Props Destructuring and Default Values

(https://github.com/learn-co-curriculum/react-hooks-props-destructuring) (https://github.com/learn-co-curriculum/react-hooks-props-destructuring/issues/new)

Learning Goals

- Use destructuring to access props more easily
- Declare default values for destructured props

Introduction

In this lesson, we'll cover props in further detail and explore how they help us make our components more dynamic and reusable. We'll also talk about *destructuring* and how to give our props *default values*.

Reviewing What We Know

Props allow us to pass values into our components. These values can be anything: strings, objects (including arrays and functions), and so on. They give us the opportunity to make our components more dynamic, and a **lot more** reusable.

For example, say we have a `<MovieCard />` component. A movie has a title, a poster image, and many other attributes (or **prop**-erties!). Let's examine what this `<MovieCard />` component would look like with hardcoded, *static* data vs. dynamic *prop* data. Here's the **static, hard-coded version**:

```
function MovieCard() {
  return (
    <div className="movie-card">
      <img
        src="https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcTn1OTYGz2GDC1XjA9tirh_1Rd571yE5UFIYsmZp4nACMd7CCHM'
        alt="Mad Max: Fury Road"
      />
```

```
        <h2>Mad Max: Fury Road</h2>
        <small>Genres: Action, Adventure, Science Fiction, Thriller</small>
      </div>
    );
```

Passing in props

Mad Max: Fury Road is a ridiculously good movie, but what if we want to render a movie card for another movie? Do we just write another component? No, that would be silly! Instead, we write our components so that they make use of props, which are passed from their parents.

To pass props to a component, you add them as attributes when you render the component, just like adding attributes to an HTML element:

```
const movieTitle = "Mad Max"
<MovieCard title={movieTitle} />
```

The value of the prop is enclosed in JSX curly braces. As we read before, this value can be anything: a variable, inline values, functions, etc. If your value is a hard-coded string, however, you can enclose it in double quotes instead:

```
<MovieCard title="Mad Max" />
```

Armed with that knowledge, let's update `MovieCard` to make it **dynamic** by using props:

```
// parent component
function App() {
  const title = "Mad Max";
  const posterURL =
    "https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcTn1OTYGz2GDC1XjA9tirh_1Rd571yE5UFIYsmZp4nACMd7CCHM";
  const genresArr = ["Action", "Adventure", "Science Fiction", "Thriller"];

  return (
    <div className="App">
      {/* passing down props from the parent component */}
```

```
        <MovieCard title={title} posterSrc={posterURL} genres={genresArr} />
      </div>
    );
  }

  // child component
  function MovieCard(props) {
    return (
      <div className="movie-card">
        <img src={props.posterSrc} alt={props.title} />
        <h2>{props.title}</h2>
        <small>{props.genres.join(", ")}</small>
      </div>
    );
  }
```

Now, does that not look cleaner and more reusable compared to the hard-coded version?

Destructuring Props

Since we know that a React function will only ever get called with one argument, and that argument will be the **props** object, we can take advantage of a modern JavaScript feature called **destructuring** (https://ui.dev/object-array-destructuring/) to make our component even cleaner:

```
function MovieCard({ title, posterSrc, genres }) {
  return (
    <div className="movie-card">
      <img src={posterSrc} alt={title} />
      <h2>{title}</h2>
      <small>{genres.join(", ")}</small>
    </div>
```

```
  );
}
```

In this example, we're **destructuring** the `props` object in the parameter in this function, which will have `title`, `posterSrc`, and `genres` as keys. Destructuring takes the **keys from the props object** and creates **variables with the same names**. That way, in our JSX, we don't have to use `props.whatever` everywhere — the value associated with each key is stored in the corresponding variable, so it can be accessed directly!

Another benefit of destructuring props is that it makes it easier to tell what props a component expects to be passed down from its parent. Consider these two versions of the same component:

```
// Without Destructuring
function MovieCard(props) {
  return (
    <div className="movie-card">
      <img src={props.posterSrc} alt={props.title} />
      <h2>{props.title}</h2>
      <small>{props.genres.join(", ")}</small>
    </div>
  );
}

// With Destructuring
function MovieCard({ title, posterSrc, genres }) {
  return (
    <div className="movie-card">
      <img src={posterSrc} alt={title} />
      <h2>{title}</h2>
      <small>{genres.join(", ")}</small>
    </div>
  );
}
```

Looking at the version without destructuring, we'd have to find all the places where `props` is referenced in the component to determine what props this component expects. Looking at the version with destructuring, all we have to do is examine the function parameters and we can see exactly what props the component needs.

