deathmood

Follow 777 Followers About

How to write your own Virtual DOM

deathmood Jun 1, 2016 · 8 min read

There are two things you need to know to build your own Virtual DOM. You do not even need to dive into React's source. Or into source code of any other Virtual DOM implementations. They are so large and complex — but in reality the main part of Virtual DOM can be written in less than ~50 lines of code. 50. Lines. Of. Code. !!!

Here are these two concepts:

- Virtual DOM is any kind of representation of a real DOM
- When we change something in our Virtual DOM Tree, we get a new Virtual Tree. Algorithm compares these two trees (old and new), finds differences and makes only necessary small changes to real DOM so it reflects virtual

That's all! Let's dive deeper into each of these concepts.

Well, first we need to store somehow our DOM tree in memory. And we can do that with plain old JS objects. Suppose we have this tree:

```
<ul class=”list”>
  <li>item 1</li>
  <li>item 2</li>
</ul>
```

Looks pretty simple, yeah? How could we represent that with just JS objects?

```
{ type: ‘ul’, props: { ‘class’: ‘list’ }, children: [
  { type: ‘li’, props: {}, children: [‘item 1’] },
  { type: ‘li’, props: {}, children: [‘item 2’] }
] }
```

Here you can notice two things:

- We represent DOM elements with objects like

```
{ type: ‘…’, props: { … }, children: [ … ] }
```

- We represent DOM text nodes with plain JS strings

But writing big trees in such way is quite difficult. So let’s write a helper function, so it will be easier for us to understand structure:

```
function h(type, props, …children) {
  return { type, props, children };
}
```

Now we can write our DOM tree like this:

```
h(‘ul’, { ‘class’: ‘list’ },
  h(‘li’, {}, ‘item 1’),
  h(‘li’, {}, ‘item 2’),
);
```

you? Yes, I want it here too. So how does it work?

If you read official Babel JSX documentation here , you'll know, that Babel transpiles this code:

```
<ul className=”list”>
  <li>item 1</li>
  <li>item 2</li>
</ul>
```

Into smth like this:

```
React.createElement(‘ul’, { className: ‘list’ },
  React.createElement(‘li’, {}, ‘item 1’),
  React.createElement(‘li’, {}, ‘item 2’),
);
```

Notice any similarities? Yes, yes.. If we could just replace those ***React.createElement(…)'***s with our ***h(…)'***s calls… It turns out we can — by using smth called ***jsx pragma***. We just need to include comment-like line at the top of our source file:

```
/** @jsx h */

<ul className=”list”>
  <li>item 1</li>
  <li>item 2</li>
</ul>
```

Well, it actually tells Babel 'hey, transpile that jsx but instead of ***React.createElement***, put ***h***'. You can put anything instead of `h` there. And that will be transpiled.

So, to sum up what I've said before, we will write our DOM in such way:

```
/** @jsx h */

const a = (
  <ul className=”list”>
);
```

And that will be transpiled by Babel to this code:

```
const a = (
  h(‘ul’, { className: ‘list’ },
    h(‘li’, {}, ‘item 1’),
    h(‘li’, {}, ‘item 2’),
  );
);
```

When function `h` executes, it will return plain JS objects — our Virtual DOM representation:

```
const a = (
  { type: ‘ul’, props: { className: ‘list’ }, children: [
    { type: ‘li’, props: {}, children: [‘item 1’] },
    { type: ‘li’, props: {}, children: [‘item 2’] }
  ] }
);
```

Go ahead and try that in JSFiddle (don’t forget to set Babel as your language):

Edit in JSFiddle

- Babel + JSX
- HTML
- CSS
- Result
- Resources

```
/** @jsx h */

function h(type, props, ...children) {
  return { type, props, children };
}

const a = (
  <ul class="list">
    <li>item 1</li>
    <li>item 2</li>
  </ul>
);
```

Applying our DOM Representation

Ok, now we have our DOM tree represented as plain JS objects, with our own structure. That's cool, but we need to somehow create a real DOM from it. 'Cause we can't just append our representation into DOM.

First let's make some assumptions and set up terminology:

- I will write all variables with real DOM nodes (elements, text nodes) starting with `$` — so ***$parent*** will be real DOM element
- Virtual DOM representation will be in variable named ***node***
- Like in React, you can have only ***one root node*** — all other nodes will be inside

Ok, having that said, let us write a function ***createElement(…)*** that will take a virtual DOM node and return a real DOM node. Forget about `props` and `children` for now — we'll set up that later:

```
function createElement(node) {
  if (typeof node === 'string') {
    return document.createTextNode(node);
  }
  return document.createElement(node.type);
}
```

So, because we can have both ***text nodes*** — that are plain JS strings and ***elements*** — that are JS objects of type like:

```
{ type: '…', props: { … }, children: [ … ] }
```

Thus, we can pass here both virtual text nodes and virtual element nodes — and that will work.

