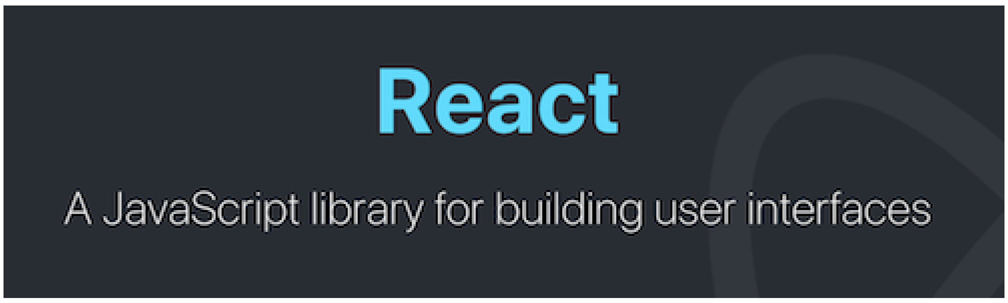
by

React as a UI Runtime

February 2, 2019

Most tutorials introduce React as a UI library. This makes sense because React *is* a UI library. That's literally what the tagline says!



React

A JavaScript library for building user interfaces

I've written about the challenges of creating user interfaces before. But this post talks about React in a different way — more as a programming runtime.

This post won't teach you anything about creating user interfaces. But it might help you understand the React programming model in more depth.

Note: If you're *learning* React, check out the docs instead.

⚠️

This is a deep dive — THIS IS NOT a beginner-friendly post. In this post, I'm describing most of the React programming model from first principles. I don't explain how to use it — just how it works.

It's aimed at experienced programmers and folks working on other UI libraries who asked about some tradeoffs chosen in React. I hope you'll find it useful!

Many people successfully use React for years without thinking about most of these topics. This is definitely a programmer-centric view of React rather than, say, a designer-centric one. But I don't think it hurts to have resources for both.

With that disclaimer out of the way, let's go!

Host Tree

Some programs output numbers. Other programs output poems. Different languages and their runtimes are often optimized for a particular set of use cases, and React is no exception to that.

React programs usually output **a tree that may change over time**. It might be a DOM tree, an iOS hierarchy, a tree of PDF primitives, or even of JSON objects. However, usually, we want to represent some UI with it. We'll call it a "*host* tree" because it is a part of the *host environment* outside of React — like DOM or iOS. The host tree usually has its own imperative API. React is a layer on top of it.

So what is React useful for? Very abstractly, it helps you write a program that predictably manipulates a complex host tree in response to external events like interactions, network responses, timers, and so on.

A specialized tool works better than a generic one when it can impose and benefit from particular constraints. React makes a bet on two principles:

- **Stability.** The host tree is relatively stable and most updates don't radically change its overall structure. If an app rearranged all its interactive elements into a completely different combination every second, it would be difficult to use. Where did that button go? Why is my screen dancing?

- **Regularity.** The host tree can be broken down into UI patterns that look and behave consistently (such as buttons, lists, avatars) rather than random shapes.

These principles happen to be true for most UIs. However, React is ill-suited when there are no stable “patterns” in the output. For example, React may help you write a Twitter client but won’t be very useful for a 3D pipes screensaver.

Host Instances

The host tree consists of nodes. We’ll call them “host instances”.

In the DOM environment, host instances are regular DOM nodes — like the objects you get when you call `document.createElement('div')`. On iOS, host instances could be values uniquely identifying a native view from JavaScript.

Host instances have their own properties (e.g. `domNode.className` or `view.tintColor`). They may also contain other host instances as children.

(This has nothing to do with React — I’m describing the host environments.)

There is usually an API to manipulate host instances. For example, the DOM provides APIs like `appendChild`, `removeChild`, `setAttribute`, and so on. In React apps, you usually don’t call these APIs. That’s the job of React.

Renderers

A *renderer* teaches React to talk to a specific host environment and manage its host instances. React DOM, React Native, and even Ink are React renderers. You can also create your own React renderer.

React renderers can work in one of two modes.

The vast majority of renderers are written to use the “mutating” mode. This mode is how the DOM works: we can create a node, set its properties, and later add or remove children from it. The host instances are completely mutable.

React can also work in a “persistent” mode. This mode is for host environments that don’t provide methods like `appendChild()` but instead clone the parent tree and always replace the top-level child. Immutability on the host tree level makes multi-threading easier. React Fabric takes advantage of that.

As a React user, you never need to think about these modes. I only want to highlight that React isn’t just an adapter from one mode to another. Its usefulness is orthogonal to the target low-level view API paradigm.

React Elements

In the host environment, a host instance (like a DOM node) is the smallest building block. In React, the smallest building block is a *React element*.

