**The DOM?**

The browser builds the DOM by parsing the code you write, it does this before it renders the page. The DOM represents documents in the page as nodes and objects, providing an interface so that programming languages can plug in and manipulate the DOM. The problem with the DOM is that it is not optimized for dynamic UI applications. So, updating the DOM can slow your application when there are a lot of things to be changed; as the browser has to reapply all styles and render new HTML elements. This also happens in situations where nothing changes.

What is Reconciliation?

Reconciliation is the process through which React updates the DOM. When a component’s state changes, React has to calculate if it is necessary to update the DOM. It does this by creating a virtual DOM and comparing it with the current DOM. In this context, the virtual DOM will contain the new state of the component.

Let’s build a simple component that adds two numbers. The numbers will be entered in an input field.

First, we’ll need to set up the initial state for the fields, then update the state when a number is entered. The component will look like this:

class App extends React.Component {

state = {

result: '',

entry1: '',

entry2: ''

}

handleEntry1 = (event) => {

this.setState({entry1: event.target.value})

}

handleEntry2 = (event) => {

this.setState({entry2: event.target.value})

}

handleAddition = (event) => {

const firstInt = parseInt(this.state.entry1)

const secondInt = parseInt(this.state.entry2)

this.setState({result: firstInt + secondInt })

}

render() {

const { entry1, entry2, result } = this.state

return(

<div>

<div>

<p>Entry 1: { entry1 }</p>

<p>Entry 2: { entry2 }</p>

<p>Result: { result }</p>

</div>

<br />

<div>

<span>Entry 1: </span>

<input type='text' onChange={this.handleEntry1} />

</div>

<br />

<div>

<span>Entry 2: </span>

<input type='text' onChange={this.handleEntry2} />

</div>

<div>

<button onClick={this.handleAddition} type='submit'>Add</button>

</div>

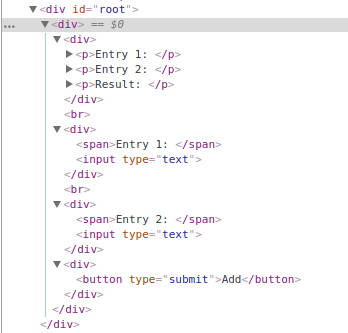
</div>

)

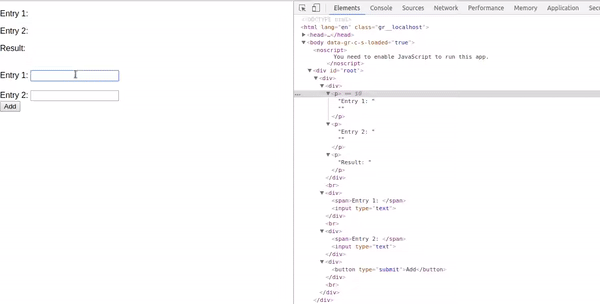
}

}

On initial render, the DOM tree will look like this;



When an entry is made in the first input field, React creates a new tree. The new tree which is the virtual DOM will contain the new state for entry1. Then, React compares the virtual DOM with the old DOM and, from the comparison, it figures out the difference between both DOMs and makes an update to only the part that is different. A new tree is created each time the state of App component changes — when a value is entered in either of the inputs field, or when the button is clicked.



**Diffing Different Elements**

When the state of a component changes so that an element needs to be changed from one type to another, React unmounts the whole tree and builds a new one from scratch. This causes every node in that tree to be destroyed.

Let’s see an example:

class App extends React.Component {

state = {

change: true

}

handleChange = (event) => {

this.setState({change: !this.state.change})

}

render() {

const { change } = this.state

return(

<div>

<div>

<button onClick={this.handleChange}>Change</button>

</div>

{

change ?

<div>

This is div cause it's true

<h2>This is a h2 element in the div</h2>

</div> :

<p>

This is a p element cause it's false

<br />

<span>This is another paragraph in the false paragraph</span>

</p>

}

</div>

)

}

}

On initial render, you will see the div and its contents and how clicking the button causes React to destroy the div’s tree with its content and build a tree for the <p> element instead. Same happens if we have the same component in both cases. The component will be destroyed alongside the previous tree it belonged to, and a new instance will be built. See the demo below;

**Diffing Lists**

React uses keys to keep track of items in a list. The keys help it figure out the position of the item on a list. What happens when a list does not have keys? React will mutate every child of the list even if there are no new changes.

In other words, React changes every item in a list that does not have keys.

Here’s an example:

const firstArr = ['codepen', 'codesandbox']

const secondArr = ['github', 'codepen', 'bitbucket', 'codesanbox']

class App extends React.Component {

state = {

change: true

}

handleChange = (event) => {

this.setState({change: !this.state.change})

}

render() {

const { change } = this.state

return(

<div>

<div>

<button onClick={this.handleChange}>Change</button>

</div>

<ul>

{

change ?

firstArr.map((e) => <li>{e}</li>)

:

secondArr.map((e) => <li>{e}</li>)

}

</ul>

</div>

)

}

}

Here, we have two arrays that get rendered depending on the state of the component. React has no way of keep track of the items on the list, so it is bound to change the whole list each time there is a need to re-render. This results in performance issues.

In your console, you will see a warning like this:

Warning: Each child in an array or iterator should have a unique "key" prop.

To fix this, you add a unique key for each item on the list. The best solution in this scenario is to create an array of objects, with each item having a unique id. If we make use of the array index, that will be an antipattern, that will come back to hurt us.

const firstArr = [

{ id: 1, name: 'codepen'},

{ id: 2, name: 'codesandbox'}

]

const secondArr = [

{ id: 1, name: 'github'},

{ id: 2, name: 'codepen'},

{ id: 3, name: 'bitbucket'},

{ id: 4, name: 'codesandbox'}

]

class App extends React.Component {

state = {

change: true

}

handleChange = (event) => {

this.setState({change: !this.state.change})

}

render() {

const { change } = this.state

return(

<div>

<div>

<button onClick={this.handleChange}>Change</button>

</div>

<ul>

{

change ?

firstArr.map((e) => <li key={e.id}>{e.name}</li>)

:

secondArr.map((e) => <li key={e.id}>{e.name}</li>)

}

</ul>

</div>

)

}

}

**Wrapping Up**

In summary, here are the two big takeaways for understanding how the concept of reconciliation works in React:

* React can make your UI fast, but it needs your help.
* React doesn’t do a full rerender of your DOM nodes. It only changes what it needs to. The diffing process is so fast that you might not notice it.