

# Build a Weather App from Scratch with Next.js



# Build a Weather App from Scratch with Next.js

Copyright © 2021 SitePoint Pty. Ltd.

- **Author:** Madars Biss
- **Cover Design:** Alex Walker

## Notice of Rights

All rights reserved. No part of this book may be reproduced, stored in a retrieval system or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical articles or reviews.

## Notice of Liability

The author and publisher have made every effort to ensure the accuracy of the information herein. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors and SitePoint Pty. Ltd., nor its dealers or distributors will be held liable for any damages to be caused either directly or indirectly by the instructions contained in this book, or by the software or hardware products described herein.

## Trademark Notice

Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.



Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood

VIC Australia 3066

Web: [www.sitepoint.com](http://www.sitepoint.com)

Email: [books@sitepoint.com](mailto:books@sitepoint.com)

## About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit [sitepoint.com](http://sitepoint.com) to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, Ruby, mobile development, design, and more.

# Build a Weather App from Scratch with Next.js

In this tutorial, we'll be building a weather app. Users will be able to check the weather across the world, and the data will be presented in a user-friendly UI.

You'll learn how to create a wireframe, design an app, create components, work with states, use service functions, add responsiveness and deploy the app.

We'll use [NextJS](#) and [OpenWeatherMap API](#) as our stack. NextJS is a [React](#) framework created by [Vercel](#) to build scalable static and dynamic websites and web applications. OpenWeatherMap is an online data platform that provides historical, current and forecasted weather data via efficient APIs.

Here's the [source code](#) of the final project and the deployed [demo](#). Make sure to use both of these as helpful references while building the app.

## Planning the Features

The most basic weather apps usually provide just the general information like location and temperature, while others are more complex—including weather forecasts, detailed data graphs, and options to filter the information.

We'll make a blend between the two scopes by providing the current weather conditions for any city in the world and also allowing user interaction to change the location and display the data in their preferred unit system.

The full list of features we'll implement include:

- the ability to search cities
- current local time and date
- temperatures in both metric and imperial units
- humidity, wind speed and direction
- sunrise and sunset times
- error handling and loading info

# Wireframing the App

First, let's create a wireframe, to help us plan out our user interface so that we can accommodate our desired features.

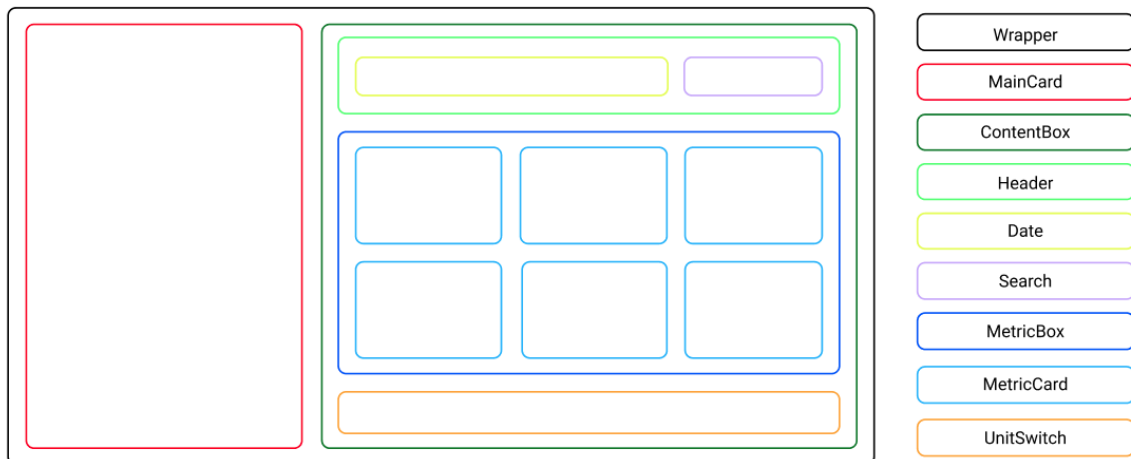
On the top level, we'll use a two-column layout.

Since the main weather conditions need to be highlighted, we'll create a separate `MainCard` component and dedicate a whole left column to it. It will hold information about the location, weather status, description, and temperatures.