Destructuring Nested Objects

We can also do some more advanced destructuring in cases when our props also contain nested objects. For example:

```
function App() {
  const socialLinks = {
    github: "https://github.com/liza",
    linkedin: "https://www.linkedin.com/in/liza/",
  };

  return (
    <div>
      <SocialMedia links={socialLinks} />
    </div>
  );
}

function SocialMedia({ socialLinks }) {
  return (
    <div>
      <a href={socialLinks.github}>{socialLinks.github}</a>
      <a href={socialLinks.linkedin}>{socialLinks.linkedin}</a>
    </div>
  );
}
```

Since `socialLinks` is an object, we can also destructure it to make our JSX cleaner, either by destructuring in the body of the function:

```
function SocialMedia({ socialLinks }) {
  const { github, linkedin } = socialLinks;

  return (
    <div>
      <a href={github}>{github}</a>
      <a href={linkedin}>{linkedin}</a>
    </div>
  );
}
```

...or by destructuring further in the parameters to our function:

```
function SocialMedia({ socialLinks: { github, linkedin } }) {
  return (
    <div>
      <a href={github}>{github}</a>
      <a href={linkedin}>{linkedin}</a>
    </div>
  );
}
```

How much destructuring you do, and where you do it, is very much a matter of preference. Try it out in some components and see what feels right to you!

Default Values for Props

Let's switch gears here and imagine we are using our application to render a list of hundreds of movies. Let's also assume the data set we have is not always reliable when it comes to the urls of the movie posters.

In this case, we want to make sure our component doesn't render as an utter disaster when the data is incomplete. In order to do this, we can use a **default value** to assign a poster url when either a bad one, or none at all, is provided. For this example, let's use the poster for Max

Headroom as a default, seeing as it is a perfect placeholder:

Instead of passing in that default poster image in case we don't have one, we can tell our `MovieCard` component to use a **default value** if the `posterSrc` prop was not provided. To do this, we can add the default value to our destructured props:

```
function MovieCard({
  title,
  genres,
  posterSrc = "https://m.media-amazon.com/images/M/MV5BOTJjNzczMTUtNzc5MC00ODk0LWEwYjgtNzdiOTEyZmQxNzhmXkEyXkFqcGdeQX\
}) {
  return (
    <div className="movie-card">
      <img src={posterSrc} alt={title} />
      <h2>{title}</h2>
      <small>{genres.join(", ")}</small>
    </div>
  );
}
```

Now, whenever we omit the `posterSrc` prop, or if it's `undefined`, the `MovieCard` component will use this default value instead. That means we don't have to worry about not passing in a poster all the time — the component will take care of this for us!

For example, this version of our component would still display its default image, even though we aren't passing a prop of `posterSrc` from the parent component:

```
function App() {
  const title = "Mad Max";
  const genresArr = ["Action", "Adventure", "Science Fiction", "Thriller"];

  return (
```

```
      <div className="App">
        {/* posterSrc is omitted, so the default value will be used instead */}
        <MovieCard title={title} genres={genresArr} />
      </div>
    );
  }
```

Why Use Default Values

An alternative way we could have handled bad urls would be to have `MovieCard` 's parent component *check* whether the `posterSrc` was valid/present, and then pass some control value as a prop when it renders `MovieCard` . This is not ideal compared to using a default prop within the `MovieCard` component.

Consider the following: in React, we want components to encapsulate the functionality that they *can and should be responsible for*. Should the parent component of `MovieCard` be responsible for managing the assignment of a default movie poster source value? In our example, we think not. It makes more sense for the component that is responsible for rendering the movie information and poster to handle missing data.

Conclusion

By using **destructuring** in our components, we can make the returned JSX elements easier to read, because we don't need to reference `props.whatever` everywhere.

Destructuring also makes it easier to tell what props a component expects, since we can just look at the destructured parameters instead of reading through the whole component and looking for references to the props object.

When we use destructuring, we can provide a **default value** for any prop keys we want, so that if the component doesn't receive those props from its parents, we can still render some default information.

Resources

- **MDN on Object Destructuring (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment#Object_destructuring)**

- **Destructuring Objects blog post (https://ui.dev/object-array-destructuring/)**