Now let's think about children — each of them is also either a text node or an element. So they can also be created with our ***createElement(…)*** function. Yeah, do you feel that? It feels recursively :)) So we can call ***createElement(…)*** for each of element's children and then ***appendChild()*** them into our element like this:

```
    }
    const $el = document.createElement(node.type);
    node.children
      .map(createElement)
      .forEach($el.appendChild.bind($el));
    return $el;
  }
```

Wow, that looks nice. Let's put aside node ***props*** for now. We'll talk about them later. We do not need them for understanding basic concepts of Virtual DOM but they'll add more complexity.

Now go ahead and try that in JSFiddle:

Edit in JSFiddle

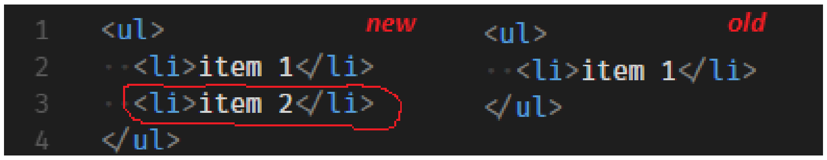
- Babel + JSX
- HTML
- CSS
- Result

```
/** @jsx h */

function h(type, props, ...children) {
  return { type, props, children };
}

function createElement(node) {
  if (typeof node === 'string') {
    return document.createTextNode(node);
  }
  const $el = document.createElement(node.type);
  node.children
    .map(createElement)
    .forEach($el.appendChild.bind($el));
  return $el;
}

const a = (
```

Handling changes

Ok, now that we can turn our virtual DOM into a real DOM, it's time to think about diffing our virtual trees. So basically we need to write an algo, that will compare two virtual trees — old and new — and make only necessary changes into real DOM.

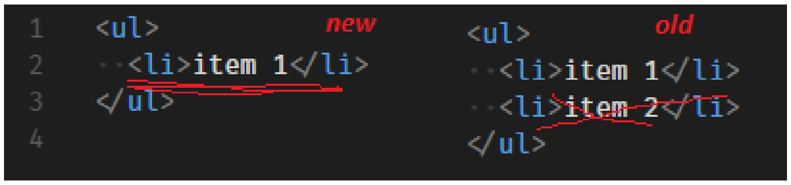
How to diff trees? Well, we need to handle next cases:



```
<ul>
  <li>item 1</li>
  <li>item 2</li>
</ul>
```

```
<ul>
  <li>item 1</li>
</ul>
```

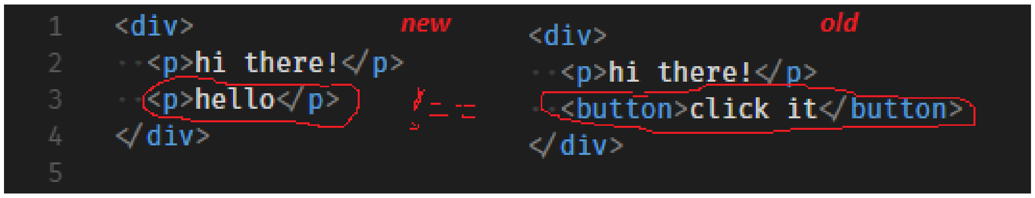
- There is no new node at some place — thus node was deleted and we need to ***removeChild(…)***



```
<ul>
  <li>item 1</li>
</ul>
```

```
<ul>
  <li>item 1</li>
  <li>item 2</li>
</ul>
```

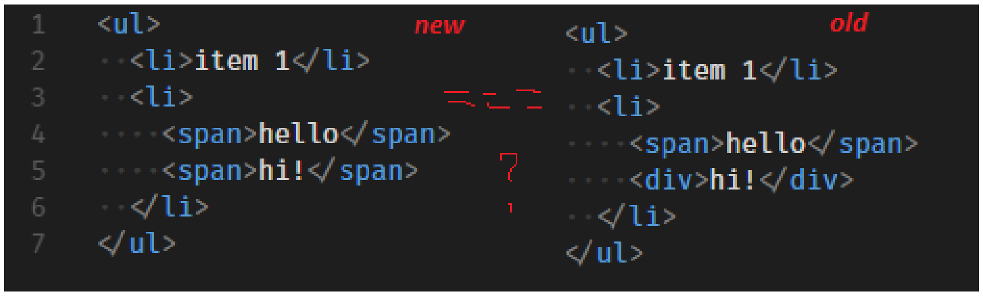
- There is a different node at that place — thus node changed and we need to ***replaceChild(…)***



```
<div>
  <p>hi there!</p>
  <p>hello</p>
</div>
```

```
<div>
  <p>hi there!</p>
  <button>click it</button>
</div>
```

