Rodrigo Pombo

November 13, 2019

Build your own React

We are going to rewrite React from scratch. Step by step. Following the architecture from the real React code but without all the optimizations and non-essential features.

If you've read any of my previous "build your own React" posts, the difference is that this post is based on React 16.8, so we can now use hooks and drop all the code related to classes.

You can find the history with the old blog posts and the code on the Didact repo. There's also a talk covering the same content. But this is a self-contained post.

Starting from scratch, these are all the things we'll add to our version of React one by one:

- **Step I**: The `createElement` Function
- **Step II**: The `render` Function
- **Step III**: Concurrent Mode
- **Step IV**: Fibers

- **Step V**: Render and Commit Phases
- **Step VI**: Reconciliation
- **Step VII**: Function Components
- **Step VIII**: Hooks

```
const element = <h1 title="foo">Hello</h1>


const container = document.getElementById("root")
ReactDOM.render(element, container)
```

Step Zero: Review

But first let's review some basic concepts. You can skip this step if you already have a good idea of how React, JSX and DOM elements work.

We'll use this React app, just three lines of code. The first one defines a React element. The next one gets

a node from the DOM. The last one renders the React element into the container.

Let's remove all the React specific code and replace it with vanilla JavaScript.

On the first line we have the element, defined with JSX. It isn't even valid JavaScript, so in order to replace it with vanilla JS, first we need to replace it with valid JS.

JSX is transformed to JS by build tools like Babel. The transformation is usually simple: replace the code inside the tags with a call to `createElement`, passing the tag name, the props and the children as parameters.

`React.createElement` creates an object from its arguments. Besides some validations, that's all it does. So we can safely replace the function call with its output.

And this is what an element is, an object with two properties: `type` and `props` (well, it has more, but we only care about these two).

The `type` is a string that specifies the type of the DOM node we want to create, it's the `tagName` you pass to `document.createElement` when you want to create an HTML element. It can also be a function, but we'll leave that for Step VII.

`props` is another object, it has all the keys and values from the JSX attributes. It also has a special property: `children`.

`children` in this case is a string, but it's usually an array with more elements. That's why elements are also trees.

The other piece of React code we need to replace is the call to `ReactDOM.render`.

`render` is where React changes the DOM, so let's do the updates ourselves.

First we create a node* using the element `type`, in this case `h1`.

Then we assign all the element `props` to that node. Here it's just the title.

** To avoid confusion, I'll use "element" to refer to React elements and "node" for DOM elements.*

Then we create the nodes for the children. We only have a string as a child so we create a text node.

Using `textNode` instead of setting `innerText` will allow us to treat all elements in the same way later. Note also how we set the `nodeValue` like we did it with the `h1` title, it's almost as if the string had `props: {nodeValue: "hello"}`.

Finally, we append the `textNode` to the `h1` and the `h1` to the `container`.

And now we have the same app as before, but without using React.

```
const element = (
  <div id="foo">
    <a>bar</a>
    <b />
  </div>
)
```

Step I: The `createElement` Function

Let's start again with another app. This time we'll replace React code with our own version of React.

We'll start by writing our own `createElement`.

Let's transform the JSX to JS so we can see the `createElement` calls.

As we saw in the previous step, an element is an object with `type` and `props`. The only thing that our function needs to do is create that object.

We use the *spread operator* for the `props` and the *rest parameter syntax* for the `children`, this way the `children` prop will always be an array.

For example, `createElement("div")` returns:

```
{
  "type": "div",
  "props": { "children": [] }
}
```

`createElement("div", null, a)` returns:

```
{
  "type": "div",
  "props": { "children": [a] }
}
```

and `createElement("div", null, a, b)` returns:

```
{
  "type": "div",
  "props": { "children": [a, b] }
}
```

The `children` array could also contain primitive values like strings or numbers. So we'll wrap everything that isn't an object inside its own element and create a special type for them: `TEXT_ELEMENT`.

React doesn't wrap primitive values or create empty arrays when there aren't `children`, but we do it because it will simplify our code, and for our library we prefer simple code than performant code.