A React element is a plain JavaScript object. It can *describe* a host instance.

```
// JSX is a syntax sugar for these objects.
// <button className="blue" />
{
  type: 'button',
  props: { className: 'blue' }
}
```

A React element is lightweight and has no host instance tied to it. Again, it is merely a *description* of what you want to see on the screen.

Like host instances, React elements can form a tree:

```
// JSX is a syntax sugar for these objects.
// <dialog>
//   <button className="blue" />
//   <button className="red" />
// </dialog>
{
  type: 'dialog',
  props: {
    children: [{
      type: 'button',
      props: { className: 'blue' }
    }, {
      type: 'button',
      props: { className: 'red' }
    }]
  }
}
```

(Note: I omitted some properties that aren't important to this explanation.)

However, remember that **React elements don't have their own persistent identity.** They're meant to be re-created and thrown away all the time.

React elements are immutable. For example, you can't change the children or a property of a React element. If you want to render something different later, you will *describe* it with a new React element tree created from scratch.

I like to think of React elements as being like frames in a movie. They capture what the UI should look like at a specific point in time. They don't change.

Entry Point

Each React renderer has an "entry point". It's the API that lets us tell React to render a particular React element tree inside a container host instance.

For example, React DOM entry point is `ReactDOM.render` :

```
ReactDOM.render(
  // { type: 'button', props: { className: 'blue' } }
  <button className="blue" />,
  document.getElementById('container')
);
```

When we say `ReactDOM.render(reactElement, domContainer)`, we mean: **"Dear React, make the `domContainer` host tree match my `reactElement`."**

React will look at the `reactElement.type` (in our example, `'button'`) and ask the React DOM renderer to create a host instance for it and set the properties:

```
// Somewhere in the ReactDOM renderer (simplified)
function createHostInstance(reactElement) {
  let domNode = document.createElement(reactElement.type);
  domNode.className = reactElement.props.className;
  return domNode;
}
```

In our example, effectively React will do this:

```
let domNode = document.createElement('button');
domNode.className = 'blue';

domContainer.appendChild(domNode);
```

If the React element has child elements in `reactElement.props.children`, React will recursively create host instances for them too on the first render.

Reconciliation

What happens if we call `ReactDOM.render()` twice with the same container?

```
ReactDOM.render(
  <button className="blue" />,
  document.getElementById('container')
);

// ... later ...

// Should this *replace* the button host instance
// or merely update a property on an existing one?
ReactDOM.render(
  <button className="red" />,
  document.getElementById('container')
);
```

Again, React's job is to *make the host tree match the provided React element tree*. The process of figuring out *what* to do to the host instance tree in response to new information is sometimes called reconciliation.

There are two ways to go about it. A simplified version of React could blow away the existing tree and re-create it from scratch:

```
let domContainer = document.getElementById('container');
// Clear the tree
domContainer.innerHTML = '';
// Create the new host instance tree
let domNode = document.createElement('button');
domNode.className = 'red';
domContainer.appendChild(domNode);
```

But in DOM, this is slow and loses important information like focus, selection, scroll state, and so on. Instead, we want React to do something like this:

```
let domNode = domContainer.firstChild;
// Update existing host instance
domNode.className = 'red';
```

In other words, React needs to decide when to *update* an existing host instance to match a new React element, and when to create a *new* one.

This raises a question of *identity*. The React element may be different every time, but when does it refer to the same host instance conceptually?

In our example, it's simple. We used to render a `<button>` as a first (and only) child, and we want to render a `<button>` in the same place again. We already have a `<button>` host instance there so why re-create it? Let's just reuse it.

This is pretty close to how React thinks about it.

If an element type in the same place in the tree "matches up" between the previous and the next renders, React reuses the existing host instance.

Here is an example with comments showing roughly what React does:

```
// let domNode = document.createElement('button');
// domNode.className = 'blue';
// domContainer.appendChild(domNode);
ReactDOM.render(
  <button className="blue" />,
  document.getElementById('container')
);

// Can reuse host instance? Yes! (button → button)
// domNode.className = 'red';
ReactDOM.render(
  <button className="red" />,
  document.getElementById('container')
);

// Can reuse host instance? No! (button → p)
// domContainer.removeChild(domNode);
// domNode = document.createElement('p');
// domNode.textContent = 'Hello';
// domContainer.appendChild(domNode);
ReactDOM.render(
  <p>Hello</p>,
  document.getElementById('container')
);

// Can reuse host instance? Yes! (p → p)
// domNode.textContent = 'Goodbye';
ReactDOM.render(
  <p>Goodbye</p>,
  document.getElementById('container')
);
```

The same heuristic is used for child trees. For example, when we update a `<dialog>` with two `<button>` s inside, React first decides whether to re-use the `<dialog>` , and then repeats this decision procedure for each child.

Conditions

If React only reuses host instances when the element types “match up” between updates, how can we render conditional content?

