Wrapper Components in React

Kamil Ciecierski

Komponenty opakowujące

Komponenty opakowujące to komponenty, które otaczają nieznane komponenty i zapewniają domyślną strukturę do wyświetlania komponentów potomnych. Ten wzorzec jest przydatny do tworzenia elementów interfejsu użytkownika (UI), które są wielokrotnie używane w projekcie, takich jak moduły, strony szablonów i kafelki informacyjne.

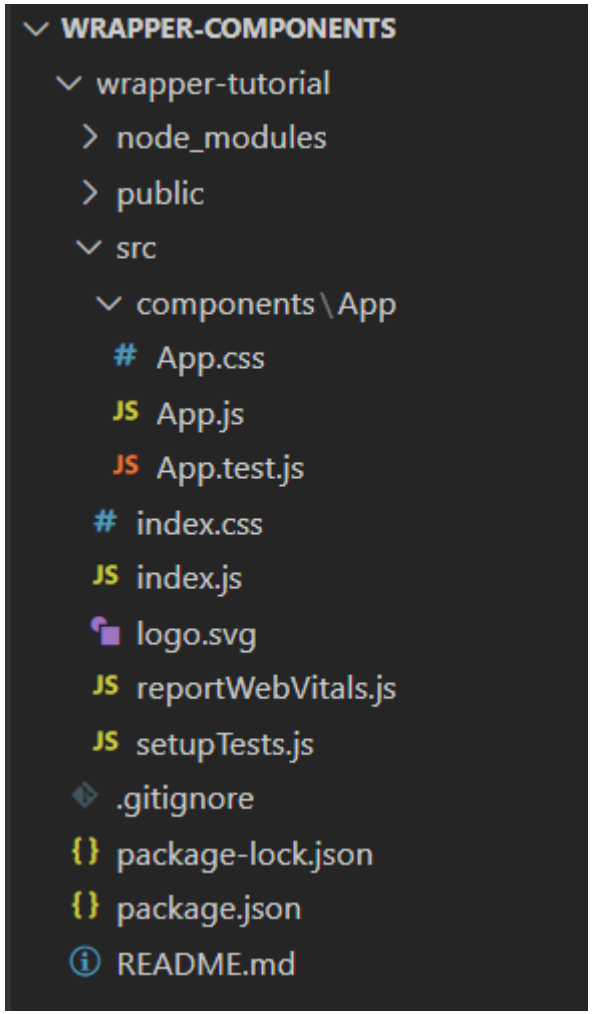
Krok 1 – Tworzenie pustego projektu

Tworzymy nowy projekt, wydając w wierszu poleceń następujące polecenia:

```
npx create-react-app wrapper-tutorial
cd wrapper-tutorial
npm start
```

Utworzenie struktury plików w projekcie

W katalogu ./src/ tworzymy katalog components, który będzie zawierał wszystkie nasze komponenty. Każdy komponent będzie miał swój własny katalog do przechowywania pliku komponentu wraz ze stylami, obrazami oraz innymi plikami, z których korzysta komponent.



Krok 2 – Korzystanie z operatorów rest oraz spread

- W tym kroku utworzymy komponent, który będzie wyświetlał zestaw danych o grupie zwierząt. Nasz komponent będzie zawierał drugi zagnieżdżony komponent, który będzie wyświetlał niektóre informacje w sposób wizualny.
- Aby połączyć komponent rodzica z komponentem zagnieżdżonym, użyjemy operatorów rest i spread do przekazania nieużywanych propsów z komponentu rodzica do komponentu dziecka, bez konieczności znajomości nazw i typów tych propsów przez rodzica.

Przygotowanie pliku z danymi

Na początku musimy utworzyć plik z zestawem danych dla naszych zwierząt.

./src/components/App/data.js

```
export default [
    {
        name: 'Lion',
        scientificName: 'Panthero leo',
        size: 140,
        diet: ['meat']
    },
    {
        name: 'Gorilla',
        scientificName: 'Gorilla beringei',
        size: 205,
        diet: ['plants', 'leaves']
    },
    {
        name: 'Zebra',
        scientificName: 'Equus quagga',
        size: 322,
        diet: ['plants']
    }
]
```

Tworzenie komponentu AnimalCard

Tworzymy katalog dla komponentu AnimalCard.

Następnie dodajemy komponent, który przyjmie: name, diet i size jako argumenty i je wyświetli.