The right column will be the `ContentBox` component, which will include three direct children components— `Header` , `MetricsBox` and `UnitSwitch` .

The `Header` component will further hold two children components— `DateAndTime` and `Search` —while `MetricsBox` will hold all the `MetricsCard` components for measurements like humidity, wind, visibility, as well as sunrise and sunset times.

If we put everything in the wireframe, we'll see that the main app will consist of eight components and the layout will be structured as shown below.



We'll also use a loading screen, which will be displayed while the data is being loaded and an error screen, which will be shown if the search query returns no results.

Both will be wrapper components with the error screen including the `Search` component below the error message. Based on their simplicity, we'll not design separate wireframes for them.

## Designing the App

The next step is for us to define a color scheme for our app.

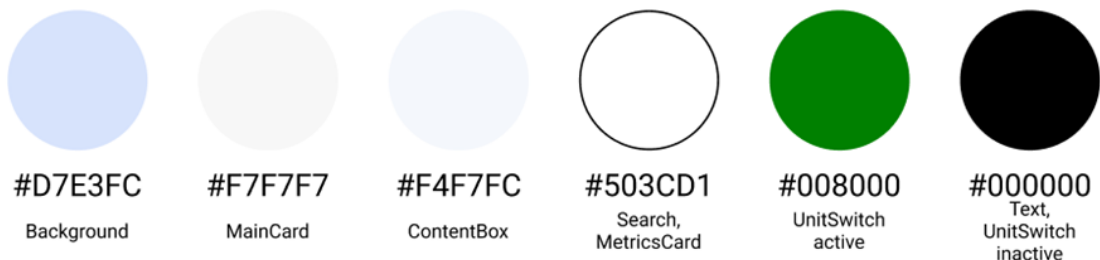
We'll use neutral tones for the backgrounds of the panels. For the body background, we'll pick a slightly darker tone just to highlight the app itself.

Inside the app, we need to distinguish the `MainCard` section from the `ContentBox`, so we'll use different shades for both of those. We'll set a light grey background and a light blue background for both elements, respectively.

A couple of other things we need to highlight would be the `Search` and the `MetricsCard` components. Since both of those will be on the light blue background inside `ContentBox`, we'll set a white background for both of them.

Finally, we need to style the `UnitSwitch` component. It will include a switch mechanism between metric and imperial units, so we need to set the colors for active and inactive options. We'll use a green shade for active, and black for inactive.

If we assign the color tones to their respective elements, we come up with a color palette that looks like the one shown below.



Also, I purposely chose a black tone for the text to give maximum contrast to the background. This way, the app will be easy for the eyes and the information will be easy to read, which is one of the cornerstones for a great UI/UX.

## Setting Up the Project

To get started with a boilerplate, we'll use [Create Next App](#), which is an officially supported CLI tool that lets you create a new NextJS project within a minute or less.

Open your terminal and run the following command:

```
npx create-next-app weather-app
```

Next, switch to the newly created `weather-app` folder with `cd weather-app`.

Now check if you have [Node](#) and [npm](#) installed. Run `node -v` to see the version of Node and `npm -v` to see the version of npm. Both are available for download [here](#).

Finally, to start the development server, run `npm run dev`.

This will start a development server on the port `3000`. After the dev server has successfully started, the terminal will display the message “Started server on <http://localhost:3000>”.

Hold the `Ctrl` key and click on the link to open it right from the terminal, or open your web browser and enter the URL `http://localhost:3000` manually.

You should be presented with NextJS default placeholder.

# Welcome to Next.js!

Get started by editing `pages/index.js`

## Documentation →

Find in-depth information about Next.js features and API.

## Learn →

Learn about Next.js in an interactive course with quizzes!

## Examples →

Discover and deploy boilerplate example Next.js projects.

## Deploy →

Instantly deploy your Next.js site to a public URL with Vercel.

Switch back to the project and see the files folder tree. Navigate to the `pages` directory and find the `index.js` file inside it. Remove all of its content for now.

Do the same for the `globals.css` and `home.module.css` files. Both you will find in the `styles`

folder in the project's root.

## Configuring OpenWeatherMap API

The core of our app will depend on the data. We'll use the OpenWeatherMap service to fetch the data from their database via their API.

