Building Your Own Hooks

Hooks are a new addition in React 16.8. They let you use state and other React features without writing a class.

Building your own Hooks lets you extract component logic into reusable functions.

When we were learning about using the Effect Hook, we saw this component from a chat application that displays a message indicating whether a friend is online or offline:

```
import React, { useState, useEffect } from 'react';

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);
  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

Now let's say that our chat application also has a contact list, and we want to render names of online users with a green color. We could copy and paste similar logic above into our `FriendListItem` component but it wouldn't be ideal:

```
import React, { useState, useEffect } from 'react';

function FriendListItem(props) {
  const [isOnline, setIsOnline] = useState(null);
  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  return (
    <li style={{ color: isOnline ? 'green' : 'black' }}>
      {props.friend.name}
    </li>
  );
}
```

INSTALLATION
MAIN CONCEPTS
ADVANCED GUIDES
API REFERENCE
HOOKS
1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
7. Hooks API Reference
8. Hooks FAQ
TESTING
CONCURRENT MODE (EXPERIMENTAL)
CONTRIBUTING
FAQ

Instead, we'd like to share this logic between `FriendStatus` and `FriendListItem`.

Traditionally in React, we've had two popular ways to share stateful logic between components: render props and higher-order components. We will now look at how Hooks solve many of the same problems without forcing you to add more components to the tree.

Extracting a Custom Hook

When we want to share logic between two JavaScript functions, we extract it to a third function. Both components and Hooks are functions, so this works for them too!

A custom Hook is a JavaScript function whose name starts with "use" and that may call other Hooks. For example, `useFriendStatus` below is our first custom Hook:

```
import { useState, useEffect } from 'react';

function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);
    };
  });

  return isOnline;
}
```

There's nothing new inside of it — the logic is copied from the components above. Just like in a component, make sure to only call other Hooks unconditionally at the top level of your custom Hook.

Unlike a React component, a custom Hook doesn't need to have a specific signature. We can decide what it takes as arguments, and what, if anything, it should return. In other words, it's just like a normal function. Its name should always start with `use` so that you can tell at a glance that the rules of Hooks apply to it.

The purpose of our `useFriendStatus` Hook is to subscribe us to a friend's status. This is why it takes `friendID` as an argument, and returns whether this friend is online:

```
function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  // ...

  return isOnline;
}
```

Now let's see how we can use our custom Hook.

Using a Custom Hook

INSTALLATION
MAIN CONCEPTS
ADVANCED GUIDES
API REFERENCE
HOOKS
1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
7. Hooks API Reference
8. Hooks FAQ
TESTING
CONCURRENT MODE (EXPERIMENTAL)
CONTRIBUTING
FAQ

In the beginning, our stated goal was to remove the duplicated logic from the `FriendStatus` and `FriendListItem` components. Both of them want to know whether a friend is online.

Now that we've extracted this logic to a `useFriendStatus` hook, we can *just use it:*

```
function FriendStatus(props) {
  const isOnline = useFriendStatus(props.friend.id);

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

```
function FriendListItem(props) {
  const isOnline = useFriendStatus(props.friend.id);

  return (
    <li style={{ color: isOnline ? 'green' : 'black' }}>
      {props.friend.name}
    </li>
  );
}
```

Is this code equivalent to the original examples? Yes, it works in exactly the same way. If you look closely, you'll notice we didn't make any changes to the behavior. All we did was to extract some common code between two functions into a separate function. **Custom Hooks are a convention that naturally follows from the design of Hooks, rather than a React feature.**

Do I have to name my custom Hooks starting with "use"? Please do. This convention is very important. Without it, we wouldn't be able to automatically check for violations of rules of Hooks because we couldn't tell if a certain function contains calls to Hooks inside of it.

Do two components using the same Hook share state? No. Custom Hooks are a mechanism to reuse *stateful logic* (such as setting up a subscription and remembering the current value), but every time you use a custom Hook, all state and effects inside of it are fully isolated.

How does a custom Hook get isolated state? Each *call* to a Hook gets isolated state. Because we call `useFriendStatus` directly, from React's point of view our component just calls `useState` and `useEffect`. And as we learned earlier, we can call `useState` and `useEffect` many times in one component, and they will be completely independent.

Tip: Pass Information Between Hooks

Since Hooks are functions, we can pass information between them.