- W tym miejscu dokonujemy destrukcji propsów na liście parametrów funkcji AnimalCard, a następnie wyświetlamy dane w div.
- Dane dotyczące diety są wypisane jako pojedynczy ciąg znaków przy użyciu metody join().
- Każda część danych zawiera odpowiadający jej PropType, aby upewnić się, że typ danych jest prawidłowy.

./src/components/AnimalCard/AnimalCard.js

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

export default function AnimalCard({ diet, name, size })
{
  return(
    <div>
      <h3>{name}</h3>
      <div>{size}kg</div>
      <div>{diet.join(', ')}.</div>
    </div>
  )
}

AnimalCard.propTypes = {
  diet: PropTypes.arrayOf(PropTypes.string).isRequired,
  name: PropTypes.string.isRequired,
  size: PropTypes.number.isRequired,
}
```

Plik App.js

- Teraz musimy zaimportować nasz komponent oraz plik data.js do głównego komponentu App.js
- Używamy metody .map() do iteracji po obiektach listy i zwracamy obiekt z odpowiednimi propsami.

./src/components/App/App.js

```
import React from 'react';
import './App.css';
import animals from './data';
import AnimalCard from '../AnimalCard/AnimalCard';

function App() {
  return (
    <div className="wrapper">
      {animals.map(animal =>
        <AnimalCard
          diet={animal.diet}
          key={animal.name}
          name={animal.name}
          size={animal.size}
        />
      )}
    </div>
  );
}

export default App;
```

Plik App.css

Usuwamy szablonową stylizację i dodajemy własne style.

./src/components/App/App.css

```
.wrapper {
  display: flex;
  flex-wrap: wrap;
  justify-content: space-between;
  padding: 20px;
}
```

Tworzenie komponentu szczegółów

Mamy teraz prosty komponent, który wyświetla dane. Załóżmy jednak, że chcielibyśmy, aby dane o pożywieniu wyświetlały się jako emoji zamiast tekstu.

Możemy to zrobić, konwertując dane w komponencie.

Możemy przekonwertować dane na kilka sposobów:

- Utworzenie funkcji wewnątrz komponentu, która przekształci tekst na emoji.
- Utworzenie i zapisanie funkcji w pliku poza komponentem, aby móc ponownie wykorzystać jej logikę w różnych komponentach.
- Utworzenie osobnego komponentu, który będzie konwertował tekst na emoji.

Tworzenie komponentu AnimalDetails

- Tworzymy katalog dla komponentu AnimalDetails.
- Następnie wewnątrz pliku tworzymy komponent, który będzie wyświetlał właściwość diet w postaci emoji.
- Za pomocą PropTypes określamy typ właściwości diet jako tablicę stringów.

./src/components/AnimalDetails/AnimalDetails.js

```
import React from "react";
import PropTypes from 'prop-types';
import './AnimalDetails.css';

function convertFood(food) {
    switch(food) {
        case 'leaves':
            return '🍃';
        case 'meat':
            return '🍖';
        case 'plants':
        default:
            return '🌱';
    }
}

export default function AnimalDetails({diet}) {
    return(
        <div className="details">
            <h4>Details:</h4>
            <div>
                Diet: {diet.map(food => convertFood(food)).join(' ')}
            </div>
        </div>
    )
}

AnimalDetails.propTypes = {
    diet: PropTypes.arrayOf(PropTypes.string).isRequired,
}
```

Plik AnimalDetails.css

./src/components/AnimalDetails/AnimalDetails.css

```css
.details {
  border-top: gray solid 1px;
  margin: 20px 0;
}
```

Dodanie komponentu AnimalDetails do komponentu AnimalCard

- Teraz, gdy mamy już nowy komponent, możemy go dodać do komponentu AnimalCard.
- Zastępujemy instrukcję diet.join nowym komponentem AnimalDetails i przekazujemy diet jako właściwość, dodając wyróżnione wiersze

./src/components/AnimalCard/AnimalCard.js

```
import React from 'react';
import PropTypes from 'prop-types';
import AnimalDetails from
'../AnimalDetails/AnimalDetails';

export default function AnimalCard({ diet, name, size })
{
  return(
    <div>
      <h3>{name}</h3>
      <div>{size}kg</div>
      <AnimalDetails
        diet={diet}
      />
    </div>
  )
}