To set it up, visit [OpenWeatherMap.org](https://openweathermap.org).

Create a new account (if you don't already have one) and sign in.

Navigate to **My API keys**. The default API key should be already generated for you. Copy the `key` value, as you'll need it later.

Then switch back to the NextJS app we're building.

Create a new `.Local.env` file in the project's root and set the copied value to the `OPENWEATHER_API_KEY` key, as shown below:

```
OPENWEATHER_API_KEY=yourapikey
```

Now you can access the value across the app using `process.env.OPENWEATHER_API_KEY`.

## Downloading Icons

We'll use graphics to display the weather conditions.

I've compiled a pack of all the necessary icons. You can download them all at once from [here](#) (direct link via DownGit).

Once the download is finished, unzip the file and copy the `icons` folder from it. Then switch back to your `weather-app` project and paste it into the `public` folder.

You can later replace the items with your own or from sites like [Icon8](#) or [FlatIcon](#). Just make sure the icons you replace correspond with the weather conditions displayed in the initial ones.

## Creating the Base

To start building the actual app, we first need to create the base to build upon. We'll create it by adding some starter code to `globals.css`, `index.js`, and `Home.module.css`.



Let's start with the `globals.css` file, which defines the styling in the global scope and is accessible throughout the whole app. You'll find it in the `styles` folder. Open the file and include the following styles:

```
@import url("https://fonts.googleapis.com/css2?family=Varela+Round&display=swap");

* {
  padding: 0;
  margin: 0;
  box-sizing: border-box;
}

body {
  display: flex;
  justify-content: center;
  align-items: center;
  width: 100vw;
  min-height: 100vh;
  background-image: radial-gradient(
    circle 993px at 0.5% 50.5%,
    rgba(137, 171, 245, 0.37) 0%,
    rgba(245, 247, 252, 1) 100.2%
  );
  font-family: "Varela Round", sans-serif;
}
```

We've imported the Varela Round font and set it up in the `body`, meaning it will be used in all elements. Then we've created some reset rules for padding, margin and box-sizing, so we shouldn't have to deal with browser defaults. It's a good idea to do this for every app you make.

Then we've set up the flex layout and configured it to center the children element in the viewport, which will be the main wrapper for our app. We've also set a background gradient based on the tones we designed and set it to full width and height of the viewport, meaning it will always fill the entire screen.

Next, we'll create a base for `index.js`, which will contain the whole logic of the app.

Navigate back to the `pages` folder, open `index.js`, and include the following code:

```
import styles from "../styles/Home.module.css";

const App = () => {
  return (
    <div className={styles.wrapper}>
```

```
    <p>Weather App wrapper</p>
  </div>
);
};

export default App;
```

We've created a very simple function that returns the main wrapper of the whole app and some simple placeholder in it, which we'll remove later.

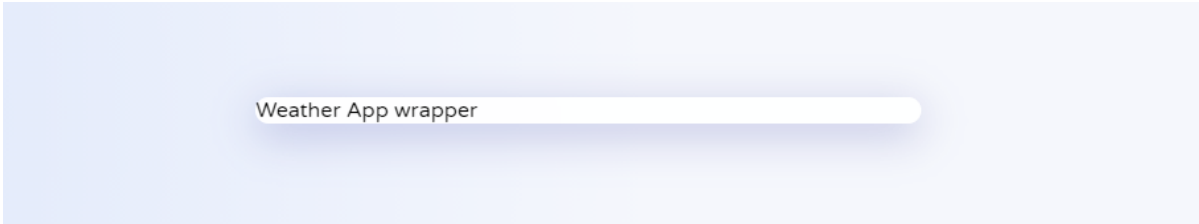
We've also imported `Home.module.css`, which we'll use to style `index.js`.

Let's add some styling rules for the imported file. Navigate back to the `styles` folder, open the `Home.module.css` file, and include the following styles:

```
.wrapper {
  display: grid;
  grid-template-columns: 1fr 2fr;
  max-width: 1200px;
  background: rgba(255, 255, 255, 0.95);
  box-shadow: 0 8px 32px 0 rgba(83, 89, 179, 0.37);
  backdrop-filter: blur(3px);
  -webkit-backdrop-filter: blur(3px);
  border-radius: 30px;
  overflow: hidden;
}
```

We've set the wrapper to use a grid layout with two columns with a 1:2 ratio. We've then defined the wrapper to not exceed specific width. Finally, we've set a background color, added box-shadow properties, rounded the corners, and made sure that the corners of the included children elements won't be displayed outside the parent.

