React Component Composition

JANUARY 30, 2019 BY ROBIN WIERUCH - EDIT THIS POST

There are various advanced concepts and patterns in React to master on becoming a React developer. In this tutorial, I want to go through React's Component Composition step by step by evolving one example around this React pattern. You can find more about the topic in the official React documentation as well.

WHAT'S COMPOSITION IN CODE?

Let's take one step backwards before tackling composition in React. What's composition in general? It's the ingredients and the arrangement of these ingredients to create something bigger out of it. It's the samples in a piece of music that make up a track. It's the fruits that

are used for the perfect smoothie. It's the choreography of dancers in a musical. And it's the internals of a function in programming that need to be arranged in a way to get the desired output:

```
const convertCurrency = (amount, fromCurrency, toCurrency) => {
  const conversionRate = getConversionRate(fromCurrency, toCurrency);
  const newAmount = applyConversionRate(amount, conversionRate);

  return newAmount;
};
```

In functional programming, the composition of functions is ubiquitous:

```
const convertCurrency = (amount, fromCurrency, toCurrency) => compose(
    applyConversionRate(amount),
    getConversionRate(fromCurrency, toCurrency),
  );
```

And suddenly we are in the domain of programming, code and functions. As you can see, everything you do within a function is a composition of ingredients and their arrangement as well. This becomes even more true when a function is made up of functions. Then it's the composition of functions within a function.

WHY REACT COMPONENT COMPOSITION?

You have seen how multiple functions can be composed together to achieve something bigger. The same applies to HTML elements and beyond to React components too. Let's encounter both, HTML element composition and React component composition, with a form that submits data. In HTML that form element could look like the following:

```
<form action="javascript:onSubmit();">
  <label>
    Your name: <input type="text" value="">
  </label>

  <button type="submit">Send</button>
</form>
```

However, it's not only the form element but all of its other ingredients and their arrangement as well. It's the input field, the button, and the form that contribute to a greater goal: submit

data. The example is taken a bit out of context, because the JavaScript function is missing, but not so the following React example. In React, a Form as React component which is rendered within a App component could look like the following:

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

const App = () => {
  const onSubmit = username => console.log(username);

  return <Form onSubmit={onSubmit} />;
};

const Form = ({ onSubmit }) => {
  const [username, setUsername] = useState('');

  return (
    <form
      onSubmit={event => {
        onSubmit(username);

        // prevents browser from reloading
        // which is the native browser behavior
        // for a form submit
        event.preventDefault();
      }}
    >
      <label>
        Your name:
        <input
          type="text"
          value={username}
          onChange={event => setUsername(event.target.value)}
        />
      </label>

      <button type="submit">Send</button>
    </form>
  );
};

export default App;
```

Note: The Form component uses React Hooks that are not released yet. If you want, you can learn more about React Hooks. Essentially they enable you to have function components with state and side-effects.

Now, wherever we use the Form component, we can capture the username of a user. It's identical to the HTML form from before, isn't it? Not really. At the moment, the Form is only capable of doing one thing. We did loose all the benefits from the HTML element

composition from before, because we ended up with a specialized Form component. It can be reused anywhere in our React application, but it handles only one case. To make it effortless to see the difference, we would have to rename the Form component:

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

const App = () => {
  const onSubmit = username => console.log(username);

  return <UsernameForm onSubmit={onSubmit} />;
};

const UsernameForm = ({ onSubmit }) => {
  const [username, setUsername] = useState('');

  return (
    <form
      onSubmit={event => {
        onSubmit(username);
        event.preventDefault();
      }}
    >
      <label>
        Your name:
        <input
          type="text"
          value={username}
          onChange={event => setUsername(event.target.value)}
        />
      </label>

      <button type="submit">Send</button>
    </form>
  );
};

export default App;
```

Since we encapsulated everything in one React component, it's difficult to tear everything apart to serve another purpose than capturing the name of a user. How do we get back what we had with the HTML form? After all, we don't want to have one specialized form, but a more general applicable form that can be reused for different scenarios.

ENTERING REACT COMPONENT COMPOSITION?

