**How To Create a Weather App with React (OpenWeatherMap API)**

By [Jarrett Retz](https://rapidapi.com/blog/author/jarretretz/) // May 28, 2021

**Table of Contents**[[hide](https://rapidapi.com/blog/weather-app-react/)]

* [What Is React (ReactJS)?](https://rapidapi.com/blog/weather-app-react/#what-is-react-reactjs)
* [Weather APIs](https://rapidapi.com/blog/weather-app-react/#weather-apis)
* [How To Build a Weather App with React](https://rapidapi.com/blog/weather-app-react/#how-to-build-a-weather-app-with-react)
  + [Prerequisites](https://rapidapi.com/blog/weather-app-react/#prerequisites)
  + [Getting Started](https://rapidapi.com/blog/weather-app-react/#getting-started)
* [Choosing a Weather API](https://rapidapi.com/blog/weather-app-react/#choosing-a-weather-api)
* [Fetching Data](https://rapidapi.com/blog/weather-app-react/#fetching-data)
* [Displaying Data](https://rapidapi.com/blog/weather-app-react/#displaying-data)
* [Add User Input](https://rapidapi.com/blog/weather-app-react/#add-user-input)
* [CSS](https://rapidapi.com/blog/weather-app-react/#css)
* [Error Handling and Loading](https://rapidapi.com/blog/weather-app-react/#error-handling-and-loading)
* [Security](https://rapidapi.com/blog/weather-app-react/#security)
* [Logo](https://rapidapi.com/blog/weather-app-react/#logo)
* [What’s Next?](https://rapidapi.com/blog/weather-app-react/#what8217s-next)

What Is React (ReactJS)?

At first glance, React is a front-end JavaScript library created by Facebook that is used to create user-interfaces (UI). If you’re [new to React](https://rapidapi.com/blog/how-to-use-an-api-with-react/), or programming, it can seem slightly confusing, but after a few iterations, anyone can catch fire using it. I am going to give you the quick details then show you how easy it is to make something useful.

ReactJS is frontend JavaScript, so the code is run on the client (i.e browser). The backend code doesn’t leave the server. When a URL is requested, frontend code leaves the server, and if someone opened up their developer tools, they could see that code. Remember this for later.

[**View the Best Weather APIs List**](https://rapidapi.com/category/Weather)

When React hits the client, it takes control of the DOM (Document Object Model) and controls what elements are rendered to the webpage. Handling the render process in this way can do wonders for the user experience (UX), but can be complex for developers as they go deeper in the code. Thankfully, React has some pillars of guidance to keep developers on the right track.

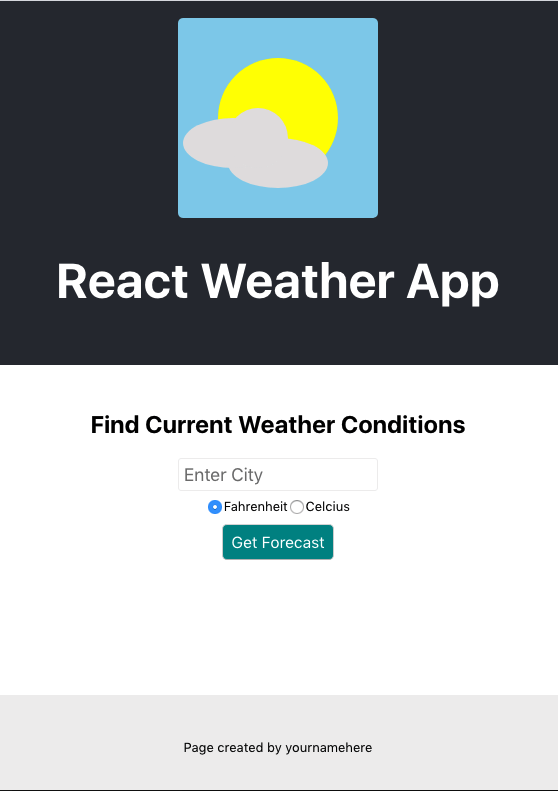
One of the important concepts that I kept in my mind when I first started writing with React is that data flows down. It didn’t click right away, but components, the building blocks in React want to be reused and want to respond to data. Either by receiving the data, or passing it along.

Whenever there is an HTML element that displays—or responds—to data there is an opportunity to create a reusable component. I called it HTML, but you may know that React uses JSX. It looks a lot like HTML but it:

* Has self-closing tags
* Can pass customs props
* Coexists with JavaScript, and
* Uses ‘camelCase’ for attributes and event-listeners.

I mentioned components, too. There are class-based components and functional components. More recently, as of React 16.8, Hooks were introduced. Basically, they are functional components that can hold state. Many times, developers would create function components, but down the road, they needed that component to hold state. Hooks didn’t replace class components, they just added another flexible tool to React. Hooks are new, so we are going to be using them in our app.

We are going to combine React, Hooks, props, JSX, and a weather API to build a stylish front-end application.

The final product.