- Nodes are the same — so we need to go deeper and diff child nodes



```
<ul>
  <li>item 1</li>
  <li>
    <span>hello</span>
    <span>hi!</span>
  </li>
</ul>
```

```
<ul>
  <li>item 1</li>
  <li>
    <span>hello</span>
    <div>hi!</div>
  </li>
</ul>
```

Ok, let us write a function called ***updateElement(…)*** that takes three parameters — ***$parent***, ***newNode*** and ***oldNode,*** where ***$parent*** is a real DOM element-parent of our virtual node. Now we'll see how to handle all cases that described above.

There is no old node

Well, it is pretty straightforward here, I won't even comment:

```
    $parent.appendChild(
      createElement(newNode)
    );
  }
}
```

There is no new node

Here we've got a problem — if there is no node at current place in new virtual tree — we should remove it from a real DOM — but how should we do that? Yeah, we know parent element (it is passed to function) and thus we are supposed to call ***$parent.removeChild(…)*** and pass real DOM element reference there. But we do not have that. Well, if we knew position of our node in parent, we could get its reference with ***$parent.childNodes[index],*** where ***index*** is position of our node in parent element.

Ok let's suppose that this ***index*** will be passed to our function (and it really will be passed — you'll see it later). So our code will be:

```
function updateElement($parent, newNode, oldNode, index = 0) {
  if (!oldNode) {
    $parent.appendChild(
      createElement(newNode)
    );
  } else if (!newNode) {
    $parent.removeChild(
      $parent.childNodes[index]
    );
  }
}
```

Node changed

First we need to write a function that will compare two nodes (old and new) and tell us if node really changed. We should consider that it can be both elements and text nodes:

```
function changed(node1, node2) {
  return typeof node1 !== typeof node2 ||
         typeof node1 === ‘string’ && node1 !== node2 ||
         node1.type !== node2.type
}
```

```
function updateElement($parent, newNode, oldNode, index = 0) {
  if (!oldNode) {
    $parent.appendChild(
      createElement(newNode)
    );
  } else if (!newNode) {
    $parent.removeChild(
      $parent.childNodes[index]
    );
  } else if (changed(newNode, oldNode)) {
    $parent.replaceChild(
      createElement(newNode),
      $parent.childNodes[index]
    );
  }
}
```

Diff children

And last, but not least — we should go through every child of both nodes and compare them — actually call ***updateElement(…)*** for each of them. Yeah, recursion again.

But there are some things to consider here before writing the code:

- We should compare children only if node is an element (text nodes can not have children)
- Now we pass reference to ***current node*** as parent
- We should compare all children one by one — even if at some point we will have \`undefined\` — it is ok — our function can handle that
- And finally ***index*** — it is just index of child node in \`children\` array

```
function updateElement($parent, newNode, oldNode, index = 0) {
  if (!oldNode) {
    $parent.appendChild(
      createElement(newNode)
    );
  } else if (!newNode) {
    $parent.removeChild(
      $parent.childNodes[index]
    );
  } else if (changed(newNode, oldNode)) {
    $parent.replaceChild(
      createElement(newNode),
      $parent.childNodes[index]
    const oldLength = oldNode.children.length;
    for (let i = 0; i < newLength || i < oldLength; i++) {
      updateElement(
        $parent.childNodes[index],
        newNode.children[i],
        oldNode.children[i],
        i
      );
    }
  }
}
```

Put That All Together

Yeah, this is it. We’re there. I’ve put all the code into JSFiddle and the implementation part took really 50 LOC — as I promised you. Go ahead and play with it.

Open up Developer Tools and watch changes applied when you press `Reload` button.

Conclusion

Congratulations! We've done that. We've written our Virtual DOM implementation. And it works. I hope that, having read this article, you understood basic concepts of how Virtual DOM should work and how React works under the hood.

However there are some things that were not highlighted here (I will try to cover them in future articles):

- Setting element attributes (props) and diffing/updating them
- Handling events — adding event listeners to our elements
- Making our Virtual DOM work with components, like React
- Getting references to real DOM nodes
- Using Virtual DOM with libraries that directly mutate real DOM — like jQuery and its plugins
- and even more…

P.S.

If there any errors in code or article, or if there are any optimizations I can do to this code — feel free to express them in comments :)) And sorry for my English :)

> *UPDATE: The second article about setting props & events in Virtual DOM is here*

JavaScript React Front End Development

Open in app

kmcpkllgbjfemcglcedbjmhnokmaikin