React Programming

The Big Nerd Ranch Guide

Loren Klingman & Ashley Parker

React Programming

The Big Nerd Ranch Guide

Loren Klingman & Ashley Parker

Big Nerd Ranch

Home Component

Before you take care of styling your home page, take a step back and look at how you are getting the items onscreen: In `App.js`, you map over the `items` array to create an array of **`Thumbnail`**s.

Normally, `App.js` is responsible only for top-level data and routing to other components. It is better to encapsulate the details of how a component is built, such as how a **`Thumbnail`** gets its image and title, in another file.

In this case, you will create a new component called **`Home`** to render the **`Thumbnail`**s. You will display this component when the user navigates to your home page or to `/`. It will also let you add the styling you need for your home page.

In Visual Studio Code, create the files for the **`Home`** component and its associated CSS stylesheet. Add the styles to `Home.css` first:

Listing 6.9 Adding home styling (`Home.css`)

```
.home-component {
    display: grid;
    grid-template-columns: repeat(auto-fit, minmax(150px, 1fr));
    grid-gap: 32px 10px;
    padding: 32px 20px;
}

.home-component .thumbnail-component:nth-child(3n + 2) div {
    background-color: #9BA699;
}

.home-component .thumbnail-component:nth-child(3n + 3) div {
    background-color: #3F3F40;
}
```

Now, in `Home.js`, import the stylesheet. Then create the **`Home`** functional component, which should return a `<div>` with the `home-component` class name. Export your newly created component with `export default Home;`.

Next, you need access to the `items` array. In the new component, accept the `items` array as a prop. This will emulate fetching data at the top level of your application. (When you do this, ESLint will flag an error related to prop types in this component. You will address this error in a moment. Also, if you are wondering why you do not import `items`, we will explain that at the end of this chapter.)

Then, in `App.js`, copy the code that renders the items:

```
    {items.map((item) => (
      <Thumbnail
        key={item.itemId}
        image={itemImages[item.imageId]}
        title={item.title}
      / >
    ))}
```

Paste this code inside the `<div>` in the **`Home`** component. (ESLint will flag another error related to prop types. You will fix this in a moment as well.)

Finally, import the `itemImages` object directly into the component. This is a static object and not likely to change.

Your **`Home`** component should look like this:

Listing 6.10 Creating the **`Home`** component (`Home.js`)

```
import Thumbnail from './Thumbnail';
import { itemImages } from '../items';
import './Home.css';

function Home({ items }) {
  return (
    <div className="home-component">
      {items.map((item) => (
        <Thumbnail
          key={item.itemId}
          image={itemImages[item.imageId]}
          title={item.title}
        />
      ))}
    </div>
  );
}

export default Home;
```

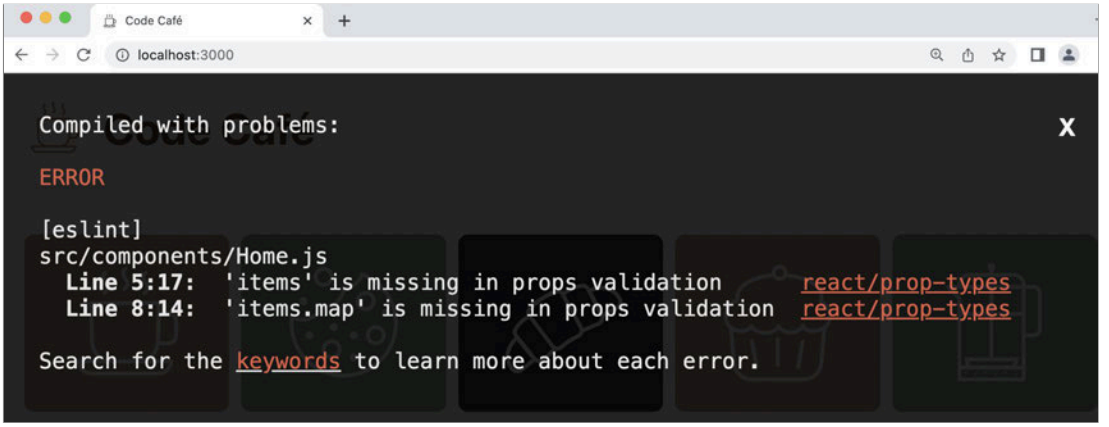
Now that **`Home`** is taking care of rendering the **`Thumbnail`**s, add the new component to `App.js` and delete the code you no longer need:

Listing 6.11 Adding the **`Home`** component (`App.js`)

```
import './App.css';
import Header from './components/Header';
~~import Thumbnail from './components/Thumbnail';~~
import Home from './components/Home';
~~import { items, itemImages } from './items';~~
import { items } from './items';

function App() {
  return (
    <div>
      <Header />
      ~~{items.map((item) => (~~
        ~~<Thumbnail~~
          ~~key={item.itemId}~~
          ~~image={itemImages[item.imageId]}~~
          ~~title={item.title}~~
        ~~/>~~
      ~~))}~~
      <Home items={items} />
    </div>
...
```

So far, you have looked at ESLint errors in your terminal and in Visual Studio Code. ESLint errors also appear in your browser when your app runs in development mode (but not in production). To see what that looks like, open your browser with your app running. An overlay on the screen displays several errors (Figure 6.11).

Figure 6.11 ESLint compilation errors



Though the errors shown are also flagged in Visual Studio Code and your terminal, you might not notice them right away. The eye-catching overlay in your browser is hard to miss when there are problems that need your attention. You can close the overlay by clicking the X in the top-right corner.

The **`Home`** component has only one prop – `items` – but two errors. ESLint marks both the prop itself and the call to **`items.map`** with problem squiggles. What is going on?

You intended for the `items` prop to be an array of objects. But ESLint does not know that, because you have not defined the prop type. So when the **`Home`** component calls **`items.map`**, ESLint has no way to verify that `items` has a method called **`map`**.

Calling an array method on a variable of a different type usually leads to an error. This is a great example of ESLint mitigating potential errors and identifying ways to make your code more robust.

To fix both errors, you need to define `items` as an array.

There are two prop types that identify a prop as an array: `array` and `arrayOf`. Use `PropTypes.array` to define a prop as an array of any type; use `PropTypes.arrayOf` to specify the type of the items inside the array.

To make your code as well documented as possible, use `arrayOf` to define the *shape*, or the underlying structure, of the objects contained in the array.

Listing 6.12 Adding `PropTypes` to the **`Home`** component (`Home.js`)

```
import PropTypes from 'prop-types';
import Thumbnail from './Thumbnail';
...
function Home({ items }) {
  ...
}

Home.propTypes = {
  items: PropTypes.arrayOf(
    PropTypes.shape({
      itemId: PropTypes.string.isRequired,
      imageId: PropTypes.string.isRequired,
      title: PropTypes.string.isRequired,
      price: PropTypes.number.isRequired,
      description: PropTypes.string,
      salePrice: PropTypes.number,
    }),
  ).isRequired,
};

export default Home;
```

You use `PropTypes.shape` to specify the shape of the item object.

Each item contains an `itemId`, `imageId`, title, and price. The price is a number, and all the other fields are strings. All the keys are required.

A couple of the items also contain the properties `description`, which is a string, and `salePrice`, which is a number. These properties are not marked as required, since they are not present in every item.

Save your files and check your browser again. Now that you have defined the types for all your props, you have no more errors.

Reusing Prop Types

You will use the `item` prop type multiple times when building Code Café. To make things easier – and to minimize the risk of errors – you can extract and reuse prop types.

Create a new `src/types` directory. Copy the code for the `item` prop type from **`Home`** and add it to a new file in the `types` folder called `item.js`.