There is one property (React prop) that helps us out with this dilemma for our React component: **the React children prop**. It's one special prop provided by React to render something within a component whereas the component isn't aware ahead of time what it will be. A basic example may be the following:

```
const Button = ({ onClick, type = 'button', children }) => (
  <button type={type} onClick={onClick}>
    {children}
  </button>
);
```

The button element becomes a reusable Button component whereas the Button component doesn't know what it renders except for the button. Let's use the children prop for our previous example to substitute our HTML form element with a Form component that renders all its inner content with React's children prop:

```
...

const UsernameForm = ({ onSubmit }) => {
  const [username, setUsername] = useState('');

  return (
    <Form
      onSubmit={event => {
        onSubmit(username);
        event.preventDefault();
      }}
    >
      <label>
        Your name:
        <input
          type="text"
          value={username}
          onChange={event => setUsername(event.target.value)}
        />
      </label>

      <button type="submit">Send</button>
    </Form>
  );
};

const Form = ({ onSubmit, children }) => (
  <form onSubmit={onSubmit}>{children}</form>
);

...
```

Let's continue with this substitution for the other React elements before we can reap the fruits of having a composable React Form component. The Button component that has been shown before can be used to render our button element:

```
...

const UsernameForm = ({ onSubmit }) => {
  const [username, setUsername] = useState('');

  return (
    <Form
      onSubmit={event => {
        onSubmit(username);
        event.preventDefault();
      }}
    >
      <label>
        Your name:
        <input
          type="text"
          value={username}
          onChange={event => setUsername(event.target.value)}
        />
      </label>

      <Button type="submit">Send</Button>
    </Form>
  );
};

const Form = ({ onSubmit, children }) => (
  <form onSubmit={onSubmit}>{children}</form>
);

const Button = ({ onClick, type = 'button', children }) => (
  <button type={type} onClick={onClick}>
    {children}
  </button>
);

...
```

Last but not least, the input field HTML element and its label. Let's extract it to another reusable React component:

```
...

const UsernameForm = ({ onSubmit }) => {
  const [username, setUsername] = useState('');
```

```
  return (
    <Form
      onSubmit={event => {
        onSubmit(username);
        event.preventDefault();
      }}
    >
      <InputField value={username} onChange={setUsername}>
        Your name:
      </InputField>

      <Button type="submit">Send</Button>
    </Form>
  );
};

const Form = ({ onSubmit, children }) => (
  <form onSubmit={onSubmit}>{children}</form>
);

const Button = ({ onClick, type = 'button', children }) => (
  <button type={type} onClick={onClick}>
    {children}
  </button>
);

const InputField = ({ value, onChange, children }) => (
  <label>
    {children}
    <input
      type="text"
      value={value}
      onChange={event => onChange(event.target.value)}
    />
  </label>
);

...
```

As you can see, the InputField component becomes generic/abstract while all props are passed to the component to specialize it. In addition, the component takes it one step further than the Form and Button components, because it offers a new kind of "HTML element" composition that encapsulates a label with an input field into one component. It can be reused as such in our Form component but anywhere else too.

All three steps from before made our Form a composable React component. The Form renders the HTML form element, but everything within is rendered with React's children. The same applies to the components within the Form component, which follow the same principle of composition in themselves, by just rendering anything that's passed to them with the children property.

GENERALIZATION VS. SPECIALIZATION FOR REACT COMPONENTS

In our case, we have one specialized Form component (UsernameForm) to capture the username information from a user. However, you could also use the Form component directly in the App component. The App component then makes it a specialized Form component by passing all the displayed information as children and other props to it:

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

const App = () => {
  const onSubmit = username => console.log(username);

  const [username, setUsername] = useState('');

  return (
    <Form
      onSubmit={event => {
        onSubmit(username);
        event.preventDefault();
      }}
    >
      <InputField value={username} onChange={setUsername}>
        Your name:
      </InputField>

      <Button type="submit">Send</Button>
    </Form>
  );
};

...
```

The UsernameForm component disappears. In the App component, you take all the ingredients (e.g. Form, InputField, Button), give them your specialized flavor (e.g. onSubmit, username, setUsername), and arrange them however you want them to appear within the Form component. What you get is a composed Form component that is specialized from the outside (App component). Anyway, you can also keep the UsernameForm, if this kind of specialized Form is used more than once in your application:

```
const App = () => {
  return (
    <div>
      <UsernameForm onSubmit={username => console.log(username)} />
      <UsernameForm onSubmit={username => console.log(username)} />
```