Now check your terminal and see if your app is still running. If it's not, run `npm run dev`. Then open up your web browser and you should see the current render of the project.



Weather App wrapper

It doesn't seem like much at this point, but we've created solid foundations for the further development of the project.

## Identifying the States

Our app will receive data based on user interactions. That means the data will be updated dynamically. To render updates properly on the screen, they need to be stored in state variables.

We'll need to keep track of the input that users have entered, execute the fetch call when the user presses `Enter`, and store the selected unit system as well as the received data from `OpenWeatherMap`.

We'll use built-in React [useState hook](#) to handle the states, which is a standard way of doing this in any React ecosystem. You can learn more about Hooks [here](#).

Add the following code to `index.js` :

```
import { useState } from "react";
import styles from "../styles/Home.module.css";

const App = () => {
  const [cityInput, setCityInput] = useState("Riga");
  const [triggerFetch, setTriggerFetch] = useState(true);
  const [weatherData, setWeatherData] = useState();
  const [unitSystem, setUnitSystem] = useState("metric");

  return (
    //...
  )
}

export default App;
```

First, we've imported the React `useState` hook at the very top of the file. Then we've created four `useState` functions to hold the state variables for `cityInput`, `triggerFetch`, `weatherData`, and `unitSystem`.

Notice that we've also set the default values for `cityInput`, `triggerFetch`, and `unitSystem`. You can change the default `cityInput` value (`Riga` in this case) to any other city you want to be presented first each time on the initial launch of the app. The same goes for `unitSystem`: the default is `metric`, but you can change it to `imperial` if you prefer imperial units on the initial launch.

## Fetching the Data

Now for the exciting part. We'll create a fetch call for the data.

We'll use React `useEffect` hook to control when the request needs to be sent and the JS Fetch API to send the actual request for the data.

Add the following code to `index.js` :

```
import { useState, useEffect } from "react";
import styles from "../styles/Home.module.css";

const App = () => {

  //...

  useEffect(() => {
    const getData = async () => {
      const res = await fetch("api/data", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ cityInput }),
      });
      const data = await res.json();
      setWeatherData({ ...data });
      setCityInput("");
    };
    getData();
  }, [triggerFetch]);

  console.log(weatherData);

  return (
    //...
  )
}

export default App;
```

First, we've imported the React `useEffect` hook, which will let us control when we want to make a request for the data. Then we've set it up inside the `App` component.

We've also passed in the `triggerFetch` variable in the brackets, meaning the `useEffect` will run each time the `triggerFetch` state updates. We'll create a function that updates `triggerFetch` each time the `Enter` key is pressed once we import the components inside `index.js`.

Inside `useEffect` we've created a `fetch` call to the `api/data` route that will process our request when received. We've then set the `request method` as `post`, added a `content type` as `json`, and included our search query in the `body` —which, at this point, is the default value for

the `cityInput` state variable.

Next, we need to create an actual API endpoint ( `api/data` ) where the call from the front end goes to.

Navigate to the `api` folder (inside the `pages` directory) and create a new file called `data.js` . If there are other files created by Create Next App, feel free to remove them.

Include the following code in `api/data.js` :

```
export default async function handler(req, res) {
  const { cityInput } = req.body;
  const getWeatherData = await fetch(
    `https://api.openweathermap.org/data/2.5/weather?q=${cityInput}&units=metric&appid=${process.env
    ↪ .OPENWEATHER_API_KEY}`
  );

  const data = await getWeatherData.json();
  res.status(200).json(data);
}
```

We're receiving a request from `index.js` and then making a fetch call to OpenWeatherMap. Notice that we've included the received search query in the fetch URL and set the `OPENWEATHER_API_KEY` from `.env.Local` we configured earlier.

Once the data has been received, the response back to the front end is being made with included data in JSON format, which is then set in the previously created `weatherData` state variable.

If you check the previous code block, you'll also notice that I included a `console.Log(weatherData)` for testing purposes. Now let's test if the fetching of the data works as expected.