Listing 6.13 Exporting the `item` prop type (`item.js`)

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

const ItemType = PropTypes.shape({
  itemId: PropTypes.string.isRequired,
  imageId: PropTypes.string.isRequired,
  title: PropTypes.string.isRequired,
  price: PropTypes.number.isRequired,
  description: PropTypes.string,
  salePrice: PropTypes.number,
});

export default ItemType;
```

You import `PropTypes` here, just as you did in the component file. You also export `ItemType` so you can use it in your components.

Save your file. Now update the copied code in **`Home`** to use your new variable.

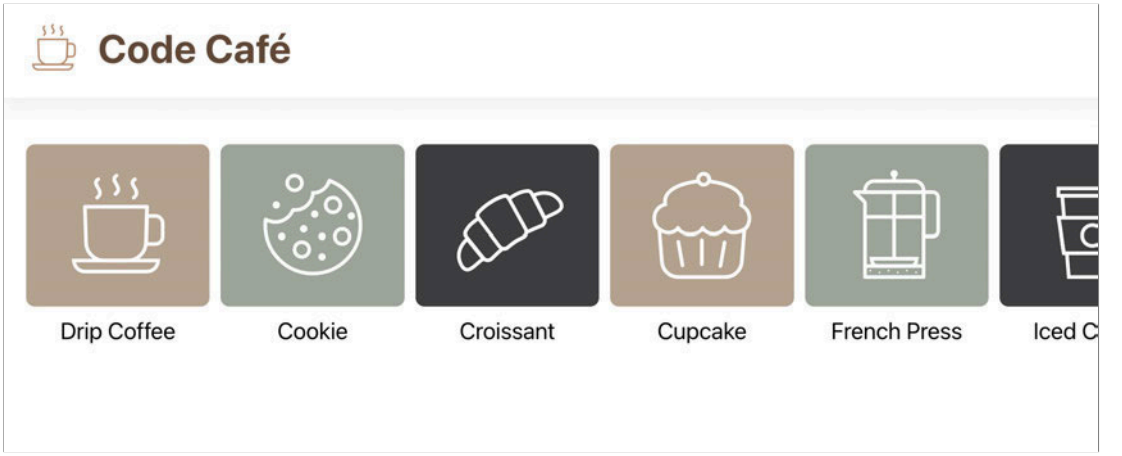
Listing 6.14 Replacing the type of `items` (`Home.js`)

```
...
import './Home.css';
import ItemType from '../types/item';

function Home({ items }) {
  ...
}

Home.propTypes = {
  items: PropTypes.arrayOf(
    PropTypes.shape({
      itemId: PropTypes.string.isRequired,
      imageId: PropTypes.string.isRequired,
      title: PropTypes.string.isRequired,
      price: PropTypes.number.isRequired,
      description: PropTypes.string,
      salePrice: PropTypes.number,
    }),
  ).isRequired,
  items: PropTypes.arrayOf(ItemType).isRequired,
};

export default Home;
```

Now you can use `ItemType` throughout your app, without having to type multiple lines of code and without having to keep the various prop types it contains in sync.

Your code is clean and well organized. Save your files and take a look at Code Café in the browser. Now it is just as well organized as your code (Figure 6.12).

Figure 6.12 Completed components



Prop Mismatch

You have seen what happens if you do not define prop types in a component. But what happens if a component receives props that do not match what it expects?

For example, you just declared that **`Home`** should receive an array of objects that each have specific fields, including `price`. What happens if the price of an item is missing? To find out, open `items/index.js` and comment out the price of an item, such as the cupcake. Then save the file.

ESLint does not immediately complain, and your website will load. Props validation is a runtime check, so your app still compiles (provided there are no other errors as a result of the mismatch). However, if you check the DevTools console, you will see an error warning you of the missing prop:

> Warning: Failed prop type: The prop `items[0].price` is marked as required in `Home`, but its value is `undefined`.

Next, uncomment the price to restore it, but add single quotes around the value, like `price: '1'`. Save the file again. Now the console shows a different warning:

> Warning: Failed prop type: Invalid prop `items[0].price` of type `string` supplied to `Home`, expected `number`.

Console warnings like these can help minimize the time you spend debugging because you forgot to pass a required prop or passed a value of the wrong type.

Before you move on, undo the changes you made to `items/index.js`.