

1.) Component and Element:

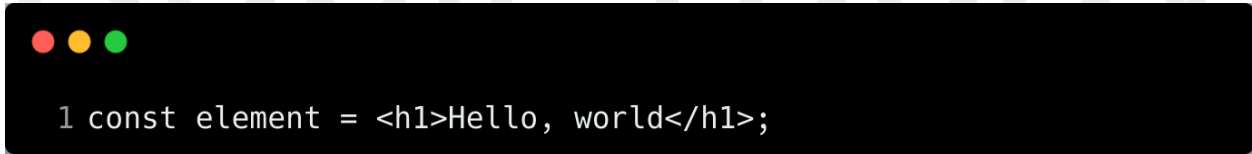
Component: class or function that outputs an element tree.
React component also takes props as an input

Element: React elements are plain JavaScript objects describing what should be rendered on the screen.

Examples in code:



```
1 import React from 'react';
2
3 // Functional Component
4 function Greeting(props) {
5   return <h1>Hello, {props.name}!</h1>;
6 }
```



```
1 const element = <h1>Hello, world</h1>;
```

React creates a tree of elements, when we call `render()` method.
The tree of elements is stored in memory, called virtual DOM.

Reconciliation refers to the process by which React updates the user interface to reflect changes in the underlying data or state. React uses a virtual DOM (a lightweight representation of the actual DOM) to efficiently determine and apply the minimum number of changes necessary to update the UI.

During reconciliation, React determines if a component needs to be updated by comparing the component's old and new props and state.

If the props or state have changed, React re-renders the component and its child components. However, React tries to optimize this process by avoiding unnecessary re-renders and only updating the components that are affected by the changes.

The reconciliation process involves comparing the previous virtual DOM tree with the new virtual DOM tree and identifying the differences between them.

The [state of the art algorithms](#) have a complexity in the order of $O(n^3)$ where n is the number of elements in the tree.

React implements a heuristic $O(n)$ algorithm based on two assumptions:

- 1.) Two elements of different types will produce different trees
- 2.) When we have a list of child elements that often change, we should provide a unique key as a prop

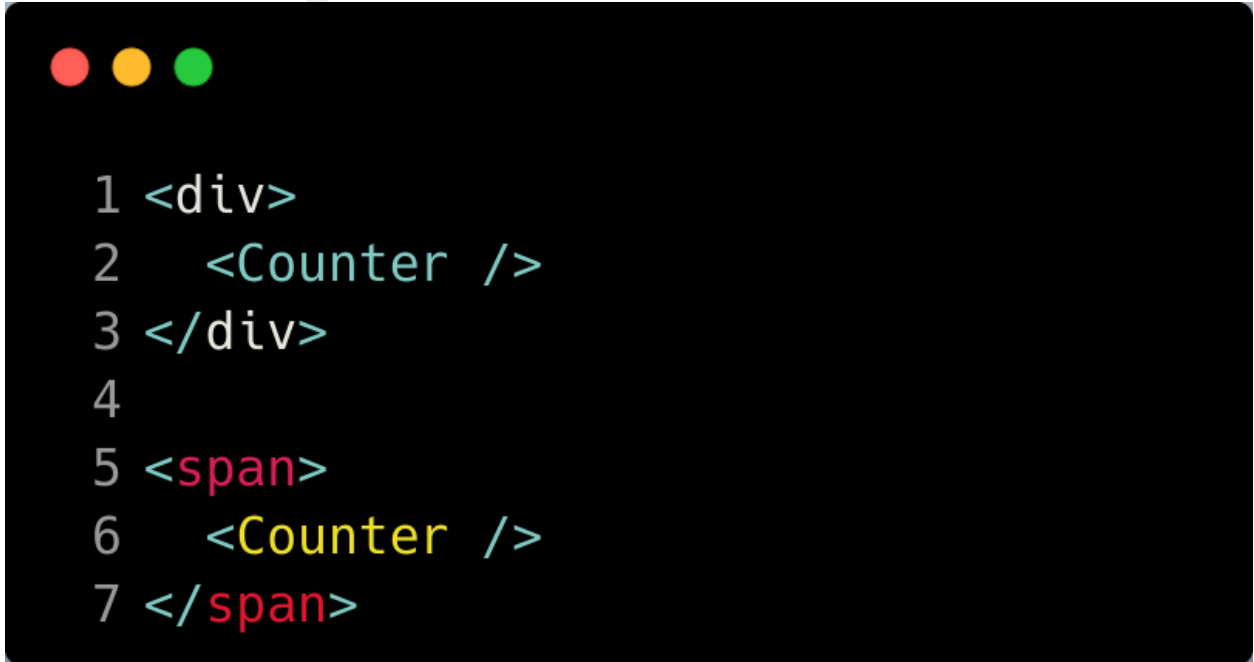
Diffing Algorithm:

When diffing two trees, React first compares the two root elements. The behavior is different depending on the types of the 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.

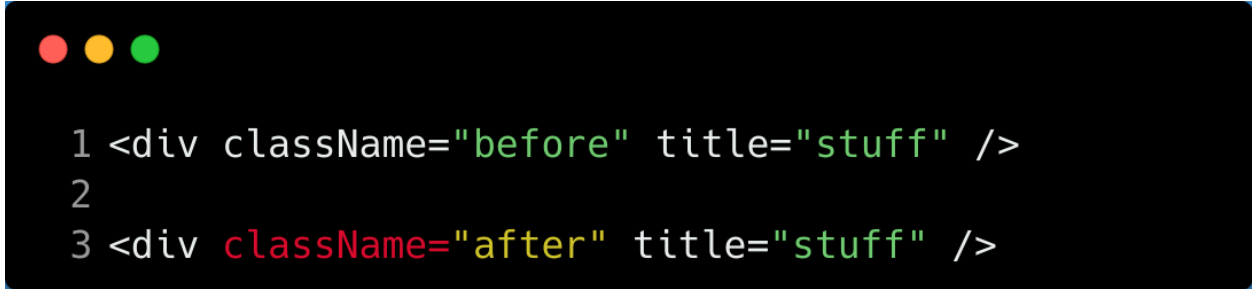
Going from `<a>` to ``, or from `<Article>` to `<Comment>`, or from `<Button>` to `<div>` - any of those will lead to a full rebuild.



```
1 <div>
2   <Counter />
3 </div>
4
5 <span>
6   <Counter />
7 </span>
```

DOM Elements Of The 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. For example:



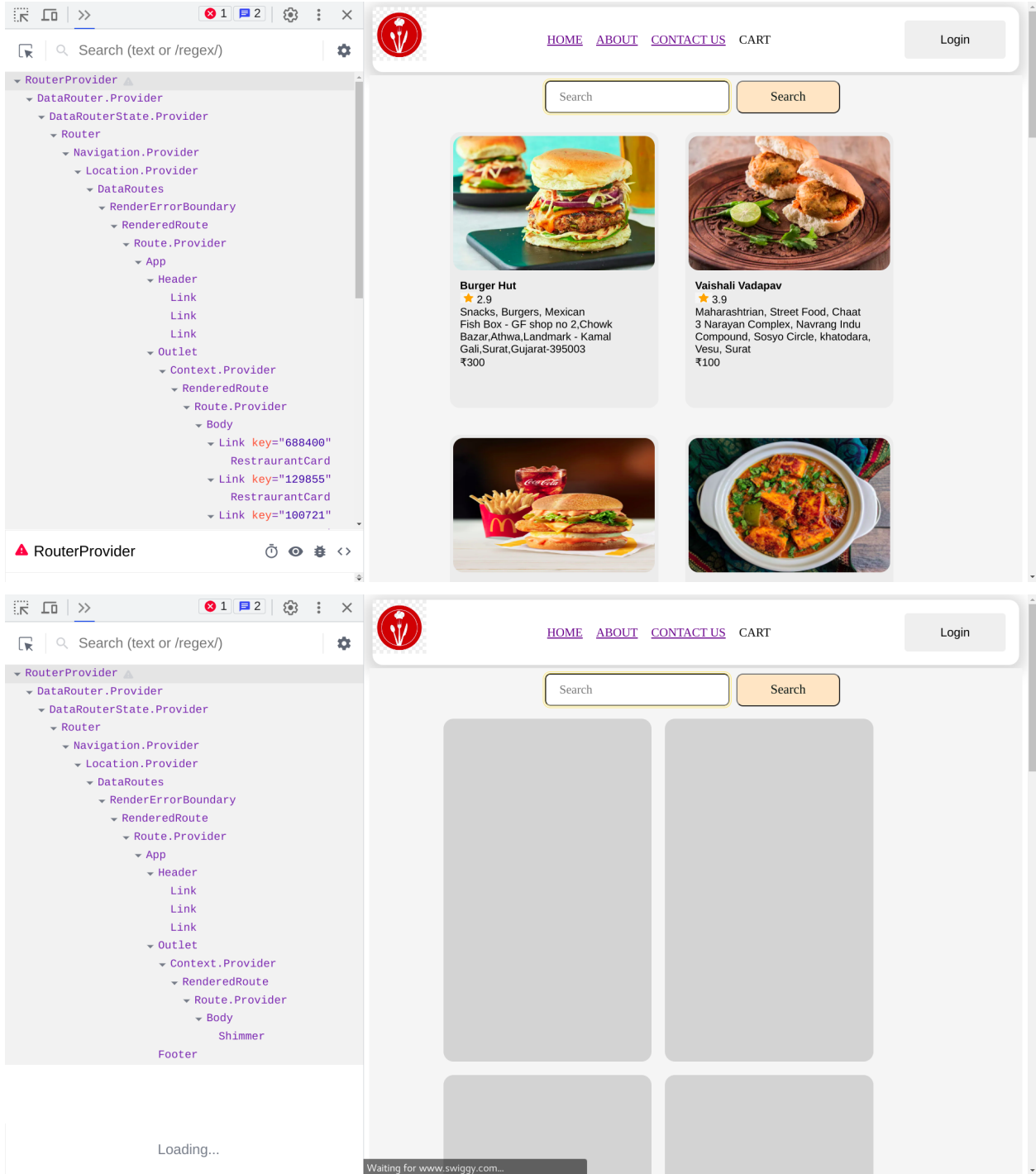
```
1 <div className="before" title="stuff" />
2
3 <div className="after" title="stuff" />
```

React's reconciliation process follows a few key principles:

1. It performs a top-down, depth-first traversal of the component tree.
2. It tries to minimize the number of updates by batching changes and applying them in a single pass.
3. It attempts to reuse existing DOM elements when possible to minimize the need for recreating elements from scratch.

To view the reconciliation process with React DevTools, you can follow these steps:

1. Install React DevTools extension for your preferred browser (e.g., Chrome, Firefox).
2. Open your React application in the browser.
3. Open the browser's developer tools (usually by right-clicking and selecting "Inspect" or by pressing Ctrl+Shift+I or Cmd+Option+I).
4. In the developer tools, you should see a tab labeled "React" or "Components" (depending on the browser). Click on it.



Why Use Key attribute in React?

Let's say we have a list of items :

```
const items = [  
  { name: 'Kapil' },  
  { name: 'Rahul' },  
  { name: 'Akshay' },  
];
```

What if we want to add one more item to the list and that too in the beginning?

The new code will Look Like this:

```
const items = [  
  { name: 'Alok' },  
  { name: 'Kapil' },  
  { name: 'Rahul' },  
  { name: 'Akshay' },  
];
```

In this updated code, an additional item { name: 'Alok' } is added to the items array without assigning a unique key.

If you implement it naively, inserting an element at the beginning has worse performance.

Converting between these two trees works poorly.

React will mutate every child instead of realizing it can keep the

```
{ name: 'Kapil' },  
{ name: 'Rahul' },  
{ name: 'Akshay' },
```

 subtrees intact. This inefficiency can be a problem.

While the code will still render the list of items without errors, it is not considered a best practice. Without keys, React may encounter difficulties when efficiently updating and reconciling the list of items.

It may lead to unexpected behavior, incorrect rendering, or performance issues, especially when items are modified, added, or removed.

To ensure optimal performance and proper reconciliation, it is recommended to assign unique and stable keys to each component in the list, even if an item is added or removed.