Switch back to the browser and open the developer console. You can do that by pressing `F12` on the keyboard or manually from the browser settings. Refresh the app by pressing `F5`.

If you've followed along and done everything correctly, you should receive the response in the developer console, as pictured below.

```

▼ {coord: {...}, weather: Array(1), base: 'stations', main: {...}, visibility: 10000, ...}
  base: "stations"
  ▶ clouds: {all: 0}
  cod: 200
  ▶ coord: {lon: 24.0833, lat: 57}
  dt: 1632599858
  id: 456173
  ▶ main: {temp: 11.04, feels_like: 10.47, temp_min: 9.95, temp_max: 11.07, pressure: 1011, ...}
  name: "Riga"
  ▶ sys: {type: 1, id: 1876, country: 'LV', sunrise: 1632543288, sunset: 1632586552}
  timezone: 10800
  visibility: 10000
  ▶ weather: [{...}]
  ▶ wind: {speed: 1.54, deg: 280}
  ▶ [[Prototype]]: Object

```

react\_devtools\_backend.js:4049

## Creating Data Converters

Since we'll be working with two different unit systems (metric and imperial), we'll need to create several converter functions before we can show the received data into our app.

Create a new folder in the project's root and name it `services`, then create a new file `converters.js` inside it and include the following code:

```

export const ctoF = (c) => (c * 9) / 5 + 32;

export const mpsToMph = (mps) => (mps * 2.236936).toFixed(2);

export const kmToMiles = (km) => (km / 1.609).toFixed(1);

export const timeTo12HourFormat = (time) => {
  let [hours, minutes] = time.split(":");
  return `${(hours % 12) ? hours : 12}:${minutes}`;
};

export const degToCompass = (num) => {
  var val = Math.round(num / 22.5);
  var arr = [
    "N",
    "NNE",
    "NE",
    "ENE",
    "E",
    "ESE",
    "SE",
    "SSE",
    "S",
    "SSW",
    "SW",
    "WSW",
    "W",
    "WNW",

```

```

    "NW",
    "NNW",
  ];
  return arr[val % 16];
};

export const unixToLocalTime = (unixSeconds, timezone) => {
  let time = new Date((unixSeconds + timezone) * 1000)
    .toISOString()
    .match(/(\d{2}:\d{2})/)[0];

  return time.startsWith("0") ? time.substring(1) : time;
};

```

`cToF` takes in temperature value in `Celsius` and returns the value in `Fahrenheit`.

`mpsToMph` takes in the wind speed value in `meters per second` and returns the value in `miles per hour`.

`kmToMiles` takes in distance value in `kilometers` and returns the value in `miles`.

`timeTo12HourFormat` takes in the time value in `24-hour format` and returns the value in `12-hour format` (for example: 22:32 to 10:32).

`degToCompass` takes in the angle in `degrees` and returns the corresponding `direction`. First, we divide the `angle` by 22.5, because there are 16 directions (360/16), then round the value to the nearest `integer`, which we then use to calculate `modulus` from and detect the position in the given `array` of direction names.

`unixToLocalTime` takes in the UNIX time in `seconds` (UTC) and the difference in the `seconds` for the local timezone. The `new Date` object is created, where both values are added and then multiplied by 1000, since `Date object` requires `milliseconds`. Then we use `regex` to get the first result that follows the `hh:mm` pattern. Finally, if the returned `string` starts with "0", we remove the first character (for example, 07:12 to 7:12).

## Adding Helper Functions

Next, we'll create some helper functions so that it's easier to work with data inside the components and we are allowed to reuse particular logic across the components.