Say we want to first show only an input, but later render a message before it:

```
// First render
ReactDOM.render(
  <dialog>
    <input />
  </dialog>,
  domContainer
);

// Next render
ReactDOM.render(
  <dialog>
    <p>I was just added here!</p>
    <input />
  </dialog>,
  domContainer
);
```

In this example, the `<input>` host instance would get re-created. React would walk the element tree, comparing it with the previous version:

- `dialog → dialog`: Can reuse the host instance? **Yes — the type matches.**
- `input → p`: Can reuse the host instance? **No, the type has changed!** Need to remove the existing `input` and create a new `p` host instance.
- `(nothing) → input`: Need to create a new `input` host instance.

So effectively the update code executed by React would be like:

```
let oldInputNode = dialogNode.firstChild;
dialogNode.removeChild(oldInputNode);

let pNode = document.createElement('p');
pNode.textContent = 'I was just added here!';
dialogNode.appendChild(pNode);

let newInputNode = document.createElement('input');
dialogNode.appendChild(newInputNode);
```

This is not great because *conceptually* the `<input>` hasn't been *replaced* with `<p>` — it just moved. We don't want to lose its selection, focus state, and content due to re-creating the DOM.

While this problem has an easy fix (which we'll get to in a minute), it doesn't occur often in React applications. It's interesting to see why.

In practice, you would rarely call `ReactDOM.render` directly. Instead, React apps tend to be broken down into functions like this:

```
function Form({ showMessage }) {
  let message = null;
  if (showMessage) {
    message = <p>I was just added here!</p>;
  }
  return (
    <dialog>
      {message}
      <input />
    </dialog>
  );
}
```

This example doesn't suffer from the problem we just described. It might be easier to see why if we use object notation instead of JSX. Look at the `dialog` child element tree:

```
function Form({ showMessage }) {
  let message = null;
  if (showMessage) {
    message = {
      type: 'p',
      props: { children: 'I was just added here!' }
    };
  }
  return {
    type: 'dialog',
    props: {
      children: [
        message,
        { type: 'input', props: {} }
      ]
    }
  };
}
```

Regardless of whether `showMessage` is `true` or `false`, the `<input>` is the second child and doesn't change its tree position between renders.

If `showMessage` changes from `false` to `true`, React would walk the element tree, comparing it with the previous version:

- `dialog → dialog`: Can reuse the host instance? **Yes — the type matches.**
- `(null) → p`: Need to insert a new `p` host instance.
- `input → input`: Can reuse the host instance? **Yes — the type matches.**

And the code executed by React would be similar to this:

```
let inputNode = dialogNode.firstChild;
let pNode = document.createElement('p');
pNode.textContent = 'I was just added here!';
dialogNode.insertBefore(pNode, inputNode);
```

No input state is lost now.

Lists

Comparing the element type at the same position in the tree is usually enough to decide whether to reuse or re-create the corresponding host instance.

But this only works well if child positions are static and don't re-order. In our example above, even though `message` could be a "hole", we still knew that the input goes after the message, and there are no other children.

With dynamic lists, we can't be sure the order is ever the same:

```
function ShoppingList({ list }) {
  return (
    <form>
      {list.map(item => (
        <p>
          You bought {item.name}
          <br />
          Enter how many do you want: <input />
        </p>
      ))}
    </form>
  )
}
```

If the `list` of our shopping items is ever re-ordered, React will see that all `p` and `input` elements inside have the same type, and won't know to move them. (From React's point of view, the *items themselves* changed, not their order.)

The code executed by React to re-order 10 items would be something like:

```
for (let i = 0; i < 10; i++) {
  let pNode = formNode.childNodes[i];
  let textNode = pNode.firstChild;
  textNode.textContent = 'You bought ' + items[i].name;
}
```

So instead of *re-ordering* them, React would effectively *update* each of them. This can create performance issues and possible bugs. For example, the content of the first input would stay reflected in first input *after* the sort — even though conceptually they might refer to different products in your shopping list!

This is why React nags you to specify a special property called `key` every time you include an array of elements in your output:

```
function ShoppingList({ list }) {
  return (
    <form>
      {list.map(item => (
        <p key={item.productId}>
          You bought {item.name}
          <br />
          Enter how many do you want: <input />
        </p>
      ))}
    </form>
  )
}
```

A `key` tells React that it should consider an item to be *conceptually* the same even if it has different *positions* inside its parent element between renders.

When React sees `<p key="42">` inside a `<form>`, it will check if the previous render also contained `<p key="42">` inside the same `<form>`. This works even if `<form>` children changed their order. React will reuse the previous host instance with the same key if it exists, and re-order the siblings accordingly.

Note that the `key` is only relevant within a particular parent React element, such as a `<form>`. React won't try to "match up" elements with the same keys between different parents. (React doesn't have idiomatic support for moving a host instance between different parents without re-creating it.)

