Functions as Children and the Render Prop Patterns

Patterns that entail using JSX expressions within React to abstract code and provide flexibility.

WE'LL COVER THE FOLLOWING

JSX Expressions as Children

Over the last few months, the React community has started shifting in an interesting direction. So far in our examples, the `children` prop was a React component. There is, however, a new pattern gaining popularity in which the same `children` prop is a JSX expression. Let's start by passing a simple object.

```
import React from 'react';

import ReactDOM from 'react-dom';
import App from './app.js';

ReactDOM.render(
  <App />,
  document.getElementById('root')
);
```

This may look weird but in fact is really powerful. Like for example when we have some knowledge in the parent component and don't necessarily want to send it down to children.

The example below prints a list of TODOs. The `App` component has all the data and knows how to determine whether a TODO is completed or not and makes the ones that are bold. The `TodoList` component simply encapsulates the

the ones that are bold. The `TodoList` component simply encapsulates the needed HTML markup.

```
import React from 'react';

function TodoList({ todos, children }) {
  return (
    <section className='main-section'>
      <ul className='todo-list'>{
        todos.map((todo, i) => (
          <li key={ i }>{ children(todo) }</li>
        ))
      }</ul>
    </section>
  );
}

export default TodoList;
```

Notice how the `App` component doesn't expose the structure of the data. `TodoList` has no idea that there is `label` or `status` properties.

The so-called *render prop* pattern is almost the same except that we use the `render` prop and not `children` for rendering the todo.

```
function TodoList({ todos, render }) {
  return (
    <section className='main-section'>
      <ul className='todo-list'>{
        todos.map((todo, i) => (
          <li key={ i }>{ render(todo) }</li>
        ))
      }</ul>
    </section>
  );
}

return (
  <TodoList
    todos={ todos }
    render={
      todo => isCompleted(todo) ?
        <b>{ todo.label }</b> : todo.label
    } />
);
```

These two patterns, *functions as children* and *render prop*, are my favorite ones since recently. They provide flexibility and help when we want to reuse code. They are also a powerful way to abstract imperative code.

```
import React from 'react';
import DataProvider from './dataprovider.js';
```

```
export default class App extends React.Component {
  render() {

    return (
      <DataProvider render={ data => <p>The data is here!</p> } />
    );
  }
}
```

Data Provider

`DataProvider` renders nothing when it first gets mounted. Five seconds later we update the state of the component and render a `<section>` followed by what the `render` prop is returning. Imagine that this same component fetches data from a remote server and we want to display it only when it is available.

```
<DataProvider render={ data => <p>The data is here!</p> } />
```

Authorize Component

We do say what we want to happen but not how. That is hidden inside the `DataProvider`. These days we use this pattern at work where we have to restrict some UI to certain users who have `read:products` permissions. And we used the *render prop* pattern.

```
<Authorize
  permissionsInclude={[ 'read:products' ]}
  render={ () => <ProductsList /> } />
```

This should be self-explanatory. `Authorize` goes to our identity provider and checks what are the permissions of the current user. If he/she is allowed to read our products we render the `ProductList`.

Quick quiz on these Patterns!

1

What is an advantage of the *functions as children* pattern?

COMPLETED 0%

1 of 3

Final thoughts

Have you wondered why HTML is still popular? It was created in the dawn of the internet and we still use it. That is because it’s highly composable. React and its JSX looks like HTML on steroids and as such, it comes with the same capabilities. So, make sure that you master composition because it is one of the greatest benefits of React.