

Intro to React

Code Cafe App

1	GENERATING CODE-CAFE REACT PROJECT	1
2	SETTING UP APP FOLDER STRUCTURE AND RESOURCES	2
3	USING PROP TYPES	2
4	FURTHER STYLING	4
5	INTERACTING WITH A BACKEND SERVER	5
5.1	USING THE USEEFFECT HOOK	6
6	IMPLEMENTING ROUTING IN AN APP	9
7	WORKING WITH CONDITIONAL RENDERING	13
8	USING USEREDUCER HOOK	15
9	EDITING THE CART	17
10	FINAL PART	ERROR! BOOKMARK NOT DEFINED.

1 Generating Code-Cafe React Project

We will generate the React app for this project using Create React App (CRA) in the same way that we have done for all our previous labs.

In the top-level folder that you created for this workshop, open a command prompt and generate a new project using CRA with:

```
create-react-app code-cafe
```

If you already have an existing development server running on your machine from another existing active React project, then you will need to specify a different port for the development server of CRA for this project.

Open a command prompt in this project folder. Here, we use an alternative port of 5000, but you can use any port that is free on your local machine:

```
SET PORT=5000
```

```
npm start
```

The newly generated app should provide the standard default view.

Create another instance of VS Code (File -> New Window) to open this project folder

2 Setting up app folder structure and resources

Folder to use: `code-cafe-app`

File to use: `index-v1.html`

`App-v1.js`

Folder to use: `code-cafe-app/resources`

File to use:

`favicon.ico`

Copy `code-cafe-app/resources/items` to the project `src/items`

Copy `code-cafe-app/resources/images` to the project `src/images`

Create a new `src/components` folder

Create `Header.js` inside here based on `Header-v1.js`

Create `Header.css` inside here based on `Header-v1.css`

We update the header in the browser (`index.html`) and also place our own custom favicon for this app, which should now show up in the browser tab.

Create a separate Header component with its own associated style sheet as well and reference it from the main App component.

Notice that if you view the source content of this app in the Sources panel of Chrome Dev Tools, you will see that the images have been given an additional name extension and bundled into the `media` subfolder. This is result of the bundling activity performed by WebPack, the default static module bundler of CRA, in order to optimize the delivery of these images to the browser at initial load time.

3 Using Prop Types

Folder to use: `code-cafe-app`

File to use: `App-v2.js`

Inside `src/components` folder

Create `Thumbnail.js` inside here based on `Thumbnail-v2.js`

Create `Thumbnail.css` inside here based on `Thumbnail-v2.css`

In the root project folder (`code-café`), installed the `PropTypes` package at the CLI with:

```
npm install prop-types --save
```

Inside `Thumbnail.js`, we use `propTypes` to ensure that the correct type of props (for both `image` and `title`) is passed to this component and that they are both mandatory and required. Notice that the `image` prop is a string type. This is because when an image is imported as a variable, the result is either a string path pointing to the image location or an inline Base64-encoded string, depending on the WebPack configuration.

The file `index.js` in `src/items` is a helper file that provides some boilerplate JavaScript code to create data structure to wrap the images in.

`itemImages` is an object made up of key-value pairs, where the key is the `imageId` and the value is the corresponding image. Importing this object into any file will give you access to all the images – without having to import each one individually.

`items` is an array of the items available for sale. It represents the information that you would normally store in a database, including details about each item such as the title, price, `imageId`, and `itemId`.

These are both imported and utilized by `App.js`. Notice that we are using a string `item.itemId` as the key value of the various `Thumbnail` components in the array. This is possible as there is no requirement for the key value to be numeric, as long as it is unique among all components in the array.

We can further refactor and decompose the current app by using a separate component `Home` to house the list of `Thumbnail` components, instead of rendering all directly in the main `App` component.

Folder to use: `code-cafe-app`

File to use: `App-v3.js`

Inside `src/components` folder

Create `Home.js` inside here based on `Home-v3.js`

Create `Home.css` inside here based on `Home-v3.css`

Create a new `src/types` folder

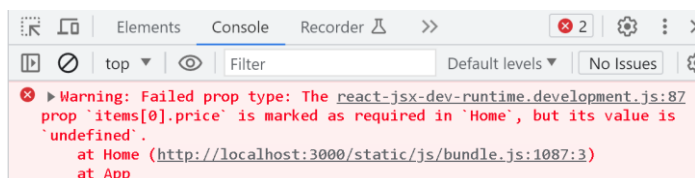
Create `item.js` inside here based on `item-v3.js`

Now we access the `Home` component from the main `App` component.