What's a good value for a `key`? An easy way to answer this is to ask: **when would *you* say an item is the "same" even if the order changed?** For example, in our shopping list, the product ID uniquely identifies it between siblings.

Components

We’ve already seen functions that return React elements:

```
function Form({ showMessage }) {
  let message = null;
  if (showMessage) {
    message = <p>I was just added here!</p>;
  }
  return (
    <dialog>
      {message}
      <input />
    </dialog>
  );
}
```

They are called *components*. They let us create our own “toolbox” of buttons, avatars, comments, and so on. Components are the bread and butter of React.

Components take one argument — an object hash. It contains “props” (short for “properties”). Here, `showMessage` is a prop. They’re like named arguments.

Purity

React components are assumed to be pure with respect to their props.

```
function Button(props) {
  // 🔴 Doesn't work
  props.isActive = true;
}
```

In general, mutation is not idiomatic in React. (We’ll talk more about the idiomatic way to update the UI in response to events later.)

However, *local mutation* is absolutely fine:

```
function FriendList({ friends }) {
  let items = [];
  for (let i = 0; i < friends.length; i++) {
    let friend = friends[i];
    items.push(
      <Friend key={friend.id} friend={friend} />
    );
  }
  return <section>{items}</section>;
}
```

We created `items` *while rendering* and no other component "saw" it so we can mutate it as much as we like before handing it off as part of the render result. There is no need to contort your code to avoid local mutation.

Similarly, lazy initialization is fine despite not being fully "pure":

```
function ExpenseForm() {
  // Fine if it doesn't affect other components:
  SuperCalculator.initializeIfNotReady();

  // Continue rendering...
}
```

As long as calling a component multiple times is safe and doesn't affect the rendering of other components, React doesn't care if it's 100% pure in the strict FP sense of the word. Idempotence is more important to React than purity.

That said, side effects that are directly visible to the user are not allowed in React components. In other words, merely *calling* a component function shouldn't by itself produce a change on the screen.

Recursion

How do we *use* components from other components? Components are functions so we *could* call them:

```
let reactElement = Form({ showMessage: true });
ReactDOM.render(reactElement, domContainer);
```

However, this is *not* the idiomatic way to use components in the React runtime.

Instead, the idiomatic way to use a component is with the same mechanism we've already seen before — React elements. **This means that you don't directly call the component function, but instead let React later do it for you**:

```
// { type: Form, props: { showMessage: true } }
let reactElement = <Form showMessage={true} />;
ReactDOM.render(reactElement, domContainer);
```

And somewhere inside React, your component will be called:

```
// Somewhere inside React
let type = reactElement.type; // Form
let props = reactElement.props; // { showMessage: true }
let result = type(props); // Whatever Form returns
```

Component function names are by convention capitalized. When the JSX transform sees `<Form>` rather than `<form>`, it makes the object `type` itself an identifier rather than a string:

```
console.log(<form />.type); // 'form' string
console.log(<Form />.type); // Form function
```

There is no global registration mechanism — we literally refer to `Form` by name when typing `<Form />`. If `Form` doesn't exist in local scope, you'll see a JavaScript error just like you normally would with a bad variable name.

Okay, so what does React do when an element type is a function? It calls your component, and asks what element *that* component wants to render.

This process continues recursively and is described in more detail here. In short, it looks like this:

- **You:** `ReactDOM.render(<App />, domContainer)`

- **React:** Hey `App` , what do you render to?
- `App` : I render `<Layout>` with `<Content>` inside.

- **React:** Hey `Layout` , what do you render to?
- `Layout` : I render my children in a `<div>` . My child was `<Content>` so I guess that goes into the `<div>` .

- **React:** Hey `<Content>` , what do you render to?
- `Content` : I render an `<article>` with some text and a `<Footer>` inside.

- **React:** Hey `<Footer>` , what do you render to?
- `Footer` : I render a `<footer>` with some more text.

- **React:** Okay, here you go:

```
// Resulting DOM structure
<div>
  <article>
    Some text
    <footer>some more text</footer>
  </article>
</div>
```

This is why we say reconciliation is recursive. When React walks the element tree, it might meet an element whose `type` is a component. It will call it and keep descending down the tree of returned React elements. Eventually, we'll run out of components, and React will know what to change in the host tree.

The same reconciliation rules we already discussed apply here too. If the `type` at the same position (as determined by index and optional `key`) changes, React will throw away the host instances inside, and re-create them.

Inversion of Control

You might be wondering: why don't we just call components directly? Why write `<Form />` rather than `Form()`?

