1. Role of keys in Virtual dom to actual dom
2. Mutating the object
3. State life cycle, SCU, Mutation of Objects
4. Use Debounce/Throttle for input params to avoid unnecessary function/api calls
5. React virtualized
6. Calculate performance with plugin
7. Bind Functions early

---

1. Keys

```
ReactDOM.render(
  <FlightContainer flightData={flights} />,
  document.getElementById(FlightContainer')
);

function FlightContainer(props){
Const flights = props.flights;
flights.map((flight) => {
<li>{flight.flightName}</li>})}
```

When you run this code, you'll be given a warning that a key should be provided for list items.

Keys help React to identify exactly which items have modified, or added, or removed. Keys should be given to the elements inside the array to give the elements a stable identity

It's better to give a unique identifier to the key, for example In the above code
You can replace LI tag with **<li key={flight.no}>{flight.flightName}
**

Avoid giving Random values in the key like <li key={Math.random()}>, What's the problem with the random is once the render happens the previous values and new value doesn't match and react couldn't identify which value is changed or added or removed. Give the values which will be stable among all the updation life cycles.

Better to use Index as last priority: It's better to avoid index **<li key={index}>** because , index will change when a particular element is deleted in an array. Once element/elements are deleted then the keys will be changed with the previous render and the latest render. So React can't identify which node is changed/modified/deleted.

---

2. Avoid Mutating the object

```
this.state = {
                        flights: [
                            'AirIndia.',
                            'Jet Flights'
                          ]
}
```

Now we decided to modify the list, If we are not enough conscious we can end mutating the state.

Mutating the object

```
let modifiedFlights = this.state.flights;
modifiedFlights.push('Indigo');
this.setState({flights:modifiedFlights});
```

As we know the arrays will be **passed by reference,** Here we are modifying the state directly, bacause the **modifiedFlights** will contain the reference to the state.

We need to avoid Mutating by creating the copy of an array → Modify the copy → then apply setState on that array

Creating a new copy to avoid Mutation

```
let modifiedFlights = [...this.state.flights];  // or // Object.assign ({}, this.state.flights);
modifiedFlights.push('Indigo');
this.setState({flights:modifiedFlights});
```

What is the advantage of Immutability:

When we create a new object, the reference will change and **ShouldComponentUpdate** function can find out there is an change in the object and return true for the updation life cycle, But when change(mutate) the existing object the reference address won't change and **ShouldComponentUpdate** couldn't find there is a change and return false and it stops the updation life cycle.

We can use any **Immutability** libraries to avoid the mutation as it helps to improve the performance by executing the component updation life cycle when requried.

---

3. ShouldComponentUpdate (Prevent Unwanted rendering):

ShouldComponentUpdate will help you stop the updation life cycle. In every SCU references are checked, If we mutate the object then this function will return false and latest updates won't be available.

```
function shouldComponentUpdate(nextProps, nextState) {
   return nextProps.id !== this.props.id;
}
```

When you want to change the data Avoid mutating the object rather create a new object.

Pure Component
If you got an requirement where you want react to check the state and props for you then go with **PureComponent.**

When you extend an component with Pure Component React will implement **ShouldComponentUpdate** for you

---

4. Using Debounce/Throttle for input Handlers

A debounce function can be used to delay certain events so that it doesn't get fired up every millisecond.

For instance, when you're typing something into the input box(for search), for every letter entered we will hit api which is waste of api calls and unnecessery rendering of response, Instead we can hit the api for regular intervals.

```
export default class SearchBar extends React.Component {
  constructor(props) {
    super(props);
    this.onChange = this.onChange.bind(this);
  }
  onChange(e) {
    this.props.onChange(e.target.value);
  }
  render () {
```

```
    return (
      <label> Search </label>
      <input onChange={this.onChange}/>
    );
  }
}
```

From the above program **onChange** function will be called for each letter change, Instead if we can use debounce we can limit the calls and can control the response rendering.

```
import {debounce} from 'throttle-debounce';

export default class Comp extends Component {
  constructor(props) {
    super(props);
    this.callAPI = debounce(500, this.callAPI);
  }
  onChange(e) {
    this.callAPI(e.target.value);
  }
  callAjax(value) {
    console.log('value :: ', value);
    // AJAX call here
  }
  render() {
    return (
      <div>
        <input type="text" onKeyUp={this.onChange.bind(this)}/>
      </div>
    );
  }
}
```

In the above code, **onChange** will be called for every letter hit but the api call won't be called all the time, It will called for each **500 milliseconds**.

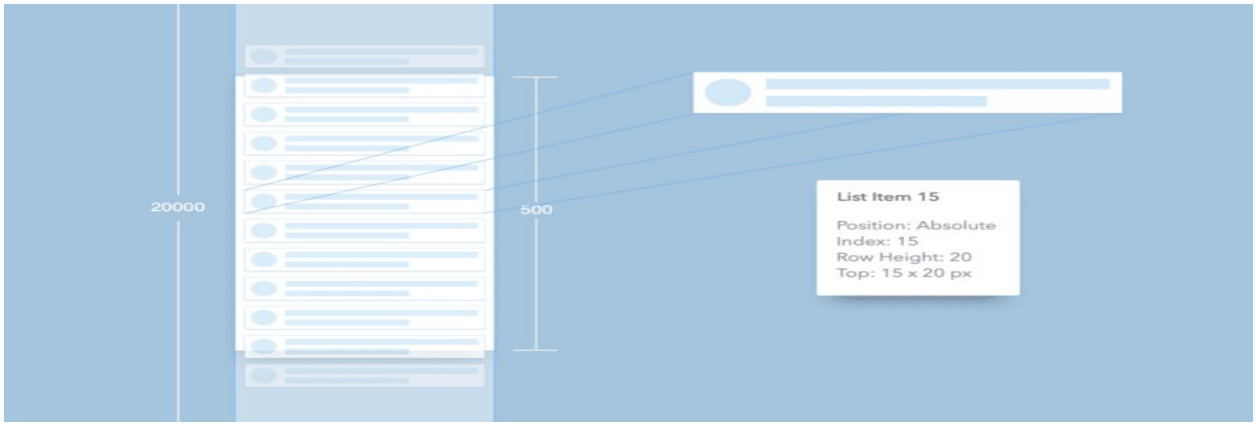
---

5. React virtualized

A virtualized list is a technique where a large list of items in a scrollable view is virtually rendered to **only show items that are visible within the the scrollable view window**. That

may sound like a handful but it basically means a virtualized list only renders items that are visible on the screen.

In other words,
Ideally react virtualized would only render the items that are visible. The items outside of the view window don't need the rendering power. Only when they come into the visible window do they need to be rendered. This is where virtualized list comes into view.



https://github.com/bvaughn/react-virtualized

---

6. **React-perf addon**

Please install `react-addons-perf` To calculate the performance of the app.

Import Perf from 'react-addons-perf'
Perf.start();
//Do some manipulations of the app
Perf.stop();
Perf.printWasted(); //will provide the wasted time(Dom was not touched but Js had exectued)
Perf.printDom(); //will print the Dom that is changed
Perf.printInclusive();// Prints overall time taken
Perf.printExclusive()//Prints overall time taken excluding ComponentWillMount and Component DId monut

---

7. Bind function early

Binding functions in the render() method is a bad idea because React will create a new function on each render.

```
class Button extends React.Component {
  handleClick() {
    console.log('Yay');
  }

  render() {
    return (
      <button onClick={() => this.handleClick()}/> // avoid this
    );
  }
}
```

Use arrow functions or bind in the constructor

```
class Button extends React.Component {
  // This is a class field
  handleClick = () => {
    console.log('Yay!');
  }

  render() {
    return (
      <button onClick={this.handleClick}/>
    );
  }
}
```

Bind in the constructor

```
class Button extends React.Component {
  constructor(props) {
    super(props);

    this.handleClick = this.handleClick.bind(this);
```

```
  }

  handleClick() {
    console.log('Yay!');
  }

  render() {
    return (
      <button onClick={this.handleClick}/>
    );
  }
}
```

Most of use face problem when you want to pass params and bind the function at the same time, then we can curry the function and avoid writing functions in render function.

```
class Button extends React.Component {
  constructor(props) {
    super(props);

    this.handleClick = (param1, param2) => (event){
          //your code goes here
      }

      handleClick() {
        console.log('Yay!');
      }

      render() {
        return (
          <button onClick={this.handleClick(param1Value,
     Param2Value)}/>
        );
      }
     }
```

Here for the first time the handleClick function gets executed and return another function which will called on onClick but still holds the param1, param2 values because the inner function is a closure here.

Using **chunk files** in webpack while loading the files on the browser.