```
      </div>
    );
  };

  const UsernameForm = ({ onSubmit }) => {
    const [username, setUsername] = useState('');

    return (
      <Form
        onSubmit={event => {
          onSubmit(username);
          event.preventDefault();
        }}
      >
        <InputField value={username} onChange={setUsername}>
          Your name:
        </InputField>

        <Button type="submit">Send</Button>
      </Form>
    );
  };

  ...
```

From there it really depends on your React application on whether you want to use the generic Form component (e.g. Form) or specialize it as standalone Form component with a special use case (e.g. UsernameForm). My recommendation: Do the latter only if you catch yourself copying and pasting the same generic Form component from A to B to reuse it somewhere else. Then I would advice to implement this specialized Form component which encapsulates all the logic and can be reused anywhere in your application. In addition, it's beneficial for testing your React component in isolation.

FINE-GRAINED PROPS CONTROL

So far, we have only discussed the composable Form component. However, the InputField and Button components are composable components in themselves too. Both of them render something by using the children prop; they are not aware ahead of time what it will be.

Also the generalizaton and specialization applies to these components. The Button component, in our case, is already a specialized case, because it doesn't use the default "button" type, but a "submit" type to make it working in our submit form. Then we don't need to pass the onClick function to the button and therefore the onSubmit from the form element is used instead.

If you would want to add more props to the Button component, you can do so without bothering the other components in the component composition of the Form component. Let's say you want to give your Button component a colored background from the outside:

```
...

const App = () => {
  const onSubmit = username => console.log(username);

  const [username, setUsername] = useState('');

  return (
    <Form
      onSubmit={event => {
        onSubmit(username);
        event.preventDefault();
      }}
    >
      <InputField value={username} onChange={setUsername}>
        Your name:
      </InputField>

      <Button color="violet" type="submit">
        Send
      </Button>
    </Form>
  );
};

...

const Button = ({
  color = 'white',
  onClick,
  type = 'button',
  children,
}) => (
  <button
    style={{ backgroundColor: color }}
    type={type}
    onClick={onClick}
  >
    {children}
  </button>
);

...
```

You can change the Button component's API (arguments = props) at one place, and can use it anywhere in your application. In contrast, imagine how implicit this component API would

become without the composition. Let's take the component from the beginning of this walkthrough. You would have to pass the color to the Form component -- not caring about the generalization/specialization of the Form component here -- to colorize another component within the Form component:

```
const Form = ({ buttonColor, onSubmit }) => {
  const [username, setUsername] = useState('');

  return (
    <form
      onSubmit={event => {
        onSubmit(username);
        event.preventDefault();
      }}
    >
      <label>
        Your name:
        <input
          type="text"
          value={username}
          onChange={event => setUsername(event.target.value)}
        />
      </label>

      <button
        style={{ backgroundColor: buttonColor }}
        type="submit"
      >
        Send
      </button>
    </form>
  );
};
```

However, the Form component shouldn't care about any props for the button element. In order to generalize the shown non composed Form component even more, it would also have to take other props as arguments (e.g. "Your name"-label, "Send"-button label):

```
const Form = ({ label, buttonLabel, buttonColor, onSubmit }) => {
  const [value, setValue] = useState('');

  return (
    <form
      onSubmit={event => {
        onSubmit(value);
        event.preventDefault();
      }}
    >
      <label>
```

```
        {label}
        <input
          type="text"
          value={value}
          onChange={event => setValue(event.target.value)}
        />
      </label>

      <button
        style={{ backgroundColor: buttonColor }}
        type="submit"
      >
        {buttonLabel}
      </button>
    </form>
  );
};
```

Suddenly you would end up with a cluttered component API for the Form component whereas the Form component takes care of everything rendered within itself. That can be avoided and that's why component compositions are such powerful pattern in React. Every component takes care about itself yet contributes to a greater goal in the component hierarchy of a React application.

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

const App = () => {
  const onSubmit = username => console.log(username);

  const [username, setUsername] = useState('');

  return (
    <Form
      onSubmit={event => {
        onSubmit(username);
        event.preventDefault();
      }}
    >
      <InputField value={username} onChange={setUsername}>
        Your name:
      </InputField>

      <Button color="violet" type="submit">
        Send
      </Button>
    </Form>
  );
};

const Form = ({ onSubmit, children }) => (
  <form onSubmit={onSubmit}>{children}</form>
```

```
);

const Button = ({
  color = 'white',
  onClick,
  type = 'button',
  children,
}) => (
  <button
    style={{ backgroundColor: color }}
    type={type}
    onClick={onClick}
  >
    {children}
  </button>
);

const InputField = ({ value, onChange, children }) => (
  <label>
    {children}
    <input
      type="text"
      value={value}
      onChange={event => onChange(event.target.value)}
    />
  </label>
);

export default App;
```



Let's continue with an often seen component composition pattern for React components.