AnimalCard.propTypes = {
  diet: PropTypes.arrayOf(PropTypes.string).isRequired,
  name: PropTypes.string.isRequired,
  size: PropTypes.number.isRequired,
}
```

Rest operator

Komponenty dobrze ze sobą współpracują, ale w AnimalCard jest pewna nieefektywność. Wyciągamy dane z argumentu props, ale ich nie używamy. Zamiast tego przekazujemy je do komponentu.

Za każdym razem, gdy chcemy przekazać nowe dane do AnimalDetails, musimy zaktualizować trzy miejsca: App, gdzie przekazujemy propsy, AnimalDetails, który konsumuje propsy, oraz AnimalCard, który jest pośrednikiem.

Lepszym rozwiązaniem jest zebranie wszystkich nieużywanych propsów wewnątrz AnimalCard i przekazanie ich bezpośrednio do AnimalDetails. Daje to możliwość wprowadzania zmian w AnimalDetails bez zmiany AnimalCard. W efekcie AnimalCard nie musi nic wiedzieć o propsach ani o PropTypach, które trafiają do AnimalDetails.

W tym celu należy użyć operatora rest. Operator ten zbiera wszystkie argumenty, które nie zostały wyciągnięte podczas destrukturyzacji i zapisuje je w nowym obiekcie.

Rest operator - przykład

```
function restTest(...args) {
  console.log(args)
}

restTest(1, 2, 3, 4, 5, 6)
```

Output

```
[1, 2, 3, 4, 5, 6]
```

```
function restTest(one, two, ...args)
{
  console.log(one)
  console.log(two)
  console.log(args)
}

restTest(1, 2, 3, 4, 5, 6)
```

Output

```
1
2
[3, 4, 5, 6]
```

Plik AnimalCard.js

- W tym przypadku przekazujemy tylko jeden props do AnimalDetails. W przypadku wielu propsów, kolejność będzie miała znaczenie.
- Później szy props będzie nadpisywał wcześniejsze propsy, więc jeśli mamy props, który ma mieć pierwszeństwo, musimy się upewnić, że jest ostatni.

./src/components/AnimalCard/AnimalCard.js

```
import React from 'react';
import PropTypes from 'prop-types';
import AnimalDetails from
'../AnimalDetails/AnimalDetails';

export default function AnimalCard({ name, size,
...props }) {
  return(
    <div>
      <h3>{name}</h3>
      <div>{size}kg</div>
      <AnimalDetails
        {...props}
      />
    </div>
  )
}

AnimalCard.propTypes = {
  name: PropTypes.string.isRequired,
  size: PropTypes.number.isRequired,
}
```

Plik App.js

Aby zobaczyć, jak obiekt ...props zwiększa elastyczność, przekażmy scientificName do AnimalDetails za pomocą komponentu AnimalCard.

./src/components/App/App.js

```
import React from 'react';
import './App.css';

import animals from './data';
import AnimalCard from '../AnimalCard/AnimalCard';

function App() {
  return (
    <div className="wrapper">
      {animals.map(animal =>
        <AnimalCard
          diet={animal.diet}
          key={animal.name}
          name={animal.name}
          size={animal.size}
          scientificName={animal.scientificName}
        />
      )}
    </div>
  );
}

export default App;
```

Plik AnimalDetails.js

./src/components/AnimalDetails/AnimalDetails.js

```
import React from 'react';
...
export default function AnimalDetails({ diet,
scientificName }) {
  return(
    <div className="details">
      <h4>Details:</h4>
      <div>
        Scientific Name: {scientificName}.
      </div>
      <div>
        Diet: {diet.map(food =>
convertFood(food)).join(' ')}
      </div>
    </div>
  )
}

AnimalDetails.propTypes = {
  diet: PropTypes.arrayOf(PropTypes.string).isRequired,
  scientificName: PropTypes.string.isRequired,
}
```

Krok 3 – React children

W tym kroku stworzymy komponent opakowujący, który może przyjąć nieznaną grupę komponentów jako argument.

React daje nam wbudowaną właściwość o nazwie children, która zbiera wszystkie komponenty potomne. Dzięki temu tworzenie komponentów opakowujących jest intuicyjne i czytelne.

Tworzenie komponentu Card

- Tworzymy komponent, który jako argumenty przyjmuje children i title.
- Wartość children może być albo elementem JSX, albo tablicą elementów JSX.
- Title jest ciągiem znaków.

./src/components/Card/Card.js