Notice that the `Home` component also uses `propTypes` to ensure that the correct type of props. 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. Since we are planning to reuse the `PropTypes` definition for the array item structure, we can isolate that to a separate file `item.js` which is used by `Home`.

To see the effect of a type mismatch, open up `items/index.js` and comment out any one of the properties of the objects in the `items` array, for e.g. the `price` property of the first object with `itemId` `coffee`.

Although the app is still rendered, an error message will appear in the Console Panel of Chrome Dev Tools.



Undo the modification you made and verify that the error message is no longer shown in the Console Pane.

4 Further styling

Notice that Home.css uses the Grid layout to provide responsive rows and columns. You can test this out using the features for simulating mobile devices in Chrome Dev Tools to help you with RWD <https://developer.chrome.com/docs/devtools/device-mode/>

Add a random color for `<p>` style rule in `Header.css`, for e.g

```
p {  
  color: red;  
}
```

Notice that the text for all the Thumbnail components (provided by the `<p>` in the JSX returned by that component) has changed color accordingly, even though the CSS style rule was in a style sheet (`Header.css`) imported by the Header component. This demonstrates that CSS style rules in any style sheet apply equally to JSX of all components in the React app.

To ensure that CSS remains specific, it is a good practice to prepend all your CSS class names with component names (such as `.thumbnail-component` and `.home-component`). This naming convention keeps the class names unique and ensures that your styles apply only to specific components.

Change the rule that you previously introduced in `Header.css` to:

```
.header-component p {  
  color: #674836;  
}
```

Notice now that the text for all the Thumbnail components (provided by the `<p>` in the JSX returned by that component) has changed back to their original color, as they are no longer affected by this rule due to the selector change to a class.

Folder to use: `code-cafe-app`

File to use: `App-v4.js`
`Home-v4.css`
`Header-v4.css`

We add in a media query at a width of 768px in both CSS files to specify slightly different values for the key properties related to Grid layout (padding, grid-gap, etc) and also to ensure that the Header remains properly aligned.

Folder to use: `code-cafe-app`

File to use:

Thumbnail-v4.css

We can also add a hover effect to the thumbnail image so that the image is a bit larger when the mouse is over it: we can accomplish this using the `transform` property. We can also use the `transition` property to animate the change in size so that it is gradual and not jarring.

5 Interacting with a backend server

We implement a simple backend app using the Express.js framework (<https://expressjs.com/>). This allows our current React app to retrieve its internal data via HTTP requests sent to the API endpoints exposed by this app, rather than hard coded into the app (as it is currently done now). This reflects the operation of real world frontend Single Page Applications (SPAs) as well, where there is constant interaction with a backend server to retrieve new content to render in the browser view.

Copy `code-café-backend` from `code-café-app` to the top level folder that you are using to create your React projects in. Open a command prompt in the folder and install the dependencies required for this backend server to run with:

```
npm install
```

Then start the server with:

```
npm start
```

The console prints `Listening on http://localhost:3030` to let you know the server is up and running. You can open a browser tab at this location, which will send off a simple HTTP GET request to this backend app which returns a simple response in JSON format that is displayed in the browser view.

You can get a more complete list of items and their related info by accessing this API end point.

<http://localhost:3030/api/items>

You can stop the backend server with `Ctrl+C` and start it up anytime with `npm start` again

By default, all browsers implement CORS (cross-origin resource sharing) policies to protect against potential cross site scripting attacks (XSS). Therefore, the browser will by default block API requests to a different origin. In this policy, even a different port on the same host machine is considered to be different origin and the browser will block it automatically.

To avoid dealing with the complexities of CORS policies, you will set up Code Café to act as a proxy to the back-end server. This way, you will not have to make any allowances for cross-origin sharing. This is a very common requirement in development work, so CRA makes it easy to set this up in the development servers it creates. Open `package.json` in the project folder and add this single line to set up a proxy:

```
"scripts": {  
  "start": "react-scripts start",
```

```
"build": "react-scripts build",
"test": "react-scripts test",
"eject": "react-scripts eject"
},
"proxy": "http://localhost:3030",
"eslintConfig": {
  "extends": [
    "react-app",
    "react-app/jest"
  ]
},
```