[**Connect to the OpenWeatherMap API**](https://rapidapi.com/community/api/open-weather-map)

Weather APIs

There are many [different types of APIs](https://rapidapi.com/blog/types-of-apis/). One of the most common types, and some of the easiest to use, are [weather APIs](https://rapidapi.com/blog/access-global-weather-data-with-these-weather-apis/). Weather APIs pair nicely with [geolocation APIs](https://rapidapi.com/blog/ip-geolocation-api/) and some actually use [geolocation behind the scenes](https://rapidapi.com/blog/geolocation-backend-node-express/). Most give back easy to understand data that can be impressively cheap to access.

Our app is going to make an HTTP call to a weather API to retrieve data whenever a user enters a city into a form. We are going to use conditional logic, styling, spinners, and error handling to improve the user experience as we wait for that HTTP call to return our data. Before we can get started, let’s go over a few prerequisites.

How To Build a Weather App with React

*You can*[*view the complete repository on Github*](https://github.com/jdretz/rapid-api-react-weather-app-tutorial)*for guidance if you run into trouble and for updates as of 2021.*

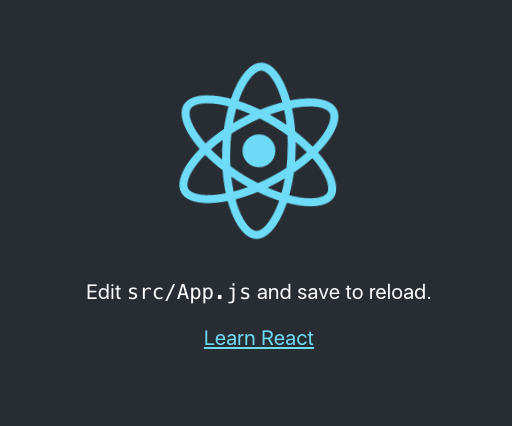
**Prerequisites**

* [You’ll need to have Node >= 8.10 and npm >= 5.6 on your machine.](https://reactjs.org/docs/create-a-new-react-app.html)
* You’ll need to have some familiarity with how to open and use a command-line tool.
* Internet connection.
* You’ll need a code editor (I recommend VS Code).
* A browser other than IE. It does not support the way we are going to call the API (fetch).
* I will be using the UNIX bash command-line tool on macOS for installation. However, using a Mac is not required, because this type of command line is not Mac-specific
* [RapidAPI account](https://rapidapi.com/signup)

**Getting Started**

1. Open a new terminal window and navigate to a comfortable directory.
2. Run npx create-react-app weather\_app in the new terminal. The download process will take a few minutes.
   1. npx is a program that we are giving a command to
   2. create-react-app is the command that makes a new React project
   3. weather\_app is an argument we are passing to name the project root directory
3. Once the download process is done, jump into the project by executing cd weather\_app. There are a lot of files but don’t worry, I will keep things simple for this tutorial and will focus on the necessary items.
4. Open your text editor. I am using Visual Studio Code, so I can use the command-line-interface provided by the editor. In the root project directory, I can enter code.
5. Back in the terminal at the root of the project run npm start. This starts the development server on our local machine so the project will compile every time we make a change and save. When the project tries to compile in development mode it informs us of any mistakes that we have made in editing. This is very helpful when starting out.

After a few moments, a new browser tab will open to the URL http://localhost:3000 and you should see the project.



We now have a React application running on our machine. Before we choose a weather API we are going to clean up a few items.

1. Open /public/index.html. This is the page that will be given to browsers when our domain is requested. Notice that there is only one HTML file in the whole project. This is why many ReactJS applications are called single-page-apps, or SPAs.
2. We will only make one change in this file: locate the title tag and change the text inside to Weather App, or anything you would like the title of the page to be. You should see the change reflected in the tab of your browser after you save.
3. Navigate to /src/App.js. You will notice that the content in App.js is being rendered in your browser. This file will become a layout container. To start, remove everything inside the header tag and replace it with:

**...**

**<h1>React Weather App</h1>**

**...**

1. Add two new layout elements (main and footer) underneath the closing header tag with your name in place of “yournamehere”.

**...**

**<main>**

**{/\* add weather fetching component \*/}**

**</main>**

**<footer>**

**Page created by yournamehere**

**</footer>**

**...**

1. In App.css, inside the App-header class, change the min-height property to 300px and replace the styling inside the App class with the below CSS.

**...**

**display:flex;**

**flex-direction: column;**

**min-height: 100vh;**

**...**

Unfortunately, the various CSS properties and values that I am using are beyond the scope of this article.

1. Add the below code to the end of App.css to provide more structure to the HTML.

**...**

**main {**

**flex-grow: 1;**

**text-align: center;**

**padding: 25px;**

**}**

**footer {**

**padding: 2px;**

**min-height: 100px;**

**display: flex;**

**flex-direction: column;**

**justify-content: center;**

**background: #EEEEEE;**

**font-size: small;**

**text-align: center;**

**}**

**header {**

**padding: 25px;**

**}**

**...**

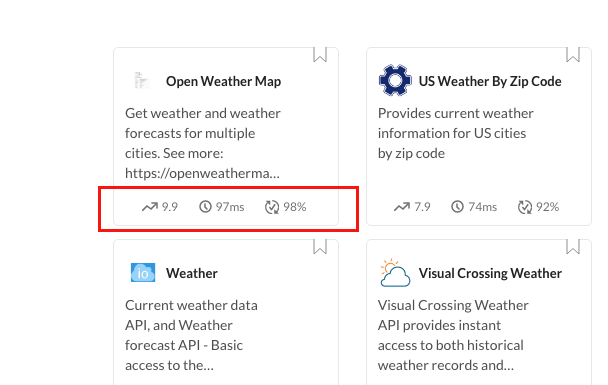
You should start to see the application layout. The page should look similar to mine.

