Żimuzo Obiechina

Follow About

Design Patterns in React(Part I): The Container Pattern

Żimuzo Obiechina Feb 8, 2020 · 4 min read

Photo by Nathan Dumlao on Unsplash

I've come to realize that understanding the 'why' of anything (people, languages, libraries) makes it easier to work with and learn them. React is no different. I was watching a tutorial in which an application was being built in React and I kept wondering why some components were written differently from others… There, began my journey into design patterns in React!

Whenever we write code in any programming language, we are (hopefully) creating solutions to solve problems. Problems, just like humans, have behaviour, such that when a variety of problems are taken apart to their smallest parts, you will notice a method to their 'madness'. This is where design patterns come in!

Design patterns can be seen as templates for how to solve problems. These are reusable solutions that can be applied to common problems encountered when writing JavaScript applications in React.

In React, web applications are broken down into components, each component built and then re-coupled to create the application.

In this series, we will be taking a look at four common design patterns in React — the Container pattern, Higher-Order-Component(HOC) pattern, Reducer pattern and the Observer pattern.

. . .

The Container Pattern

Logic refers to anything outside of the UI; for example, API calls, event handlers and data manipulation. *Presentation* refers to the contents of the render method where we create elements to be displayed on the UI.

This solves the problem of a component performing more than one action or task, making it easier to debug and improve code.

Let us take a look at an example that elucidates this concept. In the first part of this example, we will fetch a user from an API and render that user's name and email on the UI, without using the Container pattern.

Let's create a file called profile.js. Inside profile.js, create a class component called Profile and declare the state, with its property values initialized to an empty string.

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

class Profile extends Component {
  state = {
    name: '',
    email: ''
  }
}
```

A class component must have a `render()` method, which renders the required data from the API response, so let's write that:

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

class Profile extends Component {
  state = {
    name: '',
    email: ''
  }

render() {
    return (
      <div>
        <p>Name: {this.state.name}</p>
        <p>Email: {this.state.email}</p>
      </div>
    )
  }
}
```

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

class Profile extends Component {
  state = {
    name: '',
    email: ''
  }

componentDidMount() {
  fetch('https://jsonplaceholder.typicode.com/users/1')
    .then(response => response.json())
    .then(user => this.setState({
      name: user.name,
      email: user.email
    })
  )
}

render() {
    return (
      <div>
        <p>Name: {this.state.name}</p>
        <p>Email: {this.state.email}</p>
      </div>
    )
  }
}
```

As demonstrated above, the Profile component is fetching data and also presenting the data to the UI, which is less than ideal. Now, let's see how the Container pattern fixes this:

First, let's create a container component which will only handle the logic for making the API request. Create a file named profile-container.js, which will contain the container component. Its render method will have all the UI elements removed and replaced by the presentational component.

> *Best practice: Append 'container' to the end of the container component filename and give the original name -profile.js, to the presentational one. The container component is usually a class component, while the presentational component will be a functional component.*

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

import Profile from './profile.js';
```

```
  }

  componentDidMount() {
    fetch('https://jsonplaceholder.typicode.com/users/1')
      .then(response => response.json())
      .then(user => this.setState({
        name: user.name,
        email: user.email
      })
    )
  }

  render() {
      return (
        <Profile {...this.state} />
      )
    }
  }
```

> *The spread operator is a convenient way to pass attributes as opposed to writing each one manually.*

Next, we will create a stateless functional component called Profile, which will contain all our presentational elements, in a file named profile.js.

We will pass in the required data values from the state as props using object destructuring.

```
import React from 'react';

const Profile = ({ name, email }) => (
  <div>
    <p>Name: {name}</p>
    <p>Email: {email}</p>
  </div>
)
```

Finally, in keeping with best practices, let's create validation by using proptypes to make sure that the data received from the API call is valid. This will be done in profile.js:

```
import React from 'react';
import PropTypes from 'prop-types';

const Profile = ({ name, email }) => (
  <div>
    <p>Name: {name}</p>
```

```
Profile.proptypes = {
  name: PropTypes.string,
  email: PropTypes.string
};

export default Profile;
```

Conclusion

Using the Container pattern is especially beneficial in large applications, making the difference when it comes to the speed of development and maintainability of the codebase.

In addition, when working in a team, other developers can improve the container code by adding some error handling logic without affecting the presentation.

> *Thank you for reading this article. Hope you find it helpful? If you are interested in taking a deeper dive into React design patterns, you can check out this resource.*

Happy Coding!

React Design Patterns