The React library itself currently does not include any functionality to create HTTP requests, as it is primarily a UI library. Therefore, for the React app that we are working on to make HTTP requests to the backend server that is currently running, we need to install an additional JavaScript library that uses the browser's built in JavaScript Fetch API to send HTTP requests. There are many popular libraries in the React ecosystem for this purpose:

<https://byby.dev/react-data-fetching-libraries>

We will use the Axios library for our project:

<https://axios-http.com/>

Stop the development server for the React App that is currently running, and in the command prompt in the top level project folder for `code-cafe`, type:

```
npm install --save axios@0.27.2
```

Once this is done, you can restart the server again with `npm start`

5.1 Using the `useEffect` hook

When a user visits Code Café, the app should retrieve the list of items from the server and then render them on the screen. This request should happen only the first time the component renders when the app is initially loaded.

The list of items that we want to retrieve on our initial component load is at:

<http://localhost:3030/api/items>

Folder to use: `code-cafe-app`

File to use: `App-v5.js`

To trigger this call within the app, we use the `useEffect` hook to perform side effects, such as API calls that are not directly triggered by a user event or action

The `useEffect` hook takes two arguments:

- a callback function to be executed
- the dependency array

The dependency array is an array of variables that React uses to determine whether it should execute the function supplied to `useEffect`. To avoid stale data, add any values that you use within `useEffect` to the dependency array.

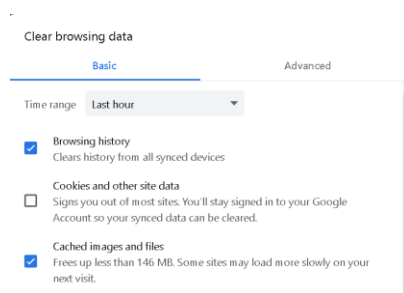
Each time the component renders, React compares the current values of the variables in the array with the values from the previous render. If any value has changed, React executes the function. Otherwise, React skips the function until the next render cycle.

Before we view the sequence of interactions between the React app in your browser and the React development server as well the new backend server, we first remove the data cached by this app from our browsers.

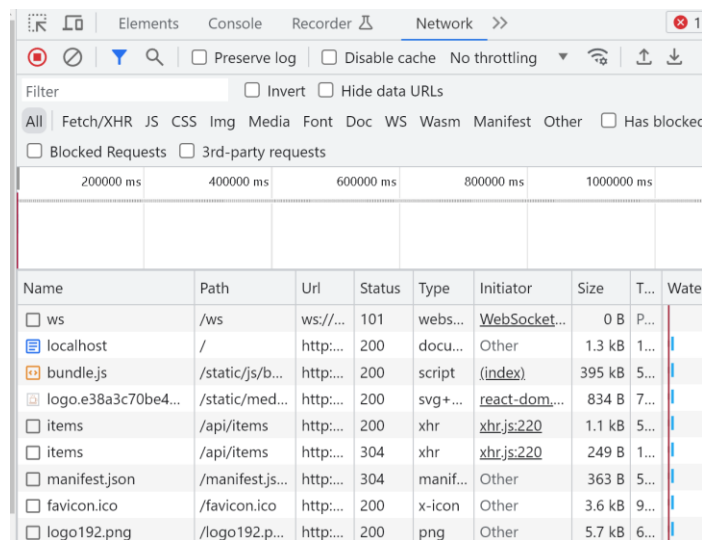
Different browsers have different ways to access the clear cache functionality:

<https://www.bluehost.com/help/article/how-to-clear-your-browser-cache>

You should clear the cache of the browser you are using from the browsing history as well as cached images in the last 1 hour (or 24 hours if you have been working on this lab session for longer than an hour). On Chrome, this is what it should look like:



Go to the Network panel in Chrome Dev Tools and refresh the browser page to reload the React app from the development server. You should now be able to see a sequence of HTTP GET requests sent from the browser to the React CRA Development Server at localhost:3000.



Some points to note:

- The first outgoing request to localhost is typically a WebSocket connection to request a change from the WebSocket protocol (one of the protocols supported by React library) to the HTTP protocol.
- The next request is to retrieve the main HTML of the React app from the CRA Development server (this is the `index.html` in the public folder of your React project). You can verify that

this is the HTML actually returned both in the Response tab of the Network panel and also by selecting View Page Source in the context menu from the browser view.