[**Connect to the OpenWeatherMap API**](https://rapidapi.com/community/api/open-weather-map)

Choosing a Weather API

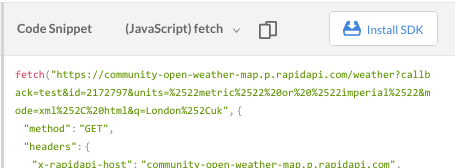
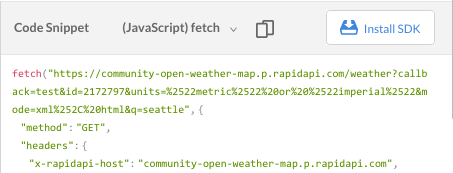
Our React weather app needs weather data, so we are going to take a look at [RapidAPI’s options](https://rapidapi.com/collection/top-weather-apis):

1. Open up the link above.  
   Each weather API option gives us four important things:
   1. Description of the API
   2. Popularity on RapidAPI
   3. Latency
   4. Reliability

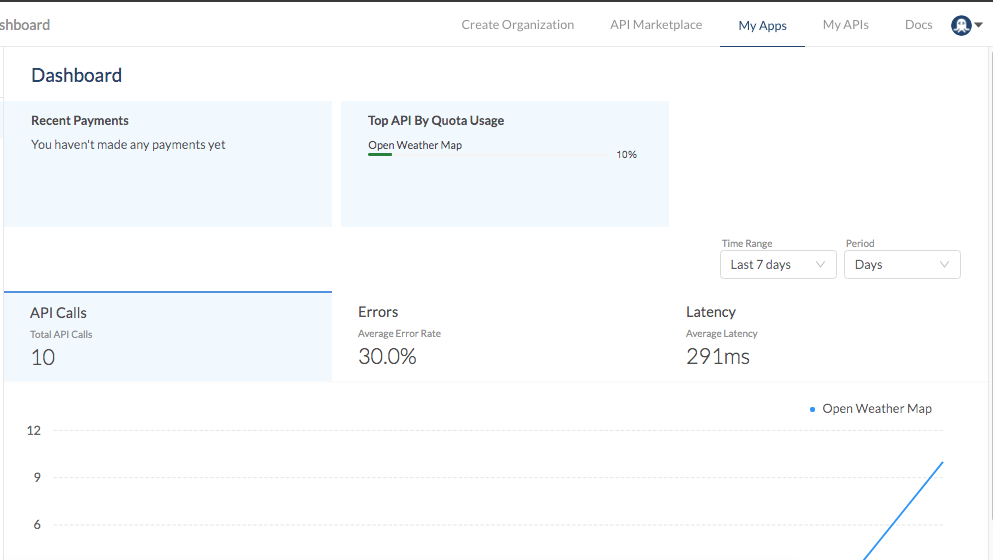
  
In the image above, I put a red box around the popularity, latency, and reliability indicators. We are going to use **Open Weather Map**:

1. Click it. The [Open Weather Map API information page](https://rapidapi.com/community/api/open-weather-map) (the page you should already be on) gives us more detailed information. On the far left side, we have a list of API endpoints, each giving us different data, and displaying what kind of HTTP request it uses. The middle section gives us more detailed information describing the selected endpoint and the parameters. Parameters are predefined labels for data that we are going to give to the API (i.e., city=Seattle).

On the right, my personal favorite of the three sections is an interactive help center that shows us code snippets from a variety of different languages, libraries, and sample responses from the endpoints. This will save us a lot of time. Notice the responses we want are in JSON, JavaScript Object Notation, which is a logical way to display information using objects and key-value pairs. TIP: the section on the right will respond to changes we make in the middle section. If we change parameters in the middle, the URL on the right should change.

1. In the code dropdown at the top of the right section select **JavaScript (fetch)**.  
   
2. Notice at the end of the URL string we see “q=London%252Cuk”. That is a URI encoded string for the ‘q’ parameter. Essentially, URI encoded strings are the URL compatible versions of the same string, but they account for special characters (i.e spaces and commas). Remember that for later.
3. Select **Current Weather Data** in the left section. In the middle section, underneath where it says **GET Current Weather Data**, in the **Required Parameters** section, change the ‘q’ parameter from ‘London,uk’ to ‘Seattle’.  
   Now, the end of the URL simply has ‘seattle’, and it looks the same as how we typed it in the parameter input because we did not have any special characters.

This is the code that we are going to copy & paste directly into our app. However, in order for us to test and use this code, we will need to have an account on RapidAPI and a subscription to Open Weather Map. You can quickly set up an account and [subscribe to the basic plan](https://docs.rapidapi.com/docs/basics-creating-a-project#subscribing-to-an-api), which gives us 100 calls per day (not exceeding 10 calls per minute) for free using the Open Weather Map API. **It’s important to track our API calls**.