We are still using React's `createElement`.

In order to replace it, let's give a name to our library. We need a name that sounds like React but also hints its *didactic* purpose.

We'll call it Didact.

But we still want to use JSX here. How do we tell babel to use Didact's `createElement` instead of React's?

If we have a comment like this one, when babel transpiles the JSX it will use the function we define.

Step II: The `render` Function

Next, we need to write our version of the `ReactDOM.render` function.

For now, we only care about adding stuff to the DOM. We'll handle updating and deleting later.

We start by creating the DOM node using the element type, and then append the new node to the container.

We recursively do the same for each child.

We also need to handle text elements, if the element type is `TEXT_ELEMENT` we create a text node instead of a regular node.

The last thing we need to do here is assign the element props to the node.

And that's it. We now have a library that can render JSX to the DOM.

Give it a try on codesandbox.

Step III: Concurrent Mode

But... before we start adding more code we need a refactor.

```
function createElement(type, props, ...children) {
  return {

    type,

    props: {

      ...props,
```

```
children: children.map(child =>
```

There’s a problem with this recursive call.

Once we start rendering, we won’t stop until we have rendered the complete element tree. If the element tree is big, it may block the main thread for too long. And if the browser needs to do high priority stuff like handling user input or keeping an animation smooth, it will have to wait until the render finishes.

So we are going to break the work into small units, and after we finish each unit we’ll let the browser interrupt the rendering if there’s anything else that needs to be done.

We use `requestIdleCallback` to make a loop. You can think of `requestIdleCallback` as a `setTimeout`, but instead of us telling it when to run, the browser will run the callback when the main thread is idle.

React doesn't use `requestIdleCallback` anymore. Now it uses the scheduler package. But for this use case it's conceptually the same.

`requestIdleCallback` also gives us a deadline parameter. We can use it to check how much time we have until the browser needs to take control again.

As of November 2019, Concurrent Mode isn't stable in React yet. The stable version of the loop looks more like this:

```
while (nextUnitOfWork) {
  nextUnitOfWork = performUnitOfWork(
    nextUnitOfWork
  )
}
```

To start using the loop we'll need to set the first unit of work, and then write a `performUnitOfWork` function that not only performs the work but also returns the next unit of work.

Step IV: Fibers

To organize the units of work we'll need a data structure: a fiber tree.

We'll have one fiber for each element and each fiber will be a unit of work.

Let me show you with an example.

Suppose we want to render an element tree like this one:

```
Didact.render(
  <div>
    <h1>
      <p />
      <a />
    </h1>
    <h2 />
  </div>,
  container
)
```

In the `render` we'll create the root fiber and set it as the `nextUnitOfWork`. The rest of the work will happen on the `performUnitOfWork` function, there we will do three things for each fiber:

1. add the element to the DOM
2. create the fibers for the element's children
3. select the next unit of work

One of the goals of this data structure is to make it easy to find the next unit of work. That's why each fiber has a link to its first child, its next sibling and its parent.

When we finish performing work on a fiber, if it has a `child` that fiber will be the next unit of work.

From our example, when we finish working on the `div` fiber the next unit of work will be the `h1` fiber.

If the fiber doesn't have a `child`, we use the `sibling` as the next unit of work.

For example, the `p` fiber doesn't have a `child` so we move to the `a` fiber after finishing it.

And if the fiber doesn't have a `child` nor a `sibling` we go to the "uncle": the `sibling` of the `parent`. Like `a` and `h2` fibers from the example.

Also, if the `parent` doesn't have a `sibling`, we keep going up through the `parent`s until we find one with a `sibling` or until we reach the root. If we have reached the root, it means we have finished performing all the work for this `render`.

Now let's put it into code.

```
function createElement(type, props, ...children) {
  return {

    type,

    props: {

      ...props,

      children: children.map(child =>
```

First, let's remove this code from the `render` function.

We keep the part that creates a DOM node in its own function, we are going to use it later.

In the `render` function we set `nextUnitOfWork` to the root of the fiber tree.