- The initial HTML file in turn sends out another request for `static/js/bundle.js` to the development server. This file `bundle.js` is produced by WebPack during the bundling process and contains the actual JavaScript code for the React app (together with the supporting React library dependencies).
- The JavaScript code in `bundle.js` subsequently sends off HTTP GET requests to the development server for all the relevant resources that your React app requires (such as `favicon.ico`, `logo192.png`, and all the various images for the café items which are in SVG format)
- All the HTTP requests have a response code of 200 OK, which indicates that the resources requested for was found and successfully returned by the development server
- There are also 2 requests for to <http://localhost:3000/api/items>, which in turn is relayed to the actual backend server running at `localhost:3030`. This URL is formed from the proxy entry in `package.json` and the URL argument to `axios.get` in `App.js`, and the development server functions as a proxy to relay the HTTP GET request to its correct endpoint at the backend server.
- The reason for 2 HTTP requests being sent (instead of only one from the single `axios.get` in `App.js`) is that the React App is running in Strict Mode by default. This is used to ensure that developers write pure component functions (see previous lab explanation on this) and is created by the `<React.StrictMode>` tag in `index.js`. Strict Mode only runs in development mode (not production), but you can turn off this feature by removing those tags in `index.js` and reloading the app again.

Refresh the browser tab for the app a few times.

Notice that the entries in the Network Panel remain the same (indicating the same set of requests are sent to the development server), with the main difference that the status code returned is now 304.

<https://kinsta.com/knowledgebase/http-304>

This is because after the initial load of all the React app resources (images, `bundle.js`, `manifest.json`, etc), these resources are stored in the local browser cache. When you refresh the browser tab, the browser again sends out HTTP GET requests to the development server which in turn informs the browser that nothing has changed on its end (status code 304). This results in the browser retrieving all these resources from its cache to minimize on network latency.

To obtain a code of 200 OK, you will have to clear the cache again (as shown previously), which forces the browser to retrieve the required resources from the development server directly.

Finally, you can check out the CORS policy of a browser that automatically blocks cross-origin HTTP calls that was discussed earlier.

Return to `App.js` and make this modification so that we make a HTTP GET call directly to the backend server:

```
axios.get('http://localhost:3030/api/items')
```

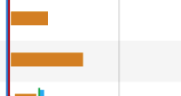
and remove this line:

```
"proxy": "http://localhost:3030",
```

from `package.json` to suspend the proxying functionality of the development server.

Now reload / refresh the app (or stop the development server and start it again).

This time, notice that the HTTP request for <http://localhost:3030/api/items> fails with a CORS error.

<input type="checkbox"/> items	/api/items	http...	COR...	xhr	xhr.js:220	0 B	5...	
<input type="checkbox"/> items	/api/items	http...	COR...	xhr	xhr.js:220	0 B	1...	

and none of the items render in the browser view (because the required data was not successfully retrieved from the backend server).

Restore back your changes and verify that the React app can start properly and render the items correctly as before.

The useEffect Hook and the contents of the state array variable items and the props that receives the state variable in the Home component are also viewable in the React Developer Tools Component Panel. Try experimenting and changing the contents of the state variables to see the result in real time. The Component Panel provides a useful way for you to check out changes to your app quickly by modifying the relevant items.

6 Implementing routing in an app

There are multiple libraries that are available to implement routing functionality in React. Client-side routing is an essential aspect of SPA that React is designed for building, as it allows users to navigate between different views or pages within the initial HTML.

<https://medium.com/geekculture/best-react-routing-libraries-17a27bd302dd>

The most well known and popular library is React Router, which is typically used in combination with CRA.

Stop the development server for the React App that is currently running, and in the command prompt in the top-level project folder for `code-cafe`, type:

```
npm install --save react-router-dom@6.2.1
```

Stop the development server for the React App that is currently running, and in the command prompt in the top level project folder for `code-cafe`, type:

Once this is done, you can restart the server again with `npm start`

If you have any issues in the restart, you may need to reinstall the Axios library again with:

```
npm install --save axios@0.27.2
```

This is due to certain dependencies in certain libraries overriding dependencies of different versions in other libraries. This is one of the major drawbacks with working with React, which requires the installation of many other libraries to augment its basic UI functionality.

Folder to use: `code-cafe-app`

File to use: `App-v6.js`

Your browser URL is currently `localhost:3000/`. We begin implementing routing in your app by adding a route that renders the Home component at the base path, using `/`

A Route component contains two pieces of information: a path and an element. The path specifies the URL, and the element specifies what should render at the URL. In this case, the path is `/` and the element is the Home component. You are still passing items to the Home component. The element prop accepts valid JSX, so passing props to a component inside a Route works the same way as passing props to any other component.