1. In the top toolbar on RapidAPI select **My Apps**. We can see our total API calls and total calls per API in the graph at the bottom.

If, at any point, you feel like you are approaching your quota throughout this tutorial (should not be an issue) double-check your limit on the RapidAPI dashboard and do not continue if you are approaching your quota limit.  
We are almost ready to add the code snippet to our app but we need to make a file for it first.

1. In the src/ directory, make a folder named components/. Inside that folder make another folder named Forecast/, and finally, inside that folder create a file Forecast.js.This may seem redundant but it is a common practice in React applications to build folder structure in this way. It makes components, and their corresponding CSS and test code, modular. It won’t be the last time we do it.
2. Next, add the below JavaScript code to our new Forecast.js file.

**import React from 'react';**

**const Forecast = () => {**

**function getForecast() {**

**// weather data fetch function will go here**

**}**

**return (**

**// JSX code will go here**

**)**

**}**

**export default Forecast;**

This file will use React Hooks, so it’s important that the name of the function is capitalized. We are now ready to add our code snippet and start programming.

[**Connect to the OpenWeatherMap API**](https://rapidapi.com/community/api/open-weather-map)

Fetching Data

1. Head back over to the Open Weather Map page, copy the fetch code snippet and paste it into the getForecast function. Remove all the extra query parameters so just the main city parameter remains. The URL should be https://community-open-weather-map.p.rapidapi.com/weather?q=seattleNow, if we had a way to run this function, we would get the real response (not the example response you saw on RapidAPI’s site). However, we need a way to display the response on the webpage. Currently, this component is not a part of our application and it does not have any JSX code.
2. Remove the comment // **JSX code will go here** and replace it with:

**...**

**<div>**

**<h2>Find Current Weather Conditions</h2>**

**<div>**

**{JSON.stringify(responseObj)}**

**</div>**

**<button onClick={getForecast}>Get Forecast</button>**

**</div>**

**...**

Our component now has a title, a place to display data, and a button that calls the getForecast function when clicked, but what are the curly brackets?

Using curly brackets in JSX allows us to mix JavaScript right in with our HTML-like code (JSX). We are stringifying our response from the Open Weather Map API using built in JavaScript code. This has to be done because the data is returned as JSON and React does not render objects.

We don’t have the responseObj variable declared, so we will set up the useState functionality that is unique to React Hooks.

1. Change the first line at the top of the page to import useState:

**import React, { useState } from 'react';**

**...**

1. Above the getForecast function, but still inside the Forecast function add:

**...**

**let [responseObj, setResponseObj] = useState({});**

**...**

We are creating the responseObj variable and the function to change the responseObj variable by calling the useState function.

Wrapping the responseObj and setResponseObj in an array is called *destructuring* and is commonly used to easily access values inside arrays or objects. We can destructure this function call because we know the useState function returns an array with a variable as the first element and a function as the second.

We pass the useState function the starting value of the variable. We expect the future value to be a JSON object, so we are setting the starting value to an empty object.

1. Modify our getForecast function to convert the response into a JSON object and then assign the response value to the responseObj variable in our state. The inner workings of asynchronous programming are beyond the scope of this article, but the then function is what happens after we get our data from the API.

**...**

**.then(response => response.json())**

**.then(response => {**

**setResponseObj(response)**

**})**

**...**

1. Head back to our App.js file to import our new component and add the component inside our main element. The App.js file should look like this:

**import React from 'react';**

**import './App.css';**

**import Forecast from "./components/Forecast/Forecast";**

**function App() {**

**return (**

**<div className="App">**

**<header className="App-header">**

**<h1>React Weather App</h1>**

**</header>**

**<main>**

**<Forecast />**

**</main>**

**<footer>**

**Page created by yournamehere**

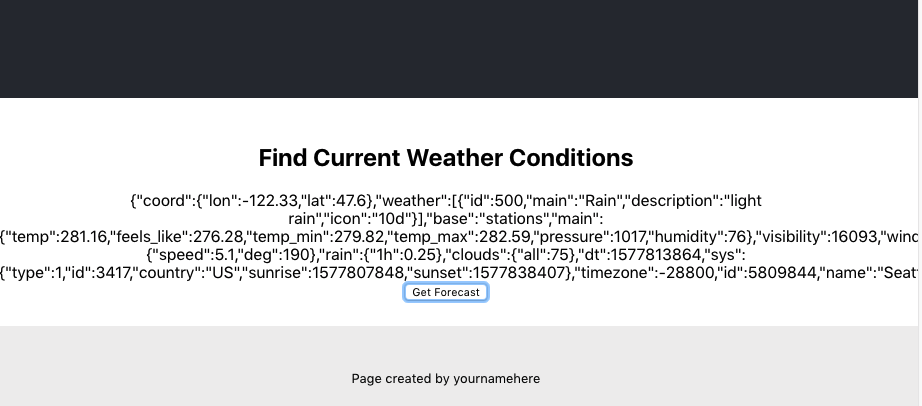
**</footer>**

**</div>**

**);**

**}**

**export default App;**

1. Click the **Get Forecast** button, you should see our response object. That response is real data that we can now use.  
   