```
import React from 'react';
import PropTypes from 'prop-types';
import './Card.css';

export default function Card({ children, title }) {
  return(
    <div className="card">
      <div className="card-details">
        <h2>{title}</h2>
      </div>
      {children}
    </div>
  )
}

Card.propTypes = {
  children: PropTypes.oneOfType([
    PropTypes.arrayOf(PropTypes.element),
    PropTypes.element.isRequired
  ]),
  title: PropTypes.string.isRequired,
}
```

Plik Card.css

./src/components/Card/Card.css

```css
.card {
  border: black solid 1px;
  margin: 10px;
  padding: 10px;
  width: 200px;
}

.card-details {
  border-bottom: gray solid 1px;
  margin-bottom: 20px;
}
```

Plik AnimalCard.js

- Używamy komponentu Card wewnątrz komponentu AnimalCard.
- W przeciwieństwie do innych propsów, nie przekazujemy children w sposób jawny.
- Zamiast tego dołącza się do nich JSX, tak jakby były elementami potomnymi HTML. Innymi słowy, po prostu zagnieżdżamy je wewnątrz elementu.

./src/components/AnimalCard/AnimalCard.js

```
import React from 'react';
import PropTypes from 'prop-types';
import Card from '../Card/Card';
import AnimalDetails from
'../AnimalDetails/AnimalDetails';

export default function AnimalCard({ name, size,
...props }) {
  return(
    <Card title="Animal">
      <h3>{name}</h3>
      <div>{size}kg</div>
      <AnimalDetails
        {...props}
      />
    </Card>
  )
}

AnimalCard.propTypes = {
  name: PropTypes.string.isRequired,
  size: PropTypes.number.isRequired,
}
```

Przekazywanie komponentów jako props

- Mamy teraz komponent Card wielokrotnego użytku, który może przyjmować dowolną liczbę zagnieżdżonych właściwości children. Podstawową zaletą tego rozwiązania jest możliwość ponownego użycia komponentu Card z dowolnym komponentem.
- Wadą używania właściwości children jest to, że można mieć tylko jedną instancję właściwości children. Czasami zdarza się, że chcemy, aby komponent zawierał niestandardowy JSX w wielu miejscach. Możemy to zrobić, przekazując JSX i komponenty React jako propsy.
- Dokonamy modyfikacji komponentu Card, aby mógł przyjmować inne komponenty jako propsy. Dzięki temu nasz komponent będzie mógł wyświetlać nieznane komponenty lub JSX w wielu miejscach na stronie.

Plik Card.js

- Zmodyfikujmy komponent Card tak, aby przyjmował dowolny element React o nazwie details.
- Ten prop będzie miał taki sam typ jak children, ale powinien być opcjonalny. Dlatego dodamy mu domyślą wartość null.

./src/components/Card/Card.js

```
import React from 'react';
import PropTypes from 'prop-types';
import './Card.css';

export default function Card({ children, details, title }) {
  return(
    <div className="card">
      <div className="card-details">
        <h2>{title}</h2>
        {details}
      </div>
      {children}
    </div>
  )
}

Card.propTypes = {
  children: PropTypes.oneOfType([
    PropTypes.arrayOf(PropTypes.element),
    PropTypes.element.isRequired
  ]),
  details: PropTypes.element,
  title: PropTypes.string.isRequired,
}

Card.defaultProps = {
  details: null,
}
```

Plik AnimalCard.js

./src/components/AnimalCard/AnimalCard.js

```
import React from 'react';
...

export default function AnimalCard({ name, size,
...props }) {
  return(
    <Card title="Animal" details={<em>Mammal</em>}>
      <h3>{name}</h3>
      <div>{size}kg</div>
      <AnimalDetails
        {...props}
      />
    </Card>
  )
}
...
```

Plik AnimalCard.js

- Dodaliśmy tylko jeden element JSX, ale możemy przekazać ich dowolną ilość.
- Nie musi to być tylko JSX. Możemy również jako prop przekazać komponent.
- W tym przykładzie jako prop przekażemy komponent AnimalDetails.

./src/components/AnimalCard/AnimalCard.js

```
import React from 'react';
...

export default function AnimalCard({ name, size,
...props }) {
  return(
    <Card
      title="Animal"
      details={
        <AnimalDetails
          {...props}
        />
      }
    >
      <h3>{name}</h3>
      <div>{size}kg</div>
    </Card>
  )
}
...
```

Dziękuję za uwagę!