Every Route component must be a direct child of a Routes component. The Routes component's job is to pick which content to display. It analyzes the paths in all its child Route components and picks the one that best matches the URL. The correct element then renders from the matching Route

If you attempt to access a URL with a longer path portion appended behind the base URL `/` in a new browser tab, for e.g.

```
http://localhost:3000/something
```

You will should only see the JSX for the Header component render in the browser view as there is no match for any other component to render based on the current path setting in the single Route component.

Folder to use: `code-cafe-app`

File to use: `App-v7.js`

You can use the `*` operator as a wildcard to match any string.

Now, if you attempt to access a URL with a longer path portion appended behind the base URL `/` in a new browser tab, the simple `<div>` with a simple error message will render.

Next, copy all CSS files from `code-café-app/resources/stylesheet`s into `src/components` of the current Code Café project. This will also override the existing `Header.css` in that current folder.

Folder to use: `code-cafe-app`

File to use: `App-v8.js`

`NotFound-v8.js` (create a new file `NotFound` in `src/components`)

We now create a new component `NotFound` that will render for all invalid paths to replace the basic `<div>` that we created earlier. The Link component from React Router renders as a `<a>` tag but includes extra click event handlers which provides functionality so that the navigation to the new path specified in the Link to props is done using the HTML5 History API rather than a full page refresh (which would require a reload of the initial HTML, and therefore violate the principle of SPA).

Now, if you attempt to access a URL with a longer path portion appended behind the base URL `/` in a new browser tab, for e.g.

```
http://localhost:3000/something
```

You should see the new component render and provide the single Link to return to the base URL representing the home page of the app.

We next create a component that provides a details page for the route to display. The details page will have two sections:

- a sidebar containing all the items for sale
- an area displaying the details for the selected item

Folder to use: `code-cafe-app`

File to use: `App-v9.js`

`Details-v9.js` (create a new file `Details` in `src/components`)

In the main App, we create a nested route inside the main level new `/details` route that maps to the new Details component. This nested route with the path of `":id"` currently maps to simple basic `<div>`

The Details component contains an inner `<div>` which is styled appropriately to provide a sidebar showing all the items for sale using the Thumbnail component that was used in the Home component to display these items on the app home page. It also has an Outlet component from React Router to render the matching child Route element (the basic `<div>` associated with the nested path of `":id"` in the main App component).

Manually type in a URL that looks something like the one below to see the side bar of components and the basic `<div>`

<http://localhost:3000/details/coffee>

Right now, as long as you have another path portion (regardless of what it is) after the `/details`, you will see a page with the side bar and the basic `<div>`
However, if you only type in the first part of the path as in

<http://localhost:3000/details>

The nested route does not render, and only a sidebar with all the café items in full view appears.

Folder to use: `code-cafe-app`

File to use: `App-v10.js`

`DetailItem-v10.js` (create a new file `DetailItem` in `src/components`)

We create a new `DetailItem` component which will provide the JSX for details regarding the selected item. For now, we just provide a basic `<div>` with explanatory text in it.

In the main App, we include the new component `DetailItem` in the nested route. We also add one more nested route under the parent route `/details`. This will be an index route, which corresponds to the situation when the parent route is accessed without any nested route component.

Now, manually typing in a URL that looks something like the one renders the side bar of components and the `DetailItem` component in the center:

<http://localhost:3000/details/coffee>

However, if you only type in the first part of the path as in

<http://localhost:3000/details>

The side bar of components will render along with the `<div>` corresponding to the index route.

Folder to use: `code-cafe-app`

File to use: `DetailItem-v11.js`

The `:id` specified in the path props for the `DetailItem` component is known as a route parameter. Parameters are dynamic information that you can access inside your component.