While still in the `services` folder, create a new `helpers.js` file and include the following code:

```

import {
  unixToLocalTime,
  kmToMiles,
  mpsToMph,
  timeTo12HourFormat,
} from "./converters";

export const getWindSpeed = (unitSystem, windInMps) =>
  unitSystem == "metric" ? windInMps : mpsToMph(windInMps);

export const getVisibility = (unitSystem, visibilityInMeters) =>
  unitSystem == "metric"
    ? (visibilityInMeters / 1000).toFixed(1)
    : kmToMiles(visibilityInMeters / 1000);

export const getTime = (unitSystem, currentTime, timezone) =>
  unitSystem == "metric"
    ? unixToLocalTime(currentTime, timezone)
    : timeTo12HourFormat(unixToLocalTime(currentTime, timezone));

export const getAMPM = (unitSystem, currentTime, timezone) =>
  unitSystem === "imperial"
    ? unixToLocalTime(currentTime, timezone).split(":")[0] >= 12
      ? "PM"
      : "AM"
    : "";

export const getWeekDay = (weatherData) => {
  const weekday = [
    "Sunday",
    "Monday",
    "Tuesday",
    "Wednesday",
    "Thursday",
    "Friday",
    "Saturday",
  ];
  return weekday[
    new Date((weatherData.dt + weatherData.timezone) * 1000).getUTCDay()
  ];
};

```

`getWindSpeed` takes in the `unit system` and wind speed in `meters in seconds`. If the metric system is used, it returns `meters per second`, otherwise `miles per hour`.

`getVisibility` takes in the `unit system` and visibility distance in `kilometers`. If the metric system is used, it returns `kilometers`, otherwise `miles`.



`getTime` takes in the `unit system`, `current time` in UNIX, and the `timezone` difference of the location. If the metric system is used, it returns time in `24-hour format` (for example, 17:11), otherwise `12-hour format` (for example, 5:11).

`getAMPM` takes in the `unit system`, `current time` in UNIX, and the `timezone` difference of the location. If the metric system is used, it returns an empty `string` (no value), otherwise `AM` or `PM`, based on the given time.

`getWeekDay` takes in the `weatherData` object. Based on the current local time from the `weatherData`, it calculates the day of the week using the built-in `.getUTCDay()` function and then returns the name of the day from the titles array.

Notice we also imported necessary converting functions from `converters.js` at the top, since most of the helper functions depend on them.

## Creating Components

If we check the wireframe we made, we notice that we'll use ten components—`MainCard`, `ContentBox`, `Header`, `DateAndTime`, `Search`, `MetricsBox`, `MetricsCard`, `UnitSwitch`, `LoadingScreen` and `ErrorScreen`.

We'll create a separate file for each of them. Also, there will be a separate CSS module file for each (except for `LoadingScreen`) so we can style each component in the local scope.

You can create files manually by adding a new `components` folder in the project's root and start adding new files for each component (such as `MainCard.js` and `MainCard.module.css`) or you can use this terminal command, which will create all the necessary files automatically:

```
mkdir components && cd components && touch MainCard.js MainCard.module.css ContentBox.js ContentBox
↳.module.css Header.js Header.module.css DateAndTime.js DateAndTime.module.css Search.js
↳Search.module.css MetricsBox.js MetricsBox.module.css MetricsCard.js MetricsCard.module.css
↳UnitSwitch.js UnitSwitch.module.css LoadingScreen.js ErrorScreen.js ErrorScreen.module.css
```

Now let's add some code and styling rules for the component files.

### MainCard

The `MainCard` component will be the most important area, highlighting the information about the location and weather conditions. It will fill an entire left block of our app.

Include the following code in the `MainCard.js` file:

```
import Image from "next/image";
import { ctoF } from "../services/converters";
import styles from "./MainCard.module.css";

export const MainCard = ({
  city,
  country,
  description,
  iconName,
  unitSystem,
  weatherData,
}) => {
  return (
    <div className={styles.wrapper}>
      <h1 className={styles.location}>
        {city}, {country}
      </h1>
      <p className={styles.description}>{description}</p>
      <Image
        width="300px"
        height="300px"
        src={`/icons/${iconName}.svg`}
        alt="weatherIcon"
      />
      <h1 className={styles.temperature}>
        {unitSystem == "metric"
          ? Math.round(weatherData.main.temp)
          : Math.round(ctoF(weatherData.main.temp))}
        °{unitSystem == "metric" ? "C" : "F"}
      </h1>
      <p>
        Feels like{" "}
        {unitSystem == "metric"
          ? Math.round(weatherData.main.feels_like)
          : Math.round(ctoF(weatherData.main.feels_like))}
        °{unitSystem == "metric" ? "C" : "F"}
      </p>
    </div>
  );
};
```

As seen in the code snippet above, the `MainCard` component will display the data such as `city`, `country`, `description` of the weather conditions, as well as the actual `temperature`. It will also include a weather `icon`, and we'll use the NextJS built-in [image component](#) for that.

