Conclusion

205

Jake Trent

Communicating Between Components in React

Jake Trent

Apr 24, 2019 • 9 Min read • 58,757 Views

Apr 24, 2019 • 9 Min read • 58,757 Views

Web Development React

Introduction

React is a component-based UI library. When the UI is split into small, focused components, they can do one job and do it well. But in order to build up a system into something that can accomplish an interesting task, multiple components are needed. These components often need to work in coordination together and, thus, must be able to communicate with each other. Data must flow between them.

Conclusion

React components are composed in a hierarchy that mimics the DOM tree hierarchy that they are used to create. There are those mponents that are higher (parents) and those components that lower (children) in the hierarchy. Let's take a look at the directional communication and data flow that React enables between components.

From Parent to Child with Props

The simplest data flow direction is down the hierarchy, from parent to child. React's mechanism for accomplishing this is called `props`. A React component is a function that receives a parameter called `props`. Props is a bag of data, an object that can contain any number of fields.

If a parent component wants to feed data to a child component, it simply passes it via `props`. Let's say that we have a `BookList` component that contains data for a list of books. As it iterates through the book list at render time, it wants to pass the details of each book in its list to the child `Book` component. It can do that through `props`. These `props` are passed to the child component as attributes in JSX:

js

```
function BookList() {
  const list = [
    { title: 'A Christmas Carol', author: 'Charles Dickens' },
    { title: 'The Mansion', author: 'Henry Van Dyke' },
    // ...
  ]

  return (
    <ul>
      {list.map((book, i) => <Book title={book.title} author={book.author
    </ul>
  )
}
```

Then the `Book` component can receive and use those fields as contained in the `props` parameter to its function:

js

```
function Book(props) {
  return (
    <li>
      <h2>{props.title</h2>
      <div>{props.author}</div>
    </li>
  )
}
```

Favor this simplest form of data passing whenever it makes sense.

Conclusion

205

There is a limitation here, however, because `props` are immutable. Data that is passed in `props` should never be changed. But then w does a child communicate back to its parent component? One swer is callbacks.

From Child to Parent with Callbacks

For a child to talk back to a parent (unacceptable, I know!), it must first receive a mechanism to communicate back from its parent. As we learned, parents pass data to children through `props`. A "special" prop of type `function` can be passed down to a child. At the time of a relevant event (eg, user interaction) the child can then call this function as a callback.

Let's say that a book can be edited from a `BookTitle` component:

js

```
function BookTitle(props) {
  return (
    <label>
      Title:
      <input onChange={props.onTitleChange} value={props.title} />
    </label>
  )
}
```

It receives a `onTitleChange` function in the `props`, sent from its parent. It binds this function to the `onChange` event on the `<input` field. When the input changes, it will call the `onTitleChange` callback, passing the change `Event` object.

Because the parent, `BookEditForm`, has reference to this function, it can receive the arguments that are passed to the function:

```js
import React, { useState } from 'react'

function BookEditForm(props) {
  const [title, setTitle] = useState(props.book.title)
  function handleTitleChange(evt) {
    setTitle(evt.target.value)
  }
  return (
    <form>
      <BookTitle onTitleChange={handleTitleChange} title={title} />
    </form>
  )
}
```

In this case, the parent passed `handleTitleChange`, and when it's called, it sets the internal state based on the value of `evt.target.value` -- a value that has come as a callback argument from the child component.

There are some cases, however, when data sent through `props` might not be the best option for communicating between components. For these cases, React provides a mechanism called context.

From Parent to Child with Context

If we desire something to be globally available -- in many components and levels in the hierarchy -- `props` passing has the potential to be cumbersome. Think of some data that we might like to broadcast to all child components that they react to wherever they are, such as theming data. Instead of passing theme `props` to every component down the tree or a subtree in the hierarchy, we can define a theme context to be provided at the top and then consume it in whichever child needs it down the line.

Let's say we went back to the example of a list of books in `BookList` and had a parent component above that called `BookPage`. In that component we could provide a context for the theme:

js

```
const ThemeContext = React.createContext('dark')

```

```
function BookPage() {
  return (
    <ThemeContext.Provider value="light">
      <BookList />
    </ThemeContext.Provider>
  )
}
```

205

The `ThemeContext` need only be created once and, thus, is created outside the component function. It is given a default of `"dark"` as a fallback theme name. The context object contains a `Provider` function which we wrap our rendered child component in. We can specify a `value` to override the default theme. Here, we are saying our `BookPage` will always show the `"light"` theme. Note also that `BookList` does not receive any theme props. We can leave its implementation as-is. But let's say that we want our `Book` component to respond to theming. We could adjust it to something like:

js

```
import React, { useContext } from 'react'

function Book(props) {
  const theme = useContext(ThemeContext)
  const styles = {
    dark: { background: 'black', color: 'white' },
    light: { background: 'white', color: 'black' }
  }
  return (
```

```
10          <li style={styles[theme]}>
11            <h2>{props.title</h2>
12            <div>{props.author}</div>
13          </li>
14        )
15      }
```


`Book` needs to have access to the `ThemeContext` object created next to `BookPage`, and that context object is fed to the `useContext` hook. From there, we create a simple `styles` map and select the appropriate styling based on the value of `theme`. Based on the value of `theme` provided in `BookPage`, we'll have a black color on white background styling shown here.

The thing that's special about context is that `theme` did not come from `props` but rather was simply available because a parent component provided it to any and all children which used it.

As with most global code patterns, use context sparingly. It creates coupling between components that can lead to less-reusable code and relationships or between components that are less clear.

If the context value was a callback function, we could see this being used for child to parent communication as well.

Sideways With Non-React Options

As we've seen, React provides patterns for communicating up and down the component hierarchy. Since all components exist in this hierarchy, this is natural and effective.

What if, however, we want to communicate "sideways" where data doesn't come from a parent or back up from a child? React can accomplish this by a combination of passing data up the hierarchy, then back down taking a different path to sibling components. But if we really want the data to not flow through a parent or child relationship, we have to step outside of React.

When we step outside React, the data is not going to come from `props`, context, or React-passed callbacks. It's going to come from vanilla JavaScript-type sources such as a module we `import` or a JavaScript object we observe.

There are some libraries that have formalized patterns for working with data flow outside of React but that work well with React. Redux is a common example of this, where a single state tree is maintained outside the component hierarchy but which is designed to connect easily to your components, allowing sideways-access to data.

If parent-child communication doesn't make sense for some reason, keep this non-React set of options in mind.

Conclusion

Conclusion

React's mechanisms for communicating between components are simple and effective. `props` allow data to flow down the component hierarchy from parent to child. When a child wants to communicate back up to a parent, a callback function is passed through `props`. Context provides additional convenience and makes interesting code patterns possible through globally providing data across the component tree hierarchy. There are also additional libraries and patterns that integrate well with React to communicate across components.

Experiment with all these communication patterns. Then stick with the simplest, most natural option to fit the problem you're solving. The data must flow.