React can do its job better if it "knows" about your components rather than if it only sees the React element tree after recursively calling them.

```
// 🔴 React has no idea Layout and Article exist.
// You're calling them.
ReactDOM.render(
  Layout({ children: Article() }),
  domContainer
)

// ✅ React knows Layout and Article exist.
// React calls them.
ReactDOM.render(
  <Layout><Article /></Layout>,
  domContainer
)
```

This is a classic example of inversion of control. There's a few interesting properties we get by letting React take control of calling our components:

- **Components become more than functions.** React can augment component functions with features like *local state* that are tied to the component identity in the tree. A good runtime provides fundamental abstractions that match the problem at hand. As we already mentioned, React is oriented specifically at programs that render UI trees and respond to interactions. If you called components directly, you'd have to build these features yourself.

- **Component types participate in the reconciliation.** By letting React call your components, you also tell it more about the conceptual structure of your tree. For example, when you move from rendering `<Feed>` to the `<Profile>` page, React won't attempt to re-use host instances inside them — just like when you replace `<button>` with a `<p>`. All state will be gone — which is usually good when you render a conceptually different view. You wouldn't want to preserve input state between `<PasswordForm>` and `<MessengerChat>` even if the `<input>` position in the tree accidentally "lines up" between them.

- **React can delay the reconciliation.** If React takes control over calling our components, it can do many interesting things. For example, it can let the browser do some work between the component calls so that re-rendering a large component tree doesn't block the main thread. Orchestrating this manually without reimplementing a large part of React is difficult.

- **A better debugging story.** If components are first-class citizens that the library is aware of, we can build rich developer tools for introspection in development.

The last benefit to React calling your component functions is *lazy evaluation*. Let's see what this means.

Lazy Evaluation

When we call functions in JavaScript, arguments are evaluated before the call:

```
// (2) This gets computed second
eat(
  // (1) This gets computed first
  prepareMeal()
);
```

This is usually what JavaScript developers expect because JavaScript functions can have implicit side effects. It would be surprising if we called a function, but it wouldn't execute until its result gets somehow "used" in JavaScript.

However, React components are relatively pure. There is absolutely no need to execute it if we know its result won't get rendered on the screen.

Consider this component putting `<Comments>` inside a `<Page>`:

```
function Story({ currentUser }) {
  // return {
  //   type: Page,
  //   props: {
  //     user: currentUser,
  //     children: { type: Comments, props: {} }
  //   }
  // }
  return (
    <Page user={currentUser}>
      <Comments />
    </Page>
  );
}
```

The `Page` component can render the children given to it inside some `Layout`:

```
function Page({ user, children }) {
  return (
    <Layout>
      {children}
    </Layout>
  );
}
```

(`<A><B /></A>` in JSX is the same as `<A children={<B />} />`.)

But what if it has an early exit condition?

```
function Page({ user, children }) {
  if (!user.isLoggedIn) {
    return <h1>Please log in</h1>;
  }
  return (
    <Layout>
      {children}
    </Layout>
  );
}
```

If we called `Comments()` as a function, it would execute immediately regardless of whether `Page` wants to render them or not:

```
// {
//   type: Page,
//   props: {
//     children: Comments() // Always runs!
//   }
// }
<Page>
  {Comments()}
</Page>
```

But if we pass a React element, we don't execute `Comments` ourselves at all:

```
// {
//   type: Page,
//   props: {
//     children: { type: Comments }
//   }
// }
<Page>
  <Comments />
</Page>
```

This lets React decide when and *whether* to call it. If our `Page` component ignores its `children` prop and renders `<h1>Please log in</h1>` instead, React won't even attempt to call the `Comments` function. What's the point?

This is good because it both lets us avoid unnecessary rendering work that would be thrown away, and makes the code less fragile. (We don't care if `Comments` throws or not when the user is logged out — it won't be called.)

State

We talked earlier about identity and how an element's conceptual "position" in the tree tells React whether to re-use a host instance or create a new one. Host instances can have all kinds of local state: focus, selection, input, etc. We want to preserve this state between updates that conceptually render the same UI. We also want to predictably destroy it when we render something conceptually different (such as moving from `<SignupForm>` to `<MessengerChat>`).

Local state is so useful that React lets *your own* components have it too. Components are still functions but React augments them with features that are useful for UIs. Local state tied to the position in the tree is one of these features.