[**Connect to the OpenWeatherMap API**](https://rapidapi.com/community/api/open-weather-map)

Displaying Data

We need to pull the information that we want from this object and display it in a meaningful way. To do that we are going to make a functional component named Conditions that will receive the responseObj from the Forecast component via props.

1. Create a new folder in the components/ folder. Inside of that folder create a directory named Conditions/ and inside Conditions/ create the Conditions.js file.The file will look similar to the last, but our function will take a single argument (props). You may have noticed earlier that an empty object was displayed even though there was not any data. We will add a ternary JavaScript operator to display the data only when the HTTP response code is 200.
2. Inspect the responseObj we will see that there is a “cod” key that has the value 200. If we did not enter a valid city in the query parameter the code would be 404. If we did not enter any city it would be a 400. Being able to view the example response—or test response—on RapidAPI helps us see how we can access data in the JSON response object without having to continuously call the API.

I have already inspected the response and decided that I want the:

* + current temperature
  + name of the city
  + description of conditions.

With that information, I can create a sentence that is easy for a user of the site to read.

1. The conditions component will have the following code:

**import React from 'react';**

**const Conditions = (props) => {**

**return (**

**<div>**

**{props.responseObj.cod === 200 ?**

**<div>**

**<p><strong>{props.responseObj.name}</strong></p>**

**<p>It is currently {Math.round(props.responseObj.main.temp)} degrees out with {props.responseObj.weather[0].description}.</p>**

**</div>**

**: null**

**}**

**</div>**

**)**

**}**

**export default Conditions;**

We are accessing data from the responseObj that is inside the props object. This is data flowing top-down. However, we have not set up the conditions component to receive that information yet.

Back in the Forecast component, let’s import the conditions component and pass it the responseObj.

1. At the top add:

**...**

**import Conditions from '../Conditions/Conditions';**

**...**

1. Remove the div element with our JSON.stringify call and add our conditions component underneath the button element.The JSX code, in Forecast.js, should look like that code below:

**...**

**<div>**

**<h2>Find Current Weather Conditions</h2>**

**<button onClick={getForecast}>Get Forecast</button>**

**<Conditions**

**responseObj={responseObj}**

**/>**

**</div>**

**...**

Now when we click the Get Forecast button we see clean data, but you will notice some things are missing:

* + The temperature is in Kelvin, with no option to choose between different units.
  + We are only getting the weather in Seattle, with no option to enter our own city.

We need an input form.

[**Connect to the OpenWeatherMap API**](https://rapidapi.com/community/api/open-weather-map)

Add User Input

The form is going to give the user the option to enter a city name and select between Celcius (metric) or Fahrenheit (imperial). The default will be Fahrenheit. The form could be built in a different component but for our weather example, we are going to build it into our Forecast component.

We are going to be adding dynamic values to our URL query string so we will need to hold those values in state so we can add those variables to the query string on submit.

1. Add the two new values, using useState above our current responseObj declaration.

**// Forecast.js**

**...**

**let [city, setCity] = useState('');**

**let [unit, setUnit] = useState('imperial');**

**...**

Notice we are setting the initial unit value to ‘imperial’. This will act as a default.

The city-input will need to be URI encoded before we put it in our URL string. JavaScript has a built-in function that takes care of this.

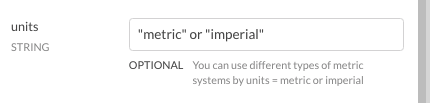
1. Below the useState declarations, create a new variable that converts the input string. We will use const because we do not want this new value to change.

**...**

**const uriEncodedCity = encodeURIComponent(city);**

**...**

If we use a template literal (template string) we can dynamically add variables to our URL string.

1. To do this, change the quotes to single backticks ( ` ) and place our variables inside brackets with a leading ( $ ).Our new URL string (inside of the fetch call) will look like: `https://community-open-weather-map.p.rapidapi.com/weather?units=${unit}&q=${uriEncodedCity}`  
   Back in the middle section of the Open Weather API dashboard, you can read that the parameter “units” take arguments for how the temperature data should be converted. Depending on what the user selects, we will either pass “imperial” or “metric” to the URL string.
2. Now, there is a dynamic string that is formed to fully access the global weather data. Let’s add the form and inputs to the JSX code, and convert the current button to call the form’s submit function when clicked.
3. In Forecast.js replace the button element with the form below.

**...**

**<form onSubmit={getForecast}>**

**<input**

**type="text"**

**placeholder="Enter City"**

**maxLength="50"**

**value={city}**

**onChange={(e) => setCity(e.target.value)}**

**/>**

**<label>**

**<input**

**type="radio"**

**name="units"**

**checked={unit === "imperial"}**

**value="imperial"**

**onChange={(e) => setUnit(e.target.value)}**

**/>**

**Fahrenheit**

**</label>**

**<label>**

**<input**

**type="radio"**

**name="units"**

**checked={unit === "metric"}**

**value="metric"**

**onChange={(e) => setUnit(e.target.value)}**

**/>**

**Celcius**

**</label>**

**<button type="submit">Get Forecast</button>**

**</form>**

**...**

We now have an unstyled form that takes user input—but don’t try to use it. If you clicked the **Get Forecast** button the page would refresh and we would lose our state and information. In order to make sure this doesn’t happen we pass the event object to our getForecast function and call the preventDefault method on it.

