REACT JS

Keys

Sreekanth M. E.

Freelance Trainer & Consultant

http://www.SreekanthME.com

When React SPA is loaded, the render() function creates a tree of React elements (Virtual DOM).

On the next state or props update, the render() function will create another tree of React elements.

React then runs a diffing algorithm to find out delta between the two trees.

When diffing two trees, React first compares the two root elements.

ELEMENTS OF DIFFERENT TYPES:

Whenever the root elements have different types, React will tear down the old tree and build the new tree from scratch.

ELEMENTS OF DIFFERENT TYPES → EXAMPLE:

```
<div>
  <Counter />
</div>


<span>
  <Counter />
</span>
```

After diffing the LHS root elements, React will destroy the old Counter and remount a new one.

ELEMENTS OF SAME TYPE:

When comparing two React DOM elements of the same type, React looks at the attributes of both, keeps the same underlying DOM node, and only updates the changed attributes.

ELEMENTS OF SAME TYPE → EXAMPLE:

```
<div className="before" title="stuff" >
    <Counter />
</div>

<div className="after" title="stuff" >
    <Counter />
</div>
```

By comparing these two elements, React will modify ONLY the className on the DIV node.

ELEMENTS OF THE SAME TYPE BUT WITH CHILD ELEMENTS:

When comparing two React DOM elements of the same type but different set of child nodes, the children are recursed.

React just iterates over both lists of children at the same time and updates HTML DOM when there is a difference.

SAME TYPE BUT WITH CHILD ELEMENTS → EXAMPLE-1:

```
<ul>
  <li>first</li>
  <li>second</li>
</ul>


<ul>
  <li>first</li>
  <li>second</li>
  <li>third</li>
</ul>
```

React will match the two <li>first</li> trees, match the two <li>second</li> trees, and then inserts ONLY the <li>third</li> into the tree.

SAME TYPE BUT WITH CHILD ELEMENTS → EXAMPLE-2:

```
<ul>
  <li>second</li>
  <li>third</li>
</ul>

<ul>
  <li>first</li>
  <li>second</li>
  <li>third</li>
</ul>
```

React will update every child instead of realizing it can keep the `<li>second</li>` and `<li>third</li>` subtrees intact. This will lead to update performance inefficiency.

In order to solve this issue, React supports a key attribute. When children have keys, React uses the key to match children in the original tree with children in the subsequent tree.

SAME TYPE BUT WITH CHILD ELEMENTS → EXAMPLE-3:

```
<ul>
  <li key="2015">second</li>
  <li key="2016">third</li>
</ul>

<ul>
  <li key="2014">first</li>
  <li key="2015">second</li>
  <li key="2016">third</li>
</ul>
```

Now React knows that the element with key '2014' is the new one, and the elements with the keys '2015' and '2016' have just moved. So only element with key "2014" is updated.

Key acts as an unique identifier for an element in the list and acts as a hint at which child elements may be stable across different renders.

Keys used within arrays should be unique among their siblings.

However they don't need to be globally unique. Same keys can be used with two different arrays

For a given value in the list, key must remain unchanged across renders.

Keys help React identify which items have changed / added / removed.

Math.random() cannot be used to generate keys because:

- There is a rare chance of same random value being generated more than once.
- The generated keys won't remain unchanged across renders.

Best to use a unique identifier associated with the data itself as key. For example, a primary key in the database table.

Array index number can be used ONLY if the array will remain static. i.e. its contents will remain unchanged.

As it is usually unlikely that array will remain unchanged, best to avoid using array indexes as keys.

Keys must be added to ONLY that React element (whether HTML element or Custom element) which is generated as a list (through iteration). Not to its child elements.

END OF CHAPTER

APPENDIX