We set each of the required data as props, which we'll provide once we'll include the `MainCard`

component into `index.js` a bit later.

Then switch to `MainCard.module.css` and include the following styles:

```
.wrapper {
  text-align: center;
  padding: 30px;
}

.location {
  font-size: 38px;
  margin-bottom: 10px;
}

.description {
  font-size: 24px;
  margin-bottom: 20px;
}

.temperature {
  font-size: 84px;
}
```

For the wrapper, we've set all the included text to be centered and have some padding.

For the `Location`, `description`, and `temperature`, we've used a specific font size, with the first two also having a margin at the bottom.

## ContentBox

The `ContentBox` component will be a simple wrapper to hold all the children components—`Header`, `MetricsBox` and `UnitSwitch`.

Include the following code in the `ContentBox.js` file:

```
import styles from "../ContentBox.module.css";

export const ContentBox = ({ children }) => {
  return <div className={styles.wrapper}>{children}</div>;
};
```

Then switch to `ContentBox.module.css` and include the following styles:

```
.wrapper {
  background-color: rgb(247, 247, 247);
  padding: 30px;
}
```

We set the background color and some nice padding for the wrapper.

## Header

The `Header` component will be a simple wrapper for `DateAndTime` and `Search` components.

Include the following code in the `Header.js` file:

```
import styles from "./Header.module.css";

export const Header = ({ children }) => {
  return <div className={styles.wrapper}>{children}</div>;
};
```

Then switch to `Header.module.css` and include the following styles:

```
.wrapper {
  display: grid;
  grid-template-columns: 2fr 1fr;
  gap: 20px;
  margin-bottom: 20px;
}
```

We set the wrapper to use grid layout and split it into two columns using a 2:1 ratio. We've also provided some gap space between the two columns and added some margin below the wrapper.

## DateAndTime

The `DateAndTime` component will include the `name` of the weekday and the `time` displayed based on the selected option (24-hour format or 12-hour format).

Include the following code in the `DateAndTime.js` file:

```
import { getWeekDay, getTime, getAMPM } from "../services/helpers";
import styles from "./DateAndTime.module.css";

export const DateAndTime = ({ weatherData, unitSystem }) => {
```

```

return (
  <div className={styles.wrapper}>
    <h2>
      {`${getWeekDay(weatherData)}, ${getTime(
        unitSystem,
        weatherData.dt,
        weatherData.timezone
      )} ${getAMPM(unitSystem, weatherData.dt, weatherData.timezone)}`}
    </h2>
  </div>
);
};

```

We've made use of three helper functions we created earlier ( `getWeekDay()` , `getTime()` and `getAMPM()` ) and provided the necessary arguments as props, which we'll pass in later, once we include the `DateAndTime` component in `index.js` .

Then switch to `DateAndTime.module.css` and include the following styles:

```

.wrapper {
  display: flex;
  align-items: center;
}

```

We set the wrapper to have a flex layout and center the included `h1` children vertically.

## Search

The `Search` component will allow user interaction with the app.

Include the following code in the `Search.js` file:

```

import styles from "./Search.module.css";

export const Search = ({
  placeholder,
  value,
  onFocus,
  onChange,
  onKeyDown,
}) => {
  return (
    <input
      className={styles.search}

```

```
    type="text"
    placeholder={placeholder}
    value={value}
    onFocus={onFocus}
    onChange={onChange}
    onKeyDown={onKeyDown}
  />
);
};
```

We set the input `type` to `text` and set props for `placeholder`, `value`, `onFocus`, `onChange`, `onKeyDown`. We'll pass those in later, once we include the `Search` component in the `index.js` file.

Then switch to `Search.module.css` and include the following styles:

```
.search {
  height: 40px;
  font-size: 18px;
  font-family: "Varela Round", sans-serif;
  color: #303030;
  text-align: right;
  padding: 0 10px;
  border: none;
  border-radius: 10px;
}
```

We've defined the specific height to the search bar and set the `size`, `family`, `color`, and `alignment` for the text. We've also added some padding to the top