Conclusion

20

Chris Dobson

- Introduction
- Retrieving the Data
- Viewing the Data
- Conclusion
- Top ^

20

Building a Simple React Weather App

Chris Dobson

Aug 6, 2020 • 13 Min read • 3,412 Views

Aug 6, 2020 • 13 Min read • 3,412 Views

Web Development Front End Web Dev... Client-side Framew... React

Introduction

Making calls to a third-party API and using the returned data to drive a React view is something a web developer will need to do many times.

This guide will show how to build a React app that shows users the current weather anywhere in the world. The weather API used in the

Conclusion

20

- Introduction
- Retrieving the Data
- Viewing the Data
- Conclusion
- Top ^

guide is the Open Weather Map API; however, the concepts are equally relevant to using any API.

20

e app will allow the user to add a panel, type in a location for each panel, and retrieve the current weather in the location. Any number of panels can be added and the locations will be stored in browser `localStorage` and retrieved whenever the app is reloaded. The source for the app can be found here, (https://github.com/ChrisDobby/react-simple-weather-app) along with details of how to set up the weather API. The app behaves like this:

Retrieving the Data

- Introduction
- Retrieving the Data
- Viewing the Data
- Conclusion
- Top ^

The most important part of this app is the weather information. To get the weather for a particular location, a `GET` request is sent to weather API, which then returns the weather information as ON. The function to do this looks like this:

javascript

```
async function getLocationWeather(location) {
  const result = await fetch(`https://api.openweathermap.org/data/2.5/weat
  return result.json();
}
```

and can be consumed like this:

javascript

```
await getLocationWeather("London");
```

The above implementation of `getLocationWeather` is very naive in that it assumes that the API will be running, the entered location will always be found, and there will be no errors in the API. Instead of simply returning the `json()` result from the function, you can check the status of the call to `fetch` and return an appropriate result:

javascript

```
const result = await fetch(
`https://api.openweathermap.org/data/2.5/weather?q=${location}&appid=${pro
```

Conclusion

20

- Introduction
- Retrieving the Data
- Viewing the Data
- Conclusion
- Top ^

```
    if (result.status === 200) {
      return { success: true, data: await result.json() };
    }

    return { success: false, error: result.statusText };
```

20

This tests the `status` property of the result. If it's `200`—the http status code for OK—then the API call has succeeded and you can return the result of the call to `json()`. However, if the `status` isn't `200`, then the API call has failed for some reason and the `statusText` property can be returned as an error description. So that a consumer of this function can simply identify whether the call was successful or not, the function returns a `success` property: if `success` is `true` then the result will have a `data` property—which will be the weather data for the location—and if `success` is `false`, then the result will have an `error` property that is a description of the error. For this fairly simple app, the error description is simply the `statusText`, but in a real world app it should be a more user-friendly description.

Finally, the `getLocationWeather` function should wrap the call to `fetch` in a `try...catch` block in case an exception is thrown. If an exception is caught, `success` should be `false` and the error description will be the exception message text but, as above, in a

Conclusion

20

- Introduction
- Retrieving the Data
- Viewing the Data
- Conclusion
- Top ^

real-world app, this should be more user friendly. The function ends up like this:

20

javascript

```
async function getLocationWeather(location) {
  try {
    const result = await fetch(`https://api.openweathermap.org/data/2.5/v

    if (result.status === 200) {
      return { success: true, data: await result.json() };
    }

    return { success: false, error: result.statusText };
  } catch (ex) {
    return { success: false, error: ex.message };
  }
}
```

The consumer of `getLocationWeather` can then test the `success` property and show the weather data or an error appropriately. This will be dealt with in the next section.

Viewing the Data

The UI for the demo app has been built using Material UI, but whether using this, a different UI library, or no UI library at all, the

- Introduction
- Retrieving the Data
- Viewing the Data
- Conclusion
- Top ^

concepts for building the view will be the same.

The `App` component is the root component for the app and stores the currently visible locations in component state as an array of `string`. When initializing the state, the last used list of locations is retrieved from `localStorage` and stored as the state like this:

```javascript
const [weatherLocations, setWeatherLocations] = React.useState(readFromLo
```

The `readFromLocalStorage` function checks whether there are any locations in `localStorage`. If there are, it returns them, and if not, it returns an empty array.

This component also includes a helper function—`updateLocations`—that accepts an array of string as a parameter and both writes the array into `localStorage` and sets the weather location's state. As long as any updates to locations go through this function, then `localStorage` and state will be kept synchronized:

```javascript
const updateLocations = locations => {
  setWeatherLocations(locations);
  saveToLocalStorage(locations);
};
```

Conclusion

20

- Introduction
- Retrieving the Data
- Viewing the Data
- Conclusion
- Top ^

The structure of the view looks like this:

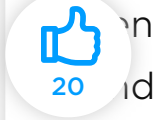
jsx

```jsx
>
      <AppBar position="static">
        ...
      </AppBar>
      <Grid>
        {weatherLocations.map((location, index) => (
          <Grid key={location} xs={12} sm={6} md={4} lg={3} item>
            <WeatherCard
              location={location}
              canDelete={!location || canAddOrRemove}
              onDelete={removeAtIndex(index)}
              onUpdate={updateAtIndex(index)}
            />
          </Grid>
        ))}
      </Grid>
      <Fab
        onClick={handleAddClick}
        color="secondary"
        disabled={!canAddOrRemove}
      >
        <AddIcon />
      </Fab>
    </div>

```

20

The `Fab` component is a Material UI button, which when clicked will call `handleAddClick`, which simply adds an empty location string to the state `setWeatherLocations([...weatherLocations, ""])`.