A parameter in React Router has two parts:

- the colon, which you use to identify a parameter to React Router
- the parameter name (`id`, in this case), which follows the colon

We can then use the `useParams` hook to return an object containing all parameters from the current path

`useParams` returns an object containing a key/value pair, with the value passed to it as the key. In this case, because the route is `/details/:id`, `useParams` returns an object with a key/value pair that has `id` as the key. Using object destructuring, you can immediately get the value of `id` with `const { id } = useParams();`

Now, manually typing in a URL that looks something like the one below renders the side bar of components and the `DetailItem` component in the center, which in turn shows the value of the last part of the path (the `id` parameter)

<http://localhost:3000/details/anythingyouwant>

Folder to use: `code-cafe-app`

File to use: `Header-v12.js`

`Details-v12.js`

`Home-v12.js`

`Thumbnail-v12.js`

In most web apps, there is usually a way to navigate back to the home page from any other page on the website that a user has navigated to. This is usually through a title or logo that remains present on the header regardless of which page the user has navigated to. We can use the `<Link>` component from React Router in the Header component for this purpose.

Finally we also update the Thumbnail component to allow a link to the item details page that we have already constructed. To do this, we need to have an additional `itemId` prop to be passed to the Thumbnail component where it is currently being accessed (in the `Details` and `Home` component) and then subsequently use add this prop to become the final path `/details/itemid` in a Link component in the Thumbnail component itself.

Now at the home page of the app, you can click on any of the item icons to navigate to the page which displays its id in the main page, and then click on the main app logo to return to the home page again.

7 Working with conditional rendering

Conditional rendering lets you specify which parts of a component should render based on the state of your app. This keeps the UI and the application state in sync.

Folder to use: `code-cafe-app`

File to use: `App-v13.js`

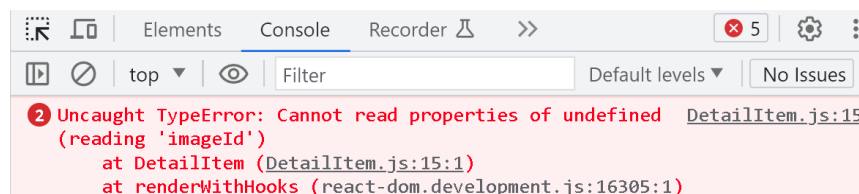
`DetailItem-v13.js`

In the App component, we pass the items array as props to the DetailItem component, where it can access more info related to the item and render this. We use the JavaScript array method `find` to iterate through the list of items and find one with an `itemId` that matches the ID from the parameters. When displaying the item price, we use the `toFixed` JavaScript method that you can call on a number; it returns a string with the specified number of decimal places. It addresses two potential problems when working with prices.

- When you convert a decimal that ends in zero to a string, the final zero is removed, so 7.50 becomes 7.5.
- Because of rounding errors, sometimes a number such as 7.50 ends up displaying as 7.499998.

Notice that if you directly type in a URL such as the one below (instead of navigating to it by clicking on the appropriate icon), an error occurs in the Console Panel of Chrome Dev Tools:

<http://localhost:3000/details/iced-coffee>



The issue is with this specific line of code:

```
src={itemImages[detailItem.imageId]}
```

Here `DetailItem` is located by using the array `find` method on the `items` array. When the browser reloads the app from a manually typed URL, App has to refetch the values for this array from the via the API call at the start of the App component (which in turn retrieves the data from the backend server or from the local browser cache). Until this refetch is complete, `DetailItem` remains undefined. Attempting to access properties of an undefined object results in this error.

We also face a similar issue in the Home component, however the situation here is subtly different. Here we call `map` on an initially empty array item, which will not crash the app, but will just simply not return any Thumbnail components to render until the `item` array is populated

We do not face this issue when navigating to the specified URL from the home page by clicking an image of an item because the images rendering in the browser view already indicates that the `items`

array is already populated correctly with the data returned from the API call at the start of the Home component.

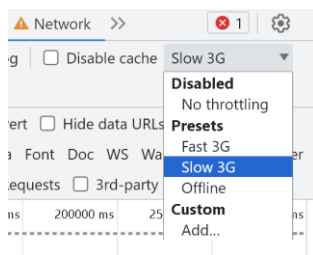
When you refresh the browser through manual navigation, `DetailItem` is initially undefined. You cannot control how long it takes until `DetailItem` has a defined value. But instead of crashing, you can show a loading message while the items requested is being fetch and then render them when they become available.

Folder to use: `code-cafe-app`

File to use: `App-v14.js`

With the change above in `Home.js`, you should see the Loading message appear for a short while at the initial load (when the browser is refreshed at the home page or when a valid URL is typed in manually). This is while the app is waiting for the `items` array to be populated at initial load time, after which all the components are rerendered again due to the change in its state.

To see the temporary display of the Loading message more clearly, you can clear the browser cache (browsing history and cached images), simulate a slow network connection via the Network panel (see below) and then refresh the browser.