Then, when the browser is ready,it will call our `workLoop` and we'll start working on the root.

First, we create a new node and append it to the DOM.

We keep track of the DOM node in the `fiber.dom` property.

Then for each child we create a new fiber.

And we add it to the fiber tree setting it either as a child or as a sibling, depending on whether it's the first child or not.

Finally we search for the next unit of work. We first try with the child, then with the sibling, then with the uncle, and so on.

And that's our `performUnitOfWork`.

Step V: Render and Commit Phases

We have another problem here.

We are adding a new node to the DOM each time we work on an element. And, remember, the browser could interrupt our work before we finish rendering the whole tree. In that case, the user will see an incomplete UI. And we don't want that.

So we need to remove the part that mutates the DOM from here.

Instead, we'll keep track of the root of the fiber tree. We call it the work in progress root or `wipRoot`.

And once we finish all the work (we know it because there isn't a next unit of work) we commit the whole fiber tree to the DOM.

We do it in the `commitRoot` function. Here we recursively append all the nodes to the dom.

Step VI: Reconciliation

So far we only *added* stuff to the DOM, but what about updating or deleting nodes?

That's what we are going to do now, we need to compare the elements we receive on the `render` function to the last fiber tree we committed to the DOM.

So we need to save a reference to that "last fiber tree we committed to the DOM" after we finish the commit. We call it `currentRoot`.

We also add the `alternate` property to every fiber. This property is a link to the old fiber, the fiber that we committed to the DOM in the previous commit phase.

Now let's extract the code from `performUnitOfWork` that creates the new fibers...

...to a new `reconcileChildren` function.

Here we will reconcile the old fibers with the new elements.

We iterate at the same time over the children of the old fiber (`wipFiber.alternate`) and the array of elements we want to reconcile.

If we ignore all the boilerplate needed to iterate over an array and a linked list at the same time, we are left with what matters most inside this while: `oldFiber` and `element`. **The `element` is the thing we want to render to the DOM and the `oldFiber` is what we rendered the last time.**

We need to compare them to see if there's any change we need to apply to the DOM.

To compare them we use the type:

- if the old fiber and the new element have the same type, we can keep the DOM node and just update it with the new props
- if the type is different and there is a new element, it means we need to create a new DOM node
- and if the types are different and there is an old fiber, we need to remove the old node

Here React also uses keys, that makes a better reconciliation. For example, it detects when children change places in the element array.

When the old fiber and the element have the same type, we create a new fiber keeping the DOM node from the old fiber and the props from the element.

We also add a new property to the fiber: the `effectTag`. We'll use this property later, during the commit phase.

Then for the case where the element needs a new DOM node we tag the new fiber with the `PLACEMENT` effect tag.

And for the case where we need to delete the node, we don't have a new fiber so we add the effect tag to the old fiber.

But when we commit the fiber tree to the DOM we do it from the work in progress root, which doesn't have the old fibers.

So we need an array to keep track of the nodes we want to remove.

And then, when we are commiting the changes to the DOM, we also use the fibers from that array.

Now, let's change the `commitWork` function to handle the new `effectTags`.

If the fiber has a `PLACEMENT` effect tag we do the same as before, append the DOM node to the node

from the parent fiber.

If it's a `DELETION`, we do the opposite, remove the child.

And if it's an `UPDATE`, we need to update the existing DOM node with the props that changed.

We'll do it in this `updateDom` function.

We compare the props from the old fiber to the props of the new fiber, remove the props that are gone, and set the props that are new or changed.

One special kind of prop that we need to update are event listeners, so if the prop name starts with the "on" prefix we'll handle them differently.

If the event handler changed we remove it from the node.

And then we add the new handler.

Try the version with reconciliation on codesandbox.

```
function createElement(type, props, ...children) {
```

```
return {

  type,

  props: {

    ...props,

    children: children.map(child =>
```

Step VII: Function Components

The next thing we need to add is support for function components.

First let's change the example. We'll use this simple function component, that returns an `h1` element.