- Introduction
- Retrieving the Data
- Viewing the Data
- Conclusion
- Top ^

The view for a location is the responsibility of the `WeatherCard` component. If the `location` prop being viewed is an empty string, then this location has been newly added and the `WeatherCard` will render a component allowing the user to enter a location. If the location has already been entered—`location` prop is not an empty string—then the `LocationWeather` component is rendered, which is responsible for retrieving the weather for a location and displaying it inside the `WeatherCard`.

LocationWeather Component

This component accepts a single prop—`location`—and stores two states: one for the weather data that has been retrieved and the other for any error message that was returned from the API.

To retrieve the weather data, you can use the effect hook with a parameter of the `location` prop; this means the side effect will be called whenever the `location` prop changes, which in this app is when the component is mounted. The code looks like this:

javascript

```
React.useEffect(() => {
  const getWeather = async () => {
    const result = await getLocationWeather(location);
    setWeatherData(result.success ? result.data : {});
    setApiError(result.success ? "" : result.error);
  };

```

```
        getWeather();
    }, [location]);
```

Conclusion

- Introduction
- Retrieving the Data
- Viewing the Data
- Conclusion
- Top ^

Because the function passed into `useEffect` cannot be an `async` function, an async inline function is declared inside the effect function—`getWeather`—which is called (but not awaited) by the effect. This function calls the `getLocationWeather` function and, depending on the `success` property of the result, sets the weather data and error states; if the API call was successful, then the weather data is set to the result and error description to an empty string, and if the call failed, then weather data is set to an empty object and the error description to the `error` property. The component can then be rendered.

Currently, while the app is waiting for the API call to return, the component will show nothing. If the call returns quickly, then this is fine; however, if the app is running on a slow network or the API is running slowly, then leaving the component view blank is not a very good user experience. To improve this experience, show a loading indicator if the API call takes longer than a specific length of time. To do this, wadd a new state to the component that will be set to `true` when you want to display a loading indicator:

javascript

```
    const [isLoading, setIsLoading] = React.useState(false);
```

- Introduction
- Retrieving the Data
- Viewing the Data
- Conclusion
- Top ^

20

Then, inside `useEffect`, set a timeout that will call `tIsLoading(true)` after 500ms, meaning that 500ms after `tLocationWeather` is called, a loading indicator will be shown. To ensure the indicator isn't shown if the API call has taken less than 500ms, the timeout should be cleared after the call has completed. The timeout should also be cleared in the return of the effect so that the component doesn't create a memory leak by updating state after a component has been dismounted. So the final `useEffect` will look like this:

javascript

```
React.useEffect(() => {
  const loadingIndicatorTimeout = setTimeout(() => setIsLoading(true), 5(
  const getWeather = async () => {
    const result = await getLocationWeather(location);
    clearTimeout(loadingIndicatorTimeout);
    setIsLoading(false);
    setWeatherData(result.success ? result.data : {});
    setApiError(result.success ? "" : result.error);
  };

  getWeather();
  return () => clearTimeout(loadingIndicatorTimeout);
}, [location]);
```

The view for this component looks like this:

Conclusion

20

- Introduction
- Retrieving the Data
- Viewing the Data
- Conclusion
- Top ^

jsx

```jsx
<div>
      <LoadingIndicator isLoading={isLoading} />
      <ErrorMessage apiError={apiError} />
      <WeatherDisplay weatherData={weatherData} />
    </div>
```

20

The `LoadingIndicator` and `ErrorMessage` components simply display a spinner and error text, respectively, and the `WeatherDisplay` component transforms the data from the API and displays it in a view.

WeatherDisplay Component

Finally, take the data from the weather API and show it to the user. The data returned by the OpenWeatherMap API can be found here. This app will show the temperature, weather icon, wind speed, wind direction, and a description of the weather.

`WeatherDisplay` accepts one prop—`weatherData` in the OpenWeatherMap format—and inside a useMemo hook will take this prop and transform it into an object that can be displayed. The code to do this is:

javascript

```javascript
const { temp, description, icon, windTransform, windSpeed } = React.useMe
  const [weather] = weatherData.weather || [];
```

Conclusion

20

- Introduction
- Retrieving the Data
- Viewing the Data
- Conclusion
- Top ^

```
    return {
      temp: weatherData.main && weatherData.main.temp ? Math.round(weather[
      description: weather ? weather.description : "",
      icon: weather ? `http://openweathermap.org/img/wn/${weather.icon}@2x.
      windTransform: weatherData.wind ? weatherData.wind.deg - 90 : null,
      windSpeed: weatherData.wind ? Math.round(weatherData.wind.speed) : 0,
    };
  }, [weatherData]);
```

20

For everything except the `windTransform` property, this code checks that the required properties exist in the `weatherData`. If they do, return them; otherwise, set a blank or empty value. The `windTransform` property will be used to create a css transform on a right arrow icon. Therefore, if a wind direction has been returned in the `wind.deg` property, then it needs to be reduced by 90 degrees. The code for the view looks like this:

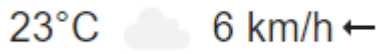
jsx

```
<>
  {temp && <Typography variant="h6">{temp}&deg;C</Typography>}
  {icon && (
    <Avatar className={classes.largeAvatar} alt={description} src={icon}
  )}
  {windSpeed > 0 && (
    <>
      <Typography variant="h6">{`${windSpeed} km/h`}</Typography>
      {windTransform !== null && (
        <ArrowRightAltIcon style={{ transform: `rotateZ(${windTransform}(
      )}
    </>
```

- Introduction
- Retrieving the Data
- Viewing the Data
- Conclusion
- Top ^


```
13          )}
14        </>
```


nich will render a view like this:



Conclusion

This guide has shown how to use the React hooks API to create an app that uses a third-party API to retrieve data.

The source code for the demo app can be found here.