To illustrate this, we'll use another component from our hypothetical chat example. This is a chat message recipient picker that displays whether the currently selected friend is online:

```
const friendList = [
  { id: 1, name: 'Phoebe' },
  { id: 2, name: 'Rachel' },
  { id: 3, name: 'Ross' },
];

function ChatRecipientPicker() {
  const [recipientID, setRecipientID] = useState(1);
  const isRecipientOnline = useFriendStatus(recipientID);
```

INSTALLATION
MAIN CONCEPTS
ADVANCED GUIDES
API REFERENCE
HOOKS
1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
7. Hooks API Reference
8. Hooks FAQ
TESTING
CONCURRENT MODE (EXPERIMENTAL)
CONTRIBUTING
FAQ

```
  return (
    <>
      <Circle color={isRecipientOnline ? 'green' : 'red'} />
      <select
        value={recipientID}
        onChange={e => setRecipientID(Number(e.target.value))}
      >
        {friendList.map(friend => (
          <option key={friend.id} value={friend.id}>
            {friend.name}
          </option>
        ))}
      </select>
    </>
  );
}
```

We keep the currently chosen friend ID in the `recipientID` state variable, and update it if the user chooses a different friend in the `<select>` picker.

Because the `useState` Hook call gives us the latest value of the `recipientID` state variable, we can pass it to our custom `useFriendStatus` Hook as an argument:

```
const [recipientID, setRecipientID] = useState(1);
const isRecipientOnline = useFriendStatus(recipientID);
```

This lets us know whether the *currently selected* friend is online. If we pick a different friend and update the `recipientID` state variable, our `useFriendStatus` Hook will unsubscribe from the previously selected friend, and subscribe to the status of the newly selected one.

`useYourImagination()`

Custom Hooks offer the flexibility of sharing logic that wasn't possible in React components before. You can write custom Hooks that cover a wide range of use cases like form handling, animation, declarative subscriptions, timers, and probably many more we haven't considered. What's more, you can build Hooks that are just as easy to use as React's built-in features.

Try to resist adding abstraction too early. Now that function components can do more, it's likely that the average function component in your codebase will become longer. This is normal — don't feel like you *have to* immediately split it into Hooks. But we also encourage you to start spotting cases where a custom Hook could hide complex logic behind a simple interface, or help untangle a messy component.

For example, maybe you have a complex component that contains a lot of local state that is managed in an ad-hoc way. `useState` doesn't make centralizing the update logic any easier so you might prefer to write it as a Redux reducer:

```
function todosReducer(state, action) {
  switch (action.type) {
    case 'add':
      return [...state, {
        text: action.text,
        completed: false
      }];
    // ... other actions ...
    default:
```

INSTALLATION
MAIN CONCEPTS
ADVANCED GUIDES
API REFERENCE
HOOKS
1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
7. Hooks API Reference
8. Hooks FAQ
TESTING
CONCURRENT MODE (EXPERIMENTAL)
CONTRIBUTING
FAQ

```
      return state;

  }
}
```

Reducers are very convenient to test in isolation, and scale to express complex update logic. You can further break them apart into smaller reducers if necessary. However, you might also enjoy the benefits of using React local state, or might not want to install another library.

So what if we could write a `useReducer` Hook that lets us manage the *local* state of our component with a reducer? A simplified version of it might look like this:

```
function useReducer(reducer, initialState) {
  const [state, setState] = useState(initialState);

  function dispatch(action) {
    const nextState = reducer(state, action);
    setState(nextState);
  }

  return [state, dispatch];
}
```

Now we could use it in our component, and let the reducer drive its state management:

```
function Todos() {
  const [todos, dispatch] = useReducer(todosReducer, []);

  function handleAddClick(text) {
    dispatch({ type: 'add', text });
  }

  // ...
}
```

The need to manage local state with a reducer in a complex component is common enough that we've built the `useReducer` Hook right into React. You'll find it together with other built-in Hooks in the Hooks API reference.

Edit this page

Previous article
Rules of Hooks

Next article
Hooks API Reference

INSTALLATION
MAIN CONCEPTS
ADVANCED GUIDES
API REFERENCE
HOOKS
1. Introducing Hooks
2. Hooks at a Glance
3. Using the State Hook
4. Using the Effect Hook
5. Rules of Hooks
6. Building Your Own Hooks
7. Hooks API Reference
8. Hooks FAQ
TESTING
CONCURRENT MODE
(EXPERIMENTAL)
CONTRIBUTING
FAQ