JavaScript expressions can be placed within curly braces in the JSX that is returned from a component. However, we cannot use the standard `if-else` statement within the curly braces because it does not directly return values inline. Instead we must use the ternary operator, which can express the logic of a standard `if-else` statement. Besides keeping your code cleaner and easier to maintain, inline operators also allow you to reuse parent elements if needed.

If you now enter a URL with an invalid final path portion, such as:

```
http://localhost:3000/details/lalala
```

You will still obtain the same runtime error as previously. This is because even with the ternary operator present to delay rendering until the `items` array is correctly populated, the `find` method will fail because there is no such id as `lalala` resulting in the `find` method failing and the `detailItem` object still being empty.

To resolve this issue, we now need to also add a ternary operator to `DetailItem.js`

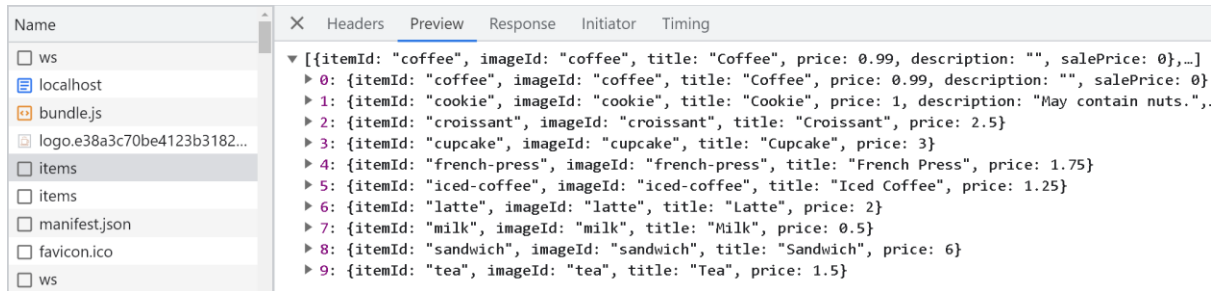
Folder to use: `code-cafe-app`

File to use: `DetailItem-v14.js`

Now, `DetailItem` will either return a message about an unknown item or display the relevant item details, depending on the value in `detailItem`. You can test this out with the previous invalid URL:

<http://localhost:3000/details/lalala>

Currently the items in the array returned from the API calls are objects with properties, and some of these objects have a description and salePrice property while others don't. Thus neither of these properties are currently displayed in DisplayItem component. You can check out the content of these items in the Network panel for the API call to <http://localhost:3000/api/items>



We can again use conditional rendering to display these properties for the case of items that have them.

Folder to use: code-cafe-app

File to use: DetailItem-v15.js

Select the cookie Item to see an additional description for it (May contain nuts) that is not present for all the other items.

We can use the AND && operator for checking whether to display the description of the item, whereby && returns either the first falsy value it encounters or, if all values are truthy, the last value of the statement.

<https://developer.mozilla.org/en-US/docs/Glossary/Truthy>

<https://www.sitepoint.com/javascript-truthy-falsy/>

For the case of item with both a sale price and a normal price, we would only want to display the sale price. For this purpose we can use the logical OR operator, represented by ||, to display the sale price or the regular price. If the code preceding || evaluates to a truthy value, JavaScript returns that value, and React ignores the rest of the statement. If the code evaluates to a falsy value, JavaScript returns the code following ||, and React renders its value.

For items with no sale price, the value of detailItem.salePrice is undefined, and React renders the second value of the statement, the price. When salePrice is 0, is is considered falsy, and React will also render the price instead (for the case of the coffee item).

8 Using useReducer hook

The useReducer hooks is an alternative to the useState hook to handle complex state management. With the useState hook, you have access to one method to update the value of the state variable. Each time you use that method, you overwrite the entire state value. When you need to interact with the state in complex ways, such as when the state represents the shopping cart on an e-commerce site, the code can get rather complex. As an alternative to this, useReducer allows you to write your

own reducer function, which can contain a variety of actions to perform on the state value. This way, the reducer handles all the logic needed for the cart actions in one place

Folder to use: `code-cafe-app`

Create a new `src/reducers` folder

Create `cartReducer.js` inside here based on `cartReducer-v16.js`

Other files to use: `App-v16.js`

The `useReducer` hook accepts two arguments: a reducer function and an initial state value. To keep your code organized, you will create these in a separate reducer directory and file. The cart state will be an array of item objects, where the key is the `itemId` and the value is the quantity of that item in the cart.