1. Our get forecast needs the below addition (instead of ‘event’ we will just use ‘e’):

**...**

**function getForecast(e) {**

**e.preventDefault();**

**...**

1. If you click the button now, the getForecast function gets called from the onSubmit listener in the form element and users can search for cities.  
   Notice we are assigning each input a corresponding value from our state and updating that value with the corresponding function using the onChange event listener. Each change is stored and passed with the ‘e’ (event) argument. We can now get current weather data, in Fahrenheit or Celcius, and display it right on our webpage.
2. We’re getting to the end of our build, but there are still a few items left. Let’s add some modular CSS code to improve the look of our app.

[**Connect to the OpenWeatherMap API**](https://rapidapi.com/community/api/open-weather-map)

CSS

React offers an easy way to create modular CSS files. This helps keep our style sheets clean and up-to-date. In the Forecast/ folder, create the file Forecast.module.css. In that file, we are going to add styles that can then be imported into Forecast.js.

1. Add the below CSS to the file:

**.TextInput {**

**font-size: large;**

**border: 1px solid #EEEEEE;**

**padding: 5px;**

**border-radius: 3px;**

**display: block;**

**margin: 5px auto;**

**}**

**.TextInput:focus {**

**outline: none;**

**border-color: #719ECE;**

**box-shadow: 0 0 10px rgb(50, 100, 153);**

**}**

**.Radio {**

**font-size: small;**

**}**

**.Button {**

**display: block;**

**margin: 10px auto;**

**padding: .5rem;**

**font-size: 1em;**

**background: darkcyan;**

**border-radius: 5px;**

**color: white;**

**cursor: pointer;**

**margin: 5px auto;**

**}**

1. In the Forecast.js file, import these classes at the top of the file with this line:

**...**

**import classes from './Forecast.module.css';**

**...**

The selectors that we put in the CSS file can be accessed on the classes object.

1. Add the CSS classes (like we would add any other JavaScript) to the JSX code.
   * The text input gets the TextInput class
   * The button gets the Button class
   * The two label elements get the Radio class

**...**

**<input**

**type="text"**

**placeholder="Enter City"**

**maxLength="50"**

**className={classes.textInput}**

**...**

**...**

**<label className={classes.Radio}>**

**...**

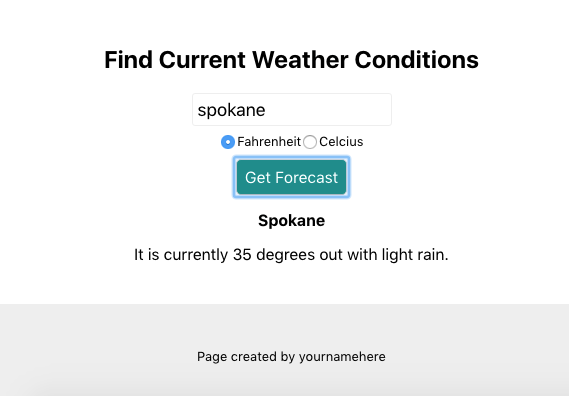
**<label className={classes.Radio}>**

**...**

**...**

**<button className={classes.Button} type="submit">Get Forecast</button>**

**...**

The form should now be stacked, and when we search for a city a nice vertical structure should be present.  


When the forecast displays it tends to push the page around. This could be a little irritating to a user and it’s an easy fix for us because we know the forecast is always going to be relatively small.

Let’s add a CSS file, the same way we did with the Forecast component, alongside the Conditions component to make the visuals a little nicer. We will use this file later for a loading spinner and some error styling, too.

1. Create Condtions.module.css file in the Conditions/ directory. Add the CSS class to it:

**.Wrapper {**

**min-height: 100px;**

**display: flex;**

**flex-direction: column;**

**}**

1. Import the file and add the class to the outer div element.

**...**

**import classes from './Conditions.module.css'**

**...**

**...**

**<div className={classes.Wrapper}>**

**...**

There should be some empty space below the **Get Forecast** button that we will fill with an error or a loading spinner depending on what’s going on with our function call. Let’s move on to error handling and loading.

[**Connect to the OpenWeatherMap API**](https://rapidapi.com/community/api/open-weather-map)

Error Handling and Loading

It could be possible that the weather API does not return a status code of 200. The user may type in the city wrong, they may not type a city at all, or the user may want to know if there request was even sent.

1. In order to cover these scenarios let’s add two more variables to our state.

**...**

**let [error, setError] = useState(false);**

**let [loading, setLoading] = useState(false);**

**...**

If the user doesn’t enter any characters into the input we do not want to fire off the request.

1. Add a check just below our e.preventDefault() call for this.

**...**

**if (city.length === 0) {**

**return setError(true);**

**}**

**...**

This ends the function and sets the error to true which will trigger a message in the Conditions component (we will add soon) and, as we planned, not fire off an empty request.