Keys

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. For example, adding a key to our inefficient example above can make the tree conversion efficient:

```
1  const items = [  
2    { id: 1, name: 'Item 1' },  
3    { id: 2, name: 'Item 2' },  
4    { id: 3, name: 'Item 3' },  
5  ];  
6  
7  const itemList = items.map((item) => (  
8    <Item key={item.id} name={item.name} />  
9  ));  
10  
11  function Item({ key, name }) {  
12    return <div key={key}>{name}</div>;  
13  }  
14
```

```

1 import React, { useState } from 'react';
2 import { v4 as uuidv4 } from 'uuid';
3
4 const ItemList = () => {
5   const [items, setItems] = useState([
6     { id: uuidv4(), name: 'Item 1' },
7     { id: uuidv4(), name: 'Item 2' },
8     { id: uuidv4(), name: 'Item 3' },
9   ]);
10
11   const addItem = () => {
12     const newItem = {
13       id: uuidv4(),
14       name: `Item ${items.length + 1}`,
15     };
16     setItems([...items, newItem]);
17   };
18
19   const removeItem = (id) => {
20     setItems(items.filter((item) => item.id !== id));
21   };
22
23   return (
24     <div>
25       <button onClick={addItem}>Add Item</button>
26       <ul>
27         {items.map((item) => (
28           <li key={item.id}>
29             {item.name}
30             <button onClick={() => removeItem(item.id)}>Remove</button>
31           </li>
32         ))}
33       </ul>
34     </div>
35   );
36 };
37
38 export default ItemList;
39

```

NOTE: Key Should be Unique.

Can we use Index as a Key??

It is generally not recommended to use the index as a key, for several reasons:

1. React may not be able to distinguish between items if they are reordered or new items are inserted/deleted. This can result in unexpected behavior and incorrect rendering.
2. If items are removed or added in the middle of the list, the indexes of subsequent items will change, potentially causing unnecessary re-rendering of unrelated components.

We can use the index as a key when our array is static and the items do not have any unique identifier.

Should we use Math.random() as a key??

We should not use Math.random() as unique key, because:

1. **Key Collisions:** It Does not guarantee unique Keys. Key collisions can lead to incorrect rendering and behavior issues.
2. **Lack of stability:** Keys in React should be stable across re-renders. When components are re-rendered, React uses keys to determine which components should be updated, added, or removed. Using Math.random() will generate different keys on each render, resulting in unnecessary re-rendering of components and potentially impacting performance.
3. **Diffing and reconciliation:** React's diffing algorithm relies on keys to efficiently update the virtual DOM and reconcile component changes. If keys are not stable or unique, React may not be able to accurately determine the component identity and may end up re-rendering more components than necessary.

NOTE: It is recommended to use stable and unique identifiers as keys. This could be an ID from your data source or a unique property of the item being rendered.

React Fiber

React Fiber is an internal reimplementation of the React reconciliation algorithm and rendering pipeline.

It was introduced in React version 16 as a way to improve performance and enable better handling of concurrent and asynchronous updates.

1. **Incremental rendering:**

React Fiber enables incremental rendering, where the rendering work is divided into smaller units and interleaved with other tasks. This allows React to respond more quickly to user interactions and provide a smoother user experience.

2. **Prioritization and scheduling:**

React Fiber introduced the concept of priority levels, which allows React to prioritize and schedule work based on its importance and urgency. It ensures that high-priority updates (such as user interactions) are processed and rendered with minimal delay.

3. **Time-slicing:**

Time-slicing is a technique enabled by React Fiber that allows the rendering work to be spread across multiple frames, preventing long-running tasks from blocking the main thread and improving the overall responsiveness of the application.

4. **Error boundaries:**

React Fiber introduced error boundaries, which are components that can catch and handle errors that occur during the rendering or lifecycle methods of their children. This helps prevent entire components or application crashes due to errors in a single component.

References :

1. <https://legacy.reactjs.org/docs/reconciliation.html>
2. <https://www.youtube.com/watch?v=7YhdqIR2Yzo>
3. Akshay Saini - Namaste React