You pass the `useReducer` hook the two values you just created – the reducer function and the initial state value. `useReducer` returns an array with two values, which you access using array destructuring. The first value of the returned array is the current state value (`cart`), and the second value is a function that is used to update the state value (`dispatch`).

The action type `add` inserts the `itemId` in the array and sets the quantity to 1 if it does not already exist, otherwise we will need to increment the item quantity by 1 instead if it already exists (we use the `findItem` method for that purpose).

One slight tweak we can make is to replace strings with `const` variables of the same name so that we can take advantage of the editor's code completion tools to reduce the likelihood of syntax errors.

Folder to use: `code-cafe-app`

File to use: `App-v17.js`

`Header-v17.js`

We will add a shopping cart icon to the header, along with the number of items in the cart. To do this, we first pass the cart state object to Header as a prop.

Folder to use: `code-cafe-app`

File to use: `App-v18.js`

`DetailItem-v18.js`

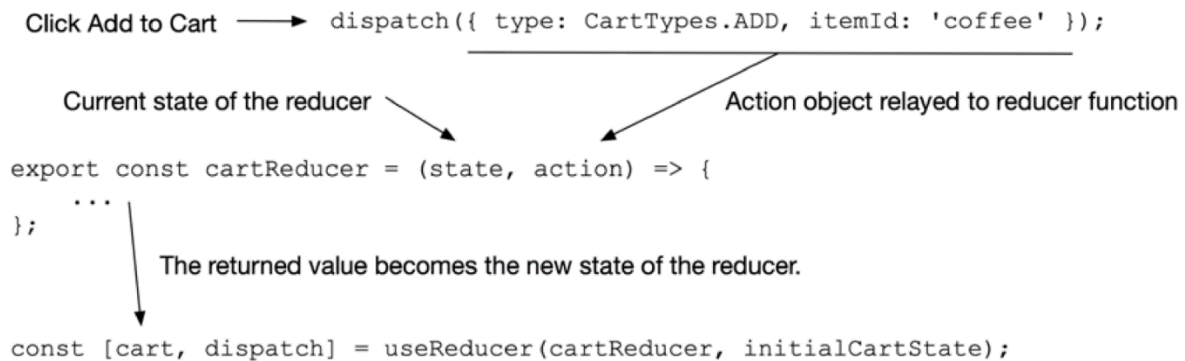
`Header-v18.js`

The user should be able to add items to the cart from a button on each item's details page. For this to happen, `DetailItem` needs access to the `dispatch` function, which we will pass to it as a prop from `App.js`. Within `DetailItem`, we can add a new Add to Cart button that calls the `dispatch` function when clicked, passing it an object containing the action type and the `itemId`.

We can use the `Array.reduce` method in `Header.js` to evaluate the quantity of items in the cart for the `cartQuantity` variable. The `reduce` method allows you to reduce the data in an array to a single value – in this case, the total number of items in the cart.

Experiment with selecting items and adding them to the cart, verifying that the number of selected items show up correctly on the cart icon in the Header and also checking the output in Console panel.

Internally, `useReducer` then calls `cartReducer`, passing it the current state of the reducer as the first argument and the object passed to `dispatch` as the second argument. Finally, the value that `cartReducer` returns becomes the new state of your reducer



Folder to use: `code-cafe-app`

File to use: `App-v19.js`

`DetailItem-v19.js`

One disadvantage of `dispatch` is that it is hard to know what other properties need to be in the action object besides the `type`. For example, although the `ADD` action requires the `itemId`, another developer might not know this. Action creators are a pattern you can use with `useReducer` to help solve this issue. They also help reduce coupling between the reducer and the component using the reducer.

Action creators are essentially helper functions that call the `dispatch` function with a specific action and the other necessary properties. Instead of calling `dispatch` directly, a component instead calls the action creator

The code should work with the same functionality after this refactoring.

9 Editing the cart

Folder to use: `code-cafe-app`

Create a new `Cart.js` in `src/components` based on `Cart-v20.js`

Other files to use: `App-v20.js`

`Header-v20.js`

We add a new route from the main App to the Cart page. We also provide a way to directly link to this route from the cart icon in the Header.

Verify this by adding in new items and clicking on the cart icon to navigate to this basic page.

Folder to use: `code-cafe-app`

Files to use: `Cart-v21.js`

`App-v21.js`