REACT COMPONENT COMPOSITION BY EXAMPLE

You have seen how component composition is mainly used for reusable React components that require a well-designed API. Often you will find this kind of component composition just to layout your application as well. For instance, a SplitPane component, where you want to show something at the left and the right as the inner content of the component, could make use of React props to render more than one child component:

```
const SplitPane = ({ left, right }) => (
  <div>
    <div className="left-pane">{left}</div>
    <div className="right-pane">{right}</div>
  </div>
);
```

Then it could be used the following way in another React component whereas you decide what you render as children in which of both slots:

```
<SplitPane
  left={
    <div>
      <ul>
        <li>
          <a href="#">Link 1</a>
        </li>
        <li>
          <a href="#">Link 2</a>
        </li>
      </ul>
    </div>
  }
  right={<Copyright label="Robin" />}
/>
```

Whereas the Copyright component is just another React component:

```
const Copyright = ({ label }) => <div>Copyright by {label}</div>;
```

This pattern, not widely known under the synonym slot pattern, is used when you have more than one children that you want to compose into another component. Again the component, in this case the SplitPane component, doesn't know ahead of time what will be rendered in there. It just receives something as props to render in these slots. In the example, two slots are used. But it scales up to any number of props you want to pass to the component to render something.

DYNAMIC COMPONENT COMPOSITIONS IN REACT

Often you see something like the following App component whereas React Router is used to compose dynamic components, depending on the selected route (URL), into the Route components:

```
import React from 'react';
import {
  BrowserRouter as Router,
  Route,
```

```
} from 'react-router-dom';

import Navigation from './Navigation';
import LandingPage from './Landing';
import SignUpPage from './SignUp';
import SignInPage from './SignIn';

const App = () => (
  <Router>
    <div>
      <Navigation />

      <hr />

      <Route exact path='/' component={LandingPage} />
      <Route path='/register' component={SignUpPage} />
      <Route path='/login' component={SignInPage} />

      <Footer />
    </div>
  </Router>
);
```

Whereas the Footer component and the Navigation component, which enables navigation from route to route (URL to URL, path to path), stay always the same, the rendered component for the Route components will adjust depending on which URL is currently visited by the user. Basically the App component displays a static frame of components that will be always visible (e.g. Navigation, Footer) whereas the inner content changes depending on the URL. React Router and its Route components take care of it. Keeping it simple, each Route component uses the component prop, to render its content, but only shows it when the matching route is selected.

. . .

In the end, React Component composition is possible foremost due to React's children prop. However, as you have seen you can create your own children prop, or have multiple children props, by defining the props yourself. You can find the example from this tutorial in this GitHub repository.

Composing React Components doesn't end here. There are two other advanced React patterns that are used for component compositions as well:

- Render Prop Components
- Higher-Order Components

React's Render Prop Components can be seen as extension of the shown slot pattern. However, in these slots you wouldn't pass directly what you want to render like in the slot pattern, but rather a function that returns the thing you want to render. By having this function at your disposal, you are able to pass information from the inner component that uses the slot pattern to the components that are used within these slots.

React's Higher-Order Components can be used for Component Compositions too. Basically a Higher-Order Component receives a React component as input and outputs an enhanced version of this component. If you take this one step further, you can also have more than one Higher-Order Component enhancing one component which leads us to a component composition again. Remember the function composition from the beginning? It can be done for React components too.

KEEP READING ABOUT REACT ›

HOW TO USE REACT REF

Using React ref and really understanding it are two different pair of shoes. To be honest, I am not sure if I understand everything correctly to this date, because it's not as often used as state or...

HOW TO USE REACT TESTING LIBRARY TUTORIAL

React Testing Library (RTL) by Kent C. Dodds got released as alternative to Airbnb's Enzyme . While Enzyme gives React developers utilities to test internals of React components, React Testing...