We call these features *Hooks*. For example, `useState` is a Hook.

```
function Example() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

It returns a pair of values: the current state and a function that updates it.

The array destructuring syntax lets us give arbitrary names to our state variables. For example, I called this pair `count` and `setCount`, but it could've been `banana` and `setBanana`. In the text below, I will use `setState` to refer to the second value regardless of its actual name in the specific examples.

(You can learn more about `useState` and other Hooks provided by React here.)

Consistency

Even if we want to split the reconciliation process itself into non-blocking chunks of work, we should still perform the actual host tree operations in a single synchronous swoop. This way we can ensure that the user doesn't see a half-updated UI, and that the browser doesn't perform unnecessary layout and style recalculation for intermediate states that the user shouldn't see.

This is why React splits all work into the "render phase" and the "commit phase". *Render phase* is when React calls your components and performs reconciliation. It is safe to interrupt and in the future will be asynchronous. *Commit phase* is when React touches the host tree. It is always synchronous.

Memoization

When a parent schedules an update by calling `setState`, by default React reconciles its whole child subtree. This is because React can't know whether an update in the parent would affect the child or not, and by default, React opts to be consistent. This may sound very expensive but in practice, it's not a problem for small and medium-sized subtrees.

When trees get too deep or wide, you can tell React to memoize a subtree and reuse previous render results during shallow equal prop changes:

```
function Row({ item }) {
  // ...
}

export default React.memo(Row);
```

Now `setState` in a parent `<Table>` component would skip over reconciling `Row`s whose `item` is referentially equal to the `item` rendered last time.

You can get fine-grained memoization at the level of individual expressions with the `useMemo()` Hook. The cache is local to component tree position and will be destroyed together with its local state. It only holds one last item.

React intentionally doesn't memoize components by default. Many components always receive different props so memoizing them would be a net loss.

Raw Models

Ironically, React doesn't use a "reactivity" system for fine-grained updates. In other words, any update at the top triggers reconciliation instead of updating just the components affected by changes.

This is an intentional design decision. Time to interactive is a crucial metric in consumer web applications, and traversing models to set up fine-grained listeners spends that precious time. Additionally, in many apps, interactions tend to result either in small (button hover) or large (page transition) updates, in which case fine-grained subscriptions are a waste of memory resources.

One of the core design principles of React is that it works with raw data. If you have a bunch of JavaScript objects received from the network, you can pump them directly into your components with no preprocessing. There are no gotchas about which properties you can access, or unexpected performance cliffs when a structure slightly changes. React rendering is O(*view size*) rather than O(*model size*), and you can significantly cut the *view size* with windowing.

There are some kinds of applications where fine-grained subscriptions are beneficial — such as stock tickers. This is a rare example of "everything constantly updating at the same time". While imperative escape hatches can help optimize such code, React might not be the best fit for this use case. Still, you can implement your own fine-grained subscription system on top of React.

Note that there are common performance issues that even fine-grained subscriptions and "reactivity" systems can't solve. For example, rendering a *new* deep tree (which happens on every page transition) without blocking the browser. Change tracking doesn't make it faster — it makes it slower because we have to do more work to set up subscriptions. Another problem is that we have to wait for data before we can start rendering the view. In React, we aim to solve both of these problems with Concurrent Rendering.

Batching

Several components may want to update state in response to the same event. This example is contrived but it illustrates a common pattern:

```
function Parent() {
  let [count, setCount] = useState(0);
  return (
    <div onClick={() => setCount(count + 1)}>
      Parent clicked {count} times
      <Child />
    </div>
  );
}

function Child() {
  let [count, setCount] = useState(0);
  return (
    <button onClick={() => setCount(count + 1)}>
      Child clicked {count} times
    </button>
  );
}
```

When an event is dispatched, the child's `onClick` fires first (triggering its `setState`). Then the parent calls `setState` in its own `onClick` handler.

If React immediately re-rendered components in response to `setState` calls, we'd end up rendering the child twice:

```
*** Entering React's browser click event handler ***
Child (onClick)
  - setState
  - re-render Child // 😞 unnecessary
Parent (onClick)
  - setState
  - re-render Parent
  - re-render Child
*** Exiting React's browser click event handler ***
```

The first `Child` render would be wasted. And we couldn't make React skip rendering `Child` for the second time because the `Parent` might pass some different data to it based on its updated state.

This is why React batches updates inside event handlers:

```
*** Entering React's browser click event handler ***
Child (onClick)
  - setState
Parent (onClick)
  - setState
*** Processing state updates                     ***
  - re-render Parent
  - re-render Child
*** Exiting React's browser click event handler  ***
```

The `setState` calls in components wouldn't immediately cause a re-render. Instead, React would execute all event handlers first, and then trigger a single re-render batching all of those updates together.

Batching is good for performance but can be surprising if you write code like:

```
  const [count, setCount] = useState(0);

  function increment() {
    setCount(count + 1);
  }

  function handleClick() {
    increment();
    increment();
    increment();
  }
```

If we start with `count` set to `0`, these would just be three `setCount(1)` calls. To fix this, `setState` provides an overload that accepts an "updater" function:

```
  const [count, setCount] = useState(0);

  function increment() {
    setCount(c => c + 1);
  }
```

```
function handleClick() {
  increment();
  increment();
  increment();
}
```

React would put the updater functions in a queue, and later run them in sequence, resulting in a re-render with `count` set to `3`.

When state logic gets more complex than a few `setState` calls, I recommend expressing it as a local state reducer with the `useReducer` Hook. It's like an evolution of this "updater" pattern where each update is given a name:

```
const [counter, dispatch] = useReducer((state, action) => {
  if (action === 'increment') {
    return state + 1;
  } else {
    return state;
  }
}, 0);

function handleClick() {
  dispatch('increment');
  dispatch('increment');
  dispatch('increment');
}
```

The `action` argument can be anything, although an object is a common choice.

Call Tree

A programming language runtime usually has a call stack. When a function `a()` calls `b()` which itself calls `c()`, somewhere in the JavaScript engine there's a data structure like `[a, b, c]` that "keeps track" of where you are and what code to execute next. Once you exit out of `c`, its call stack frame is gone — poof! It's not needed anymore. We jump back into `b`. By the time we exit `a`, the call stack is empty.

Of course, React itself runs in JavaScript and obeys JavaScript rules. But we can imagine that internally React has some kind of its own call stack to remember which component we are currently rendering, e.g. `[App, Page, Layout, Article /* we're here */]`.

React is different from a general purpose language runtime because it's aimed at rendering UI trees. These trees need to "stay alive" for us to interact with them. The DOM doesn't disappear after our first `ReactDOM.render()` call.

This may be stretching the metaphor but I like to think of React components as being in a "call tree" rather than just a "call stack". When we go "out" of the `Article` component, its React "call tree" frame doesn't get destroyed. We need to keep the local state and references to the host instances somewhere.

These "call tree" frames *are* destroyed along with their local state and host instances, but only when the reconciliation rules say it's necessary. If you ever read React source, you might have seen these frames being referred to as Fibers.

Fibers are where the local state actually lives. When the state is updated, React marks the Fibers below as needing reconciliation, and calls those components.

Context

In React, we pass things down to other components as props. Sometimes, the majority of components need the same thing — for example, the currently chosen visual theme. It gets cumbersome to pass it down through every level.

In React, this is solved by Context. It is essentially like dynamic scoping for components. It's like a wormhole that lets you put something on the top, and have every child at the bottom be able to read it and re-render when it changes.

```
const ThemeContext = React.createContext(
  'light' // Default value as a fallback
);

function DarkApp() {
  return (
    <ThemeContext.Provider value="dark">
      <MyComponents />
    </ThemeContext.Provider>
  );
}

function SomeDeeplyNestedChild() {
  // Depends on where the child is rendered
  const theme = useContext(ThemeContext);
  // ...
}
```

When `SomeDeeplyNestedChild` renders, `useContext(ThemeContext)` will look for the closest `<ThemeContext.Provider>` above it in the tree, and use its `value`.

(In practice, React maintains a context stack while it renders.)

If there's no `ThemeContext.Provider` above, the result of `useContext(ThemeContext)` call will be the default value specified in the `createContext()` call. In our example, it is `'light'`.

Effects

We mentioned earlier that React components shouldn't have observable side effects during rendering. But side effects are sometimes necessary. We may want to manage focus, draw on a canvas, subscribe to a data source, and so on.

In React, this is done by declaring an effect:

```
function Example() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
```

```
        Click me
      </button>
    </div>
  );
}
```

When possible, React defers executing effects until after the browser re-paints the screen. This is good because code like data source subscriptions shouldn't hurt time to interactive and time to first paint. (There's a rarely used Hook that lets you opt out of that behavior and do things synchronously. Avoid it.)

Effects don't just run once. They run both after a component is shown to the user for the first time, and after it updates. Effects can close over current props and state, such as with `count` in the above example.

Effects may require cleanup, such as in case of subscriptions. To clean up after itself, an effect can return a function:

```
useEffect(() => {
  DataSource.addSubscription(handleChange);
  return () => DataSource.removeSubscription(handleChange);
});
```

React will execute the returned function before applying this effect the next time, and also before the component is destroyed.

Sometimes, re-running the effect on every render can be undesirable. You can tell React to skip applying an effect if certain variables didn't change:

```
useEffect(() => {
  document.title = `You clicked ${count} times`;
}, [count]);
```

However, it is often a premature optimization and can lead to problems if you're not familiar with how JavaScript closures work.

For example, this code is buggy:

```
useEffect(() => {
  DataSource.addSubscription(handleChange);
  return () => DataSource.removeSubscription(handleChange);
}, []);
```

It is buggy because `[]` says “don’t ever re-execute this effect”. But the effect closes over `handleChange` which is defined outside of it. And `handleChange` might reference any props or state:

```
function handleChange() {
  console.log(count);
}
```

If we never let the effect re-run, `handleChange` will keep pointing at the version from the first render, and `count` will always be `0` inside of it.

To solve this, make sure that if you specify the dependency array, it includes **all** things that can change, including the functions:

```
useEffect(() => {
  DataSource.addSubscription(handleChange);
  return () => DataSource.removeSubscription(handleChange);
}, [handleChange]);
```

Depending on your code, you might still see unnecessary resubscriptions because `handleChange` itself is different on every render. The `useCallback` Hook can help you with that. Alternatively, you can just let it re-subscribe. For example, browser’s `addEventListener` API is extremely fast, and jumping through hoops to avoid calling it might cause more problems than it’s worth.

(You can learn more about `useEffect` and other Hooks provided by React here.)

Custom Hooks

Since Hooks like `useState` and `useEffect` are function calls, we can compose them into our own Hooks:

```
function MyResponsiveComponent() {
  const width = useWindowWidth(); // Our custom Hook
  return (
    <p>Window width is {width}</p>
  );
}

function useWindowWidth() {
  const [width, setWidth] = useState(window.innerWidth);
  useEffect(() => {
    const handleResize = () => setWidth(window.innerWidth);
    window.addEventListener('resize', handleResize);
    return () => {
      window.removeEventListener('resize', handleResize);
    };
  });
  return width;
}
```

Custom Hooks let different components share reusable stateful logic. Note that the *state itself* is not shared. Each call to a Hook declares its own isolated state.

(You can learn more about writing your own Hooks here.)

Static Use Order

You can think of `useState` as a syntax for defining a "React state variable". It's not *really* a syntax, of course. We're still writing JavaScript. But we are looking at React as a runtime environment, and because React tailors JavaScript to describing UI trees, its features sometimes live closer to the language space.

If `use` *were* a syntax, it would make sense for it to be at the top level:

```
// 😉 Note: not a real syntax
component Example(props) {
  const [count, setCount] = use State(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

What would putting it into a condition or a callback or outside a component even mean?

```
// 😉 Note: not a real syntax

// This is local state... of what?
const [count, setCount] = use State(0);

component Example() {
  if (condition) {
    // What happens to it when condition is false?
    const [count, setCount] = use State(0);
  }

  function handleClick() {
    // What happens to it when we leave a function?
    // How is this different from a variable?
    const [count, setCount] = use State(0);
  }
```

React state is local to the *component* and its identity in the tree. If `use` were a real syntax it would make sense to scope it to the top-level of a component too:

```
// 😉 Note: not a real syntax
component Example(props) {
  // Only valid here
  const [count, setCount] = use State(0);

  if (condition) {
    // This would be a syntax error
    const [count, setCount] = use State(0);
  }
```

This is similar to how `import` only works at the top level of a module.

Of course, `use` is not actually a syntax. (It wouldn’t bring much benefit and would create a lot of friction.)

However, React *does* expect that all calls to Hooks happen only at the top level of a component and unconditionally. These Rules of Hooks can be enforced with a linter plugin. There have been heated arguments about this design choice but in practice, I haven’t seen it confusing people. I also wrote about why commonly proposed alternatives don’t work.

Internally, Hooks are implemented as linked lists. When you call `useState`, we move the pointer to the next item. When we exit the component’s “call tree” frame, we save the resulting list there until the next render.

This article provides a simplified explanation for how Hooks work internally. Arrays might be an easier mental model than linked lists:

```
// Pseudocode
let hooks, i;
function useState() {
  i++;
  if (hooks[i]) {
    // Next renders
    return hooks[i];
  }
  // First render
  hooks.push(...);
}

// Prepare to render
i = -1;
hooks = fiber.hooks || [];
// Call the component
YourComponent();
// Remember the state of Hooks
fiber.hooks = hooks;
```

(If you're curious, the real code is here.)

This is roughly how each `useState()` call gets the right state. As we've learned earlier, "matching things up" isn't new to React — reconciliation relies on the elements matching up between renders in a similar way.

What's Left Out

We've touched on pretty much all important aspects of the React runtime environment. If you finished this page, you probably know React in more detail than 90% of its users. And there's nothing wrong with that!

There are some parts I left out — mostly because they're unclear even to us. React doesn't currently have a good story for multipass rendering, i.e. when the parent render needs information about the children. Also, the error handling API doesn't yet have a Hooks version. It's possible that these two problems can be solved together. Concurrent Mode is not stable yet, and there are interesting questions about how Suspense fits into this picture. Maybe I'll do a follow-up when they're fleshed out and Suspense is ready for more than lazy loading.

I think it speaks to the success of React's API that you can get very far without ever thinking about most of these topics. Good defaults like the reconciliation heuristics do the right thing in most cases. Warnings, like the `key` warning, nudge you when you risk shooting yourself in the foot.

If you're a UI library nerd, I hope this post was somewhat entertaining and clarified how React works in more depth. Or maybe you decided React is too complicated and you'll never look at it again. In either case, I'd love to hear from you on Twitter! Thank you for reading.

Discuss on 𝕏 · Edit on GitHub