Note that if we transform the jsx to js, it will be:

```
function App(props) {
  return Didact.createElement(
    "h1",
    null,
    "Hi ",
    props.name
  )
```

```
}
const element = Didact.createElement(App,
{
  name: "foo",
})
```

Function components are differents in two ways:

- the fiber from a function component doesn't have a DOM node
- and the children come from running the function instead of getting them directly from the `props`

We check if the fiber type is a function, and depending on that we go to a different update function.

In `updateHostComponent` we do the same as before.

And in `updateFunctionComponent` we run the function to get the children.

For our example, here the `fiber.type` is the `App` function and when we run it, it returns the `h1` element.

Then, once we have the children, the reconciliation works in the same way, we don't need to change anything there.

What we need to change is the `commitWork` function.

Now that we have fibers without DOM nodes we need to change two things.

First, to find the parent of a DOM node we'll need to go up the fiber tree until we find a fiber with a DOM node.

And when removing a node we also need to keep going until we find a child with a DOM node.

Step VIII: Hooks

Last step. Now that we have function components let's also add state.

Let's change our example to the classic counter component. Each time we click it, it increments the state by one.

Note that we are using `Didact.useState` to get and update the counter value.

Here is where we call the `Counter` function from the example. And inside that function we call `useState`.

We need to initialize some global variables before calling the function component so we can use them inside of the `useState` function.

First we set the work in progress fiber.

We also add a `hooks` array to the fiber to support calling `useState` several times in the same component. And we keep track of the current hook index.

When the function component calls `useState`, we check if we have an old hook. We check in the `alternate` of the fiber using the hook index.

If we have an old hook, we copy the state from the old hook to the new hook, if we don't we initialize the state.

Then we add the new hook to the fiber, increment the hook index by one, and return the state.

`useState` should also return a function to update the state, so we define a `setState` function that receives an action (for the `Counter` example this action is the function that increments the state by one).

We push that action to a queue we added to the hook.

And then we do something similar to what we did in the `render` function, set a new work in progress root as the next unit of work so the work loop can start a new render phase.

But we haven't run the action yet.

We do it the next time we are rendering the component, we get all the actions from the old hook queue, and then apply them one by one to the new hook state, so when we return the state it's updated.

And that's all. We've built our own version of React.

You can play with it on codesandbox or github.

Epilogue

Besides helping you understand how React works, one of the goals of this post is to make it easier for you to dive deeper in the React codebase. That's why we used the same variable and function names almost everywhere.

For example, if you add a breakpoint in one of your function components in a real React app, the call stack should show you:

- `workLoop`
- `performUnitOfWork`
- `updateFunctionComponent`

We didn't include a lot of React features and optimizations. For example, these are a few things

that React does differently:

- In Didact, we are walking the whole tree during the render phase. React instead follows some hints and heuristics to skip entire sub-trees where nothing changed.
- We are also walking the whole tree in the commit phase. React keeps a linked list with just the fibers that have effects and only visit those fibers.
- Every time we build a new work in progress tree, we create new objects for each fiber. React recycles the fibers from the previous trees.
- When Didact receives a new update during the render phase, it throws away the work in progress tree and starts again from the root. React tags each update with an expiration timestamp and uses it to decide which update has a higher priority.
- And many more...

There are also a few features that you can add easily:

- use an object for the style prop
- flatten children arrays
- useEffect hook
- reconciliation by key

If you add any of these or other features to Didact send a pull request to the GitHub repo, so others can see it.

Thanks for reading!

And if you want to comment, like or share this post you can use this tweet:

> Had to build a new blog and some tools to be able to publish this post with the format I wanted. It took some time but it's finally ready!
>
> 📢 the updated DIY guide to build React from scratch
> ✨ https://t.co/RfGrl8ARYz
> pic.twitter.com/3kih0xLHIu
>
> — Rodrigo Pombo (@pomber) November 13, 2019

* * *

Rodrigo Pombo, a.k.a. @pomber, is a software overengineer working on **Code Hike**, an open-source library that bridges the gap between Markdown and React to help developers **create rich technical content for the modern web**.