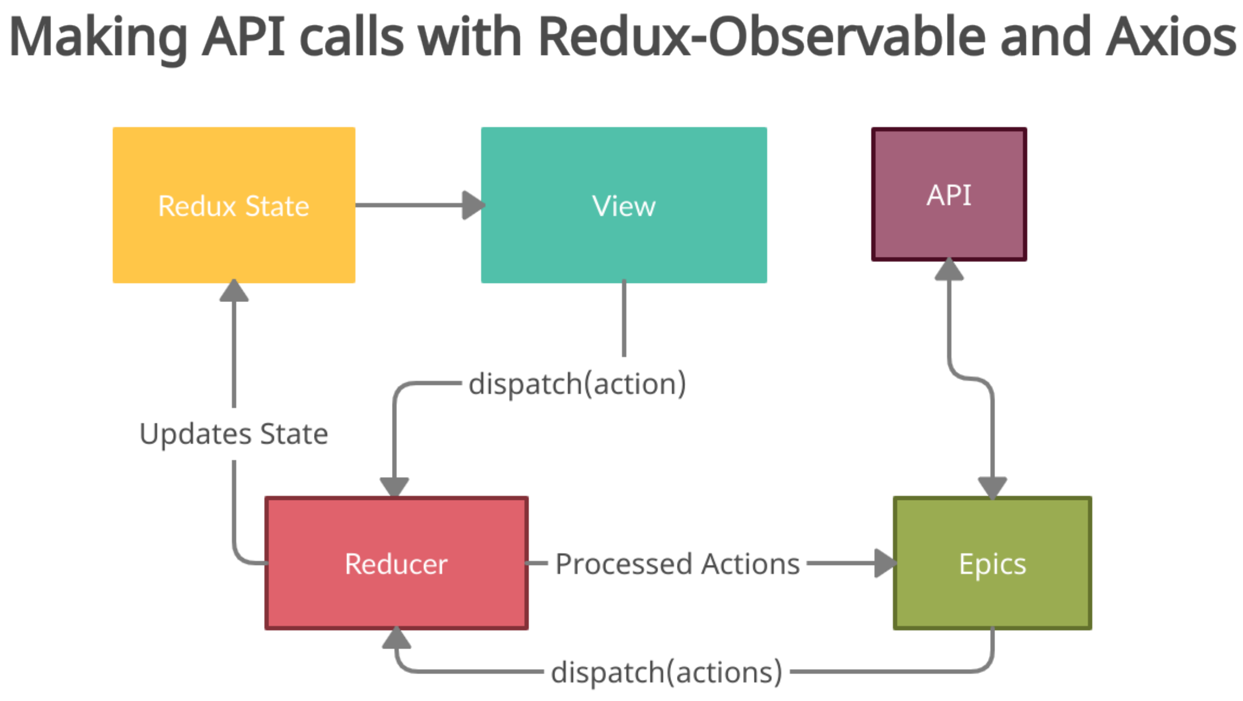
```
export const counterSlice = createSlice({
  name: 'counter',
  initialState: {
      value: 0,
  },
  reducers: {
    increment: (state) => {
      // state act only as a proxy object and
      // we are not mutating the original state
      state.value += 1
    },
    decrement: (state) => {
      state.value -= 1
    },
  }
})

const Component () => {
    const dispatch = useDispatch();
    // dispatching action
    dispatch(increment());

    const { count } = useSelector(state => state.counter);

    return (<div>{count}</div>)

}
```

Redux-observable

For managing side effects intuitively, I used a redux middleware called `redux-observable`. It is based on the popular reactive library `rxjs`, With this we can define `epics`, the core primitives of `redux-observable`, which is a function that takes a stream of actions in and returns a stream of actions out. The emitted actions will be dispatched immediately, through the normal `store.dispatch()` function.

```
epic(action$, state$).subscribe(store.dispatch)
```

Epics run after the reducers have already processed them, but they can produce multiple other actions.



Routing

For routing I used `react-router` v5 which provides component to take advantege of the browser's history API and helps rendering components based on the url.

```
export default function BasicExample() {
  return (
    <Router>
        <Switch>
          <Route exact path="/">
            <Home />
          </Route>
          <Route path="/about">
            <About />
          </Route>>
        </Switch>
    </Router>
  );
}
```

I also used `connected-react-router`, which helps dispatching navigation actions outside of the component tree. (For example, in the `epics`)

UI elements

I used primereact's design-agnostic ui elements.

API

I will omit the discussion of the underlying REST API because the main focus of this project is on the frontend development but there will be the same api implemented in .NET core and Java Spring Boot also - because they are required for another course - to produce the same interface I use swagger the define and document the endpoints.

References

> Redux observable documentation -- *https://redux-observable.js.org*

> Vue guide -- *https://vuejs.org/v2/guide/*

> React docs -- *https://reactjs.org/docs/getting-started.html*