Although the JavaScript built-in fetch function has a **catch** block, it will still run the code in the **then** block if the API request returns with a status of 400 or 404.

We don’t want that. If there is an error, we want to display an error message to the user and not set the response object.

1. Add a similar block of code to the **then** block to check for an error and throw an error if the code on the response is not 200. Inside the second **then** block of our fetch function add:

**...**

**if (response.cod !== 200) {**

**throw new Error()**

**}**

**...**

This will run the code in the **catch** block, so let’s set the error value to true in the **catch** block with setError(true).

We are almost done with the function but we need to think about how loading will work;

* + Loading should be true at the start of the function
  + Loading should be false when the function is successful
  + Loading should be false if the function fails.

It’s important to give the user feedback when making API calls or any HTTP requests. Some connections are slower than others and servers can get busy.

1. Add the above loading logic to our function with a few more clean up items. The finished getForecast function will look as follows:

**function getForecast(e) {**

**e.preventDefault();**

**if (city.length === 0) {**

**return setError(true);**

**}**

**// Clear state in preparation for new data**

**setError(false);**

**setResponseObj({});**

**setLoading(true);**

**let uriEncodedCity = encodeURIComponent(city);**

**fetch(`https://community-open-weather-map.p.rapidapi.com/weather?units=${unit}&q=${uriEncodedCity}`, {**

**"method": "GET",**

**"headers": {**

**"x-rapidapi-host": "community-open-weather-map.p.rapidapi.com",**

**"x-rapidapi-key": "apikey"**

**}**

**})**

**.then(response => response.json())**

**.then(response => {**

**if (response.cod !== 200) {**

**throw new Error()**

**}**

**setResponseObj(response);**

**setLoading(false);**

**})**

**.catch(err => {**

**setError(true);**

**setLoading(false);**

**console.log(err.message);**

**});**

**}**

It won’t matter if our logic is correct if there is nothing to display that represents the logic. We are going to use the Conditions component for error and loading feedback.

We’ll need two ternary expressions in the JSX code of the Conditions components that will turn on and off depending on the props that are passed.

1. Add these two props to the Conditions component in Forecast.js so it looks like this.

**...**

**<Conditions**

**responseObj={responseObj}**

**error={error} //new**

**loading={loading} //new**

**/>**

**...**

1. Head over to Conditions.js to add the logic.  
   This time we will use different logical shortcut syntax that looks a little cleaner. We will be directly looking at the error and loading booleans and the component will decide what elements to display.
2. Add this code at the top of our main div.

**...**

**{props.error && <small>Please enter a valid city.</small>}**

**{props.loading && <div>Loading...</div>}**

**...**

If we try to look up a city without entering a name the error message will appear, and while we wait for the response from the API we will see an element informing us that data is loading.

1. Let’s make it a little better by adding some styling to Conditions.module.css.

**...**

**.Small {**

**color: red;**

**}**

**/\* this code was pulled from**

**https://www.w3schools.com/howto/howto\_css\_loader.asp \*/**

**.Loader {**

**border: 16px solid #f3f3f3;**

**border-top: 16px solid rgb(90, 89, 89);**

**border-radius: 50%;**

**width: 50px;**

**height: 50px;**

**animation: spin 2s linear infinite;**

**margin: 10px auto;**

**}**

**@keyframes spin {**

**0% { transform: rotate(0deg); }**

**100% { transform: rotate(360deg); }**

**}**

After importing the class object and adding the corresponding classes to the proper elements the final Conditions.js page should look like this:

**import React from 'react';**

**import classes from './Conditions.module.css'**

**const conditions = (props) => {**

**return (**

**<div className={classes.Wrapper}>**

**{props.error && <small className={classes.Small}>Please enter a valid city.</small>}**

**{props.loading && <div className={classes.Loader} />}**

**{props.responseObj.cod === 200 ?**

**<div>**

**<p><strong>{props.responseObj.name}</strong></p>**

**<p>It is currently {Math.round(props.responseObj.main.temp)} degrees out with {props.responseObj.weather[0].description}.</p>**

**</div>**

**: null**

**}**

**</div>**

**)**

**}**

**export default conditions;**

Now, we have error handling and a loader for user feedback. Only two things left;

* Security
* Logo

[**Connect to the OpenWeatherMap API**](https://rapidapi.com/community/api/open-weather-map)

Security

If you didn’t already notice, I removed my API-key when I pasted my code in for the getForecast function. React is a front-end framework, so all of our code gets sent to the client. That would mean our API-key could be viewed by anyone that was able to see our source code if they opened up the developer tools.

If we wanted to create an optimized build of our application we could run npm run build and all of our code would be made small and bundled for performance. Even if we put the API-key in a .env file and imported it with process.env.REACT\_AP\_API\_KEY (this is built-in support for non-sensitive environment variables) our key could still be found.

What that means is this code should not be put on a server, or CDN, to maintain the integrity of our API-key. I can only recommend that you set up a [backend Node.js server for our weather app](https://rapidapi.com/blog/weather-app-javascript-nodejs/).

What we can prevent is committing our API-key to Github when we share it with version control. This is where the /.env file is useful.

Notice that when the app was created for us a /.gitignore file was created automatically at the root of the project directory. Files and folders in this file will not be committed to remote repositories. Explaining version control in depth is outside the scope of this article, so please bear with me.

Inside this file there are a few .env files.

* .env.local
* .env.development.local
* .env.test.local
* .env.production.local

We are going to create a permanent .env file named .env because our API-key is not a test or development key.

1. Add .env to our .gitignore file.
2. Create the .env file at the root of our directory. Inside the file, add the following line:

**REACT\_APP\_API\_KEY=yourapikey**

It’s important that the variable starts with ‘REACT\_APP\_’. If it doesn’t it will not be recognized.

1. In Forecast.js, replace your current API-key with process.env.REACT\_APP\_API\_KEY. The line should like the line below:

**...**

**"x-rapidapi-key": process.env.REACT\_APP\_API\_KEY**

**...**

1. Restart the development server (Ctrl+C in bash and then npm start to restart it).
2. Test app by entering a city name.

The application should still be able to make calls to OpenWeatherMap.

[**Connect to the OpenWeatherMap API**](https://rapidapi.com/community/api/open-weather-map)

Logo

This is just a finishing touch. I will only explain where to add the code.

1. Create a folder in the components/ directory titled Logo, and inside of that make two files: Logo.js and Logo.module.css.
2. Paste the below code into the corresponding files.

**// Logo.js**

**import React from 'react';**

**import classes from "./Logo.module.css"**

**const Logo = () => (**

**<div className={classes.sky}>**

**<div className={classes.circle}/>**

**<div className={classes.cloud1} />**

**<div className={classes.cloud2} />**

**<div className={classes.cloud3} />**

**</div>**

**)**

**export default Logo;**

**/\* Logo.module.css \*/**

**.circle {**

**width: 60%;**

**height: 60%;**

**background: yellow;**

**border-radius: 50%;**

**margin: auto;**

**}**

**.sky {**

**display: flex;**

**width: 200px;**

**height: 200px;**

**background: skyblue;**

**border-radius: 5px;**

**}**

**.cloud1 {**

**position: absolute;**

**margin-top: 120px;**

**margin-left: 50px;**

**width: 100px;**

**height: 50px;**

**background: rgb(226, 224, 224);**

**border-radius: 80px / 40px;**

**}**

**.cloud2 {**

**position: absolute;**

**margin-top: 100px;**

**margin-left: 5px;**

**width: 100px;**

**height: 50px;**

**background: rgb(226, 224, 224);**

**border-radius: 100px / 50px;**

**}**

**.cloud3 {**

**position: absolute;**

**margin-top: 90px;**

**margin-left: 50px;**

**width: 60px;**

**height: 60px;**

**background: rgb(226, 224, 224);**

**border-radius: 60px;**

**}**

1. Import the component into App.js:

**....**

**import Logo from './components/Logo/Logo';**

**...**

1. Add the Logo component inside the header element but above the h1 element.

**...**

**<header className="App-header">**

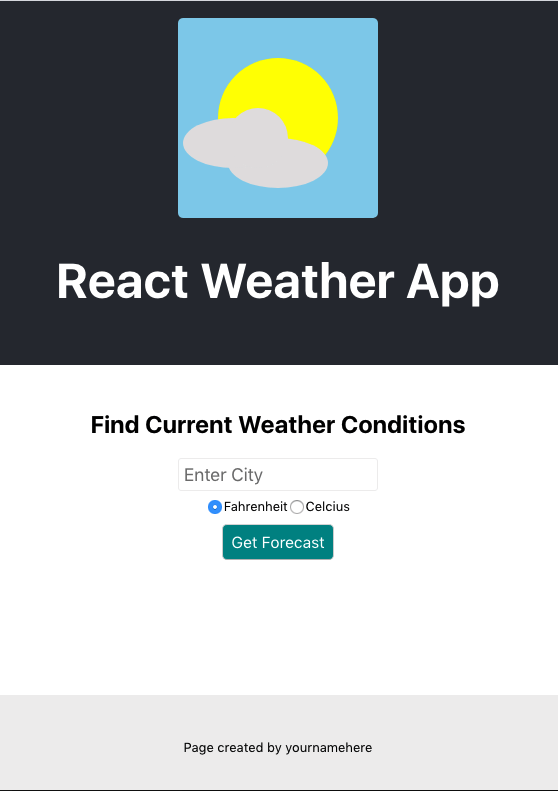
**<Logo />**

**<h1>React Weather App</h1>**

**</header>**

**...**

1. Our app is now is a little easier to look at!



Thank you for following along. I hope you learned something and feel empowered to start building APIs with React!

[**Connect to the OpenWeatherMap API**](https://rapidapi.com/community/api/open-weather-map)

What’s Next?

This app has a lot of room to improve. I have a few suggestions below for improving or extending the application.

* As previously mentioned, try to extend or change [our weather app example to use backend Node.js code](https://rapidapi.com/blog/weather-app-javascript-nodejs/)
* Move our weather form into a different component folder and import it into our Forecast component.
* Add a second API endpoint from Open Weather Map, or refactor our current endpoint to display search options.
* Integrate with [Google Maps](https://rapidapi.com/blog/google-maps-api-react/)