Props Handling (Advanced)

Learn a few tricks about props handling.

We'll cover the following

Props are passed from parent to child down the component tree. Since we use props to transport information from component A to component B frequently, and sometimes via component C, it is useful to know a few tricks to make this more convenient.

> *Note: The following refactorings are recommended for you to learn different JavaScript/React patterns, though you can still build complete React applications without them. Consider this advanced React techniques that will make your source code more concise.*

React props are a JavaScript object, else we couldn't access `props.list` or `props.onSearch` in React components. Since `props` is an object, we can apply a couple of JavaScript tricks to it. For instance, accessing its properties:

```
const user = {
 firstName: 'Robin',
 lastName: 'Wieruch',
};

// without object destructuring
const firstName = user.firstName;
const lastName = user.lastName;

console.log(firstName + ' ' + lastName);
// "Robin Wieruch"

// with object destructuring
const { firstName, lastName } = user;

console.log(firstName + ' ' + lastName);
// "Robin Wieruch"
```

src/App.js

This JavaScript feature is called object destructuring in JavaScript. If we need to access multiple properties of an object, using one line of code instead of multiple lines is simpler and more elegant. Let's transfer this knowledge to React props in our Search component:

```
const Search = props => {

 const { search, onSearch } = props;


 return (
   <div>
     <label htmlFor="search">Search: </label>
     <input
       id="search"
       type="text"

       value={search}
       onChange={onSearch}

     />
   </div>
 );
};
```

src/App.js

That's a basic destructuring of the `props` object in a React component, so that its properties can be used in the component without the `props` object. We refactored the Search component's arrow function from concise body into block body to access the properties of `props`. This would happen quite often if we followed this pattern, and it wouldn't make things easier for us. We can take it a step further by destructuring the `props` right away in the function signature, omitting the block body again:

```
const Search = ({ search, onSearch }) => (

 <div>
   <label htmlFor="search">Search: </label>
   <input
     id="search"
     type="text"

     value={search}
     onChange={onSearch}
   />
 </div>
);
```

src/App.js

React’s props are rarely used in components by themselves; rather, all the information that comes with the props object used as a container that’s actually used in the component. By destructuring the props object right away in the function signature, we can conveniently access all information without dealing with its container.

Let’s check out another scenario where its less about the props object, and more about an object that comes from it. The same lesson is applicable for the props object. First, we will extract a new Item component from the List component:

```
const List = ({ list }) =>
 list.map(item => <Item key={item.objectID} item={item} />);
const Item = ({ item }) => (
 <div>
   <span>
     <a href={item.url}>{item.title}</a>
   </span>
   <span>{item.author}</span>
   <span>{item.num_comments}</span>
   <span>{item.points}</span>
 </div>
);
```

src/App.js

We applied what we learned in the previous lesson by destructuring the props object in the function signature of each component. The incoming item of the Item component could be seen the same as the props, because it’s never directly used in the Item component. There are three potential ways to handle this problem. The first is to perform a *nested destructuring* in the component’s function signature:

```
// version 1 (final)
const Item = ({

 item: {
   title,
   url,
   author,
   num_comments,
   points,
 },

}) => (
 <div>
   <span>

     <a href={url}>{title}</a>
```

```
    </span>

    <span>{author}</span>
    <span>{num_comments}</span>
    <span>{points}</span>

  </div>
);
```

src/App.js

Nested destructuring introduces lots of clutter through indentations in the function signature. While it's not the most readable option, it can be useful in specific scenarios. On to the second way:

```
// version 2 (I)
const List = ({ list }) =>
  list.map(item => (
    <Item
      key={item.objectID}

      title={item.title}
      url={item.url}
      author={item.author}
      num_comments={item.num_comments}
      points={item.points}

    />
  ));


const Item = ({ title, url, author, num_comments, points }) => (

  <div>
    <span>
      <a href={url}>{title}</a>
    </span>
    <span>{author}</span>
    <span>{num_comments}</span>
    <span>{points}</span>
  </div>
);
```

src/App.js

Even though the Item component's function signature is more concise, the clutter ended up in the List component instead, because every property is passed to the Item component individually. We can improve this technique using JavaScript's spread operator:

```
const person = {
  firstName: 'Robin',
  lastName: 'Wieruch',
};
```

```
};

const details = {
 year: 1988,
 country: 'Germany',
};

const personWithDetails = { ...person, ...details };

console.log(personWithDetails);
// {
//   firstName: "Robin",
//   lastName: "Wieruch",
//   year: 1988,
//   country: "Germany"
// }
```

src/App.js

Instead of passing each property one at a time via props from List to Item component, we could use JavaScript's spread operator to pass all the object's key/value pairs as attribute/value pairs to the JSX element:

```
// version 2 (II)
const List = ({ list }) =>

 list.map(item => <Item key={item.objectID} {...item} />);

const Item = ({ title, url, author, num_comments, points }) => (
 ...
);
```

src/App.js

Then, we'll use JavaScript's rest parameters:

```
const person = {
 firstName: 'Robin',
 lastName: 'Wieruch',
 year: 1988,
};

const { year, ...personWithoutYear } = person;

console.log(personWithoutYear);
// {
//   firstName: "Robin",
//   lastName: "Wieruch"
// }
```

src/App.js

The JavaScript rest operator is the last part of an object destructuring. It shouldn't be mistaken with the spread operator even though it has the same syntax. Usually, the rest operator happens on the right side of an assignment, with the spread operator on the left.

Now it can be used in our List component to separate the `objectID` from the item, because the `objectID` is only used as `key` and isn't used (so far) in the Item component. Only the remaining (rest) item gets spread as attribute/value pairs into the Item component (as before):

```
// version 2 (III/final)
const List = ({ list }) =>

 list.map(({ objectID, ...item }) => (
   <Item key={objectID} {...item} />

 ));
```

src/App.js

While this version is very concise, it comes with advanced JavaScript features that may be unknown to some. The third way of handling this situation is to keep it the same as before:

```
// version 3 (final)
const List = ({ list }) =>
 list.map(item => <Item key={item.objectID} item={item} />);

const Item = ({ item }) => (
 <div>
   <span>
     <a href={item.url}>{item.title}</a>
   </span>
   <span>{item.author}</span>
   <span>{item.num_comments}</span>
   <span>{item.points}</span>
 </div>
);
```

src/App.js

The complete demonstration of above concepts is shown in the code below:

```
 □ □ □□  □   ã□  F   □□  □   □  )□     □   9□  5□  @@  □   °□  n□  □PNG
□
  IHDR   □   □□□   (-□S   äPLTE""""""""""""""""2PX=r□)7;*:>H□¤-BGE□□8do5Xb6[eK□®K□¯1MU9gs3S
□
  IHDR   □   □□□   ×©ÍÊ   □ePLTE"""""""""""""""""""""""2RZN¢¹J□«3R[J□¬)59YÁÞ0KS4W`Q«ÄL□²%+-0JR
?^q÷ñíÛ□i.},□ìsæÝ_TttÔ¾ □1#□□/(ì□-[□□□è`□è`Ì□ÚïÅðZ□d5□□□□?ÎebZ¿Þ□i.Ûæ□□□iqÎ□+1°□}Â□5ù  iç
□
  IHDR        □□   D¤□Æ   □APLTE   """""""""""""""""""""""""""2RZVºÖ_ÔôU·Ñ=r□$()'25]ÎíC□□0LS<o}
□
  IHDR   @   @□□    □·□ì   □:PLTE   """""""""""""""""""""""""""""""""""""""""""""""""""""""""""""
¢ßqÇ8Ù□´□mKË±mÆ¶mÛü·yi!è□ÎªYÏuë ÀÏ_Àï?i÷□ý+ò□□ÄA□|□ù{□□´?¿□_En□).□JËD¤<□
```

```
-ê>Û□º«¢XÕ¢î}ß¨ëÛÑ;□ÃöN´□ØvÅý□Î¸ÿ1 □ë×ÄO@&v/Äþ_□ö\ô□Ç\í.□□½+0□□;□□□!□fÊ□¦´Ó%Â JY·O□Â□'/Å]_□
```

In my opinion, this is the best technique to use when working with other people on a React project. It's not as concise as version 2, but it's more readable, gives the Item component a smaller API surface, and doesn't add too many advanced JavaScript features (spread operator, rest operator).

All these versions have their pros and cons. When refactoring a component, always aim for readability, especially when working in a team of people, and make sure they're using a common React code style.

Exercises

- Confirm the changes from the last section.
- Read more about JavaScript's destructuring assignment.
- Think about the difference between JavaScript array destructuring – which we used for React's `useState` hook – and object destructuring.
- Read more about JavaScript's spread operator.
- Read more about JavaScript's rest parameters.
- Get a good sense about JavaScript (e.g. spread operator, rest parameters, destructuring) and what's related to React (e.g. props) from the last lessons.
- Continue to use your favorite way to handle React's props. If you're still undecided, consider the version used in the previous section.