React App optimisation performance

How to optimise performance to deliver an amazing user experience.

At MachineMax optimisation is the number one thing that is on the mind of every developer when building software, including our web app. We use React with Typescript to build not only a software but an amazing experience for the user, keeping in mind speed, code maintainability and code reutilisation. In this post I'll review some features and tricks that we used to optimise our app's performance.
It's always important to keep our code DRY. Always strive to reuse components as much as possible, that's guaranteed to help in writing optimised code. To write a great app the first trick that I would like to suggest is to spend more time writing excellent code and less time re-writing mediocre code. This seems an obvious option but it's good to remember before starting any pieces of code. PLAN, PLAN and PLAN first.

Let's start to see some of the features of React that we use to optimise our webclient.

PureComponent

We are removing all Class components from our app but due the large amount of components and based on the complexity/time to convert everything in pure functional is good to know a way to optimise existing class components.
Just like what shouldComponentUpdate does to class components, so also PureComponent do.
React's PureComponent is a base component class that checks the fields of state and props to know whether the component should be updated.

Let's check an easy example to understand better the concept:

```
class MyVeryOldClassComponent extends Component {

constructor(props, context) {
    super(props, context);
    this.state = {
      searchText: null
    };
    this.inputValue = null;
  }

  handleClick = () => {
    this.setState({ searchText: this.inputValue });
  };

  onChange = evt => {
    this.inputValue = evt.target.value;
  };

  shouldComponentUpdate(nextProps, nextState) {
    if (nextState.searchText === this.state.searchText) {
      return false;
    }
    return true;
  }
  render() {
    return (
      <div>
        {this.state.searchText}
        <input onChange={this.onChange} />
        <button onClick={this.handleClick}>Start the search!</button>
      </div>
    );
  }
}
```

Let’s convert this component to a React’s PureComponent:

```
class MySuperQuickSearchComponent extends React.PureComponent {
   constructor(props, context) {
       super(props, context)
       this.state = {
           searchText: null
       }
       this.inputValue = null
   }
   handleClick = () => {
       this.setState({searchText: this.inputValue})
   }
   onChange = (evt) => {
       this.inputValue = evt.target.value
   }
   render() {
       return (
           <div>
               {this.state.searchText}
               <input onChange={this.onChange} />
               <button onClick={this.handleClick}>Start search!!</button>
           </div>
       )
   }
}
```

As you can see, I have removed the shouldComponentUpdate and made the ReactComponent extend React.PureComponent.
Type any string in the textbox and click on the ‘Start search!’ button continuously, you will see that the ReactComponent will be re-rendered only once.
It shallowly compares the fields of the previous props and state objects with the fields of the next props and state objects. it doesn’t just do the object reference comparison of them. A great optimisation! React’s PureComponent optimises our components by reducing the number of wasted renders.

useMemo()

This is a React hook that is used to cache CPU expensive functions in React.

Let's see an example:

```
function App() {
   const [count, setCount] = useState(0)

   const myLongFunction = (count)=> {
       waitSync(3000);
       return count * 30;
   }
   const resCount = myLongFunction(count)
   return (
       <>
           Count: {resCount}
           <input type="text" onChange={(e)=> setCount(e.target.value)}
placeholder="Set Count" />
       </>
   )
}
```

We have an expensive function myLongFunction that takes about 3 mins to execute, it takes an input and waits for 3 mins before returning the multiple of 30.
If we type anything, our app component is re-rendered causing the expensive function to be called. We will see that if we start to type continuously, the function will be called every render causing a massive performance bottleneck. For each input, it will take 3 minutes to be rendered.
To solve this type of problem (not the 3 minutes of execution of course!!), we use useMemo hook to cache the function.
useMemo has this structure:

```
useMemo(()=> Function, [dependencies])
```

The myLongFunction is the function we want to cache/memoize, the dependencies list is the array of inputs to the myLongFunction that the useMemo will cache against, so if they change the Function will be called.
Now, using useMemo on our functional component:

```
function App() {
   const [count, setCount] = useState(0)

   const myLongFunction = (count)=> {
       waitSync(3000);
       return count * 90;
   }
   const resCount = useMemo(()=> {
       return myLongFunction(count)
   }, [count])

   return (
       <>
           Count: {resCount}
           <input type="text" onChange={(e)=> setCount(e.target.value)}
placeholder="Set Count" />
       </>
   )
}
```

Now, here the myLongFunction results will be cached against the input when the same input occurs again useMemo will skip calling the myLongFunction and return the output cached against the input.
This will make the App component highly optimised.
See, that useMemo caching technique to speed up performance. Also, it can be used to cache functional components against its props.

Virtualise long lists

If you render large lists of data, it is recommended that you render only a small portion of the data at a time within the visible viewport of a browser, then the next data are rendered as the lists is scrolled, this is called "windowing". Awesome React libraries have been built for this, but we try to avoid to use external dependencies when it is possible, so we have built our.
We use an Hook that accepts as parameters any reference to a HTML element as input and returns a boolean that specify if the element is visible or not.

```
const useVisible = <T extends HTMLElement>(
   enabled = true,
 ): [RefObject<T>, boolean] => {
   const ref = useRef<T>(null);
   const [isVisible, setVisible] = useState(false);

   const onScroll = () => {
     if (!ref.current) {
       return;
     }
     const shouldBeVisible =
       (ref.current.offsetTop > window.scrollY - window.innerHeight &&
         ref.current.offsetTop < window.scrollY + window.innerHeight * 2) ||
       (ref.current.offsetTop + ref.current.offsetHeight >
         window.scrollY - window.innerHeight &&
         ref.current.offsetTop + ref.current.offsetHeight <
```

```
          window.scrollY + window.innerHeight * 2);

    if (isVisible !== shouldBeVisible) {
      window.requestAnimationFrame(() => setVisible(shouldBeVisible));
    }
  };

  const registerListener = () => {
    document.addEventListener('scroll', onScroll, {
      passive: true,
    });

    document.addEventListener('wheel', onScroll, {
      passive: true,
    });
  };
  const unregisterListener = () => {
    document.removeEventListener('scroll', onScroll);
    document.removeEventListener('wheel', onScroll);
  };
  useLayoutEffect(onScroll, []);

  useEffect(() => {
    if (enabled) {
      registerListener();
    } else {
      unregisterListener();
    }
    return unregisterListener;
  }, [enabled]);
 return [ref, isVisible];
};
```

The hook will register for scroll and wheel on the DOM and check if the element is contained in the visible window. With this easy approach we can render a long list of data with a good optimisation. Unfortunately with this approach there are some limitations, so in some cases it is better to use external libraries like React-virtualized.

Caching functions

Functions can be called in the React component JSX in the render method.

```
function expensiveFunc(input) {
   ...
   return output
}
class ReactCompo extends Component {
   render() {
       return (
           <div>
               {expensiveFunc}
           </div>
       )
   }
}
```

If the function is expensive to execute, for example it takes long to execute, it will hang the rest of the re-render code to finish thereby hampering the user's experience.
The expensiveFunc is rendered in the JSX, for each re-rendered the function is called and the returned value is rendered on the DOM. The function is CPU-intensive, we will see that on every re-render, it will be called and React will have to wait for it to complete before running the rest of the re-rendering algorithm.
The best thing to do is to cache the function's input against the output so that the continuous execution of the function gets faster as the same inputs occur again.

```
function expensiveFunc(input) {
   … my expensive code …
   return
}
const memoizedExpensiveFunc = memoize(expensiveFunc)
const MyComponent: React.FC<WithSomeProps> = (props: WithSomeProps) => {
       return (
           <div>
               {memoizedExpensiveFunc}
           </div>
       )
}
```

Using reselect selectors

At MachineMax we use intensively reselect. Using reselect optimises our Redux state management. As Redux practices immutability that means new object references will be created every time an action dispatch. This will impact performance because re-render will be triggered on components even when the object references change but the fields haven't changed.
Reselect library encapsulates the Redux state and checks the fields of the state and tells React when to render or not if the fields haven't changed.
So, reselect saves precious time by shallowly traversing through the previous and current Redux states fields to check if they have changed despite having different memory references. If the fields have changed it tells React to re-render, if none fields have changed it cancels the re-render despite new state object created.

Lazy loading

Lazy loading has come to be one of the optimisation techniques widely used now to speed up the load time. The prospect of lazy loading helps reduce the risk of some of the web app performance problems to a minimal.
To lazy load route components in React, we use the React.lazy() API.

To be more detailed:
"*React.lazy is a new feature added to React when React v16.6 was released, it offered an easy and straight-forward approach to lazy-loading and code-splitting our React components.*
The React.lazy function lets you render a dynamic import as a regular component. — React blog"

React.lazy make easy to create components and render them using dynamic imports. React.lazy takes a function as a parameter:

```
React.lazy(()=>{})
```

or

```
function component() {}
React.lazy(component)
```

This callback function must load the component's file using the dynamic import() syntax:

```
// SuperComponent.tsx
const SuperComponent = () => {
   return <div>MyComponent</div>
}
```

```
const MyLazyComponent = React.lazy(()=>{import('./SuperComponent.tsx')})
function AppComponent() {
return <div><MyLazyComponent /></div>
}
```

The callback function of the React.lazy returns a Promise via the import() call. The Promise resolves if the module loads successfully and rejects if there was an error in loading the module, due to network failure, wrong path resolution, no file found, etc.
When webpack walks through our code to compile and bundle, it creates a separate bundle when it hits the React.lazy() and import().

Our app will become like this:

```
react-app
dist/
- index.html
- main.b1234.js (contains App component and bootstrap code)
- supercomponent.bc4567.js (contains SuperComponent)
```

```
/** index.html **/
<head>
  <div id="root"></div>
  <script src="main.b1234.js"></script>
</head>
```

Now, our app is now separated into multiple bundles. When AppComponent gets rendered the supercomponent.bc4567.js file is loaded and the containing SuperComponent is displayed on the DOM.
Be careful because every time a new version is released the bundle name may change. This can sometimes cause some issues in the correct flow. Refreshing the page will solve this inconvenience.

React.memo()

Just like useMemo and PureComponent, React.memo() is used to memoize/cache functional components.

```
function MyComponent(props) {
   return (
       <div>
           {props.data}
       </div>
   )
}
function App() {
   const [state, setState] = useState(0)
   return (
       <>
           <button onClick={()=> setState(0)}>Click</button>
           <MyComponent data={state} />
       </>
   )
}
```

App renders MyComponent component passing the state to it via the props. Now, see the button sets the state to 0 when pressed. If the button pressed continuously, the state remains the same throughout but the My component would still be re-rendered despite the state passed to its prop being the same. This will be a huge performance bottleneck if there are thousands of components under App and MyComponent. A very common case in a large project.

To reduce this, we will wrap the MyComponent with React.memo() which will return a memoized version of MyComponent, that will be used in the App.

```
function MyComponent(props) {
   return (
       <div>
           {props.data}
       </div>
   )
}
const MemoedComponent = React.memo(MyComponent)
function App() {
   const [state, setState] = useState(0)
   return (
       <>
           <button onClick={()=> setState(0)}>Click</button>
           <MemeodComponent data={state} />
       </>
   )
}
```

With this, pressing the button Click continuously will only trigger re-rendering in MyComponent once and never again. This is because React.memo would memoize its props and would return the cached output without executing the MyComponent so far as the same inputs occur over and over.
What React.PureComponent is to class components is what Reat.memo is to functional components.

useCallback()

This hook works as useMemo but the difference is that it's used to memoize function declarations.

For example having this:

```
function MyComponent(props) {
   return (
        <>
            TestComp
            <button onClick={props.func}>Set Count</button>
        </>
    )
}
 const TestComp = React.memo(MyComponent)
 function App() {
    const [count, setCount] = useState(0)
    return (
        <>
            <button onClick={()=> setCount(count + 1)}>Set Count</button>
            <MemoComponent func={()=> setCount(count + 1)} />
        </>
    )
}
```

We have an App component that maintains a count state using useState, whenever we call the setCount function the App component will re-render. It renders a button and the MyComponent component, if we click the Set Count button the App component will re-render along with its child tree. Now, the component is memoized using memo to avoid unnecessary re-renders. React.memo memoizes a component by comparing its current/next props with its previous props if they are the same it doesn't re-render the component. MyComponent component receives as prop actually a function in a 'func' props attribute, whenever the App is re-rendering, the props 'func' of MyComponent component will be checked for sameness, if found being the same it will not be re-rendered.
The problem here is that the Test component receives a new instance of the function prop. How?

Looking at the JSX:

```
...
   return (
        <>
            ...
            <Test func={()=> setCount(count + 1)} />
        </>
   )
...
```

An arrow function declaration is passed, so whenever App is rendered a new function declaration is always created with a new reference(memory address pointer). So the shallow comparison of React.memo will record a difference and will give a go-ahead for re-rendering.
This is a problem to solve! Should we move the function outside of the function scope, it will be good but it won't have reference to the setCount function. This is where useCallback comes in, we will pass the function-props to the useCallback and specify the dependency, the useCallback hook returns a memoized version of the function-prop that's what we will pass to out Test component.

```
function App() {
   const check = 90
   const [count, setCount] = useState(0)
   const onClick = useCallback(()=> { setCount(check) }, [check]);
   return (
        <>
            <button onClick={()=> setCount(count + 1)}>Set Count</button>
            <Test func={onClick} />
        </>
   )
}
```

Here, onClick will not be re-created in every re-render unless its dependency changes, so when we repeatedly click on the button Test component will not re-rendered. useCallback will check the 'check' variable if not same as its previous value it will return the function passed so Test component would see a new reference and re-render the component, if not same it would return nothing so React.memo() would see a function reference the same as its previous value and cancel the re-render.

Conclusions

In this post I have tried to cover as much as possible the ways to optimise a react app to create an amazing user experience. There is a lot more to cover and of course this depends on the user cases and can be different for a different app.
Remember that the first optimisation starts on a good pre-development plan and writing clean and maintainable code.
React is a great library and with some attention it's possible to create great applications.

Keep coding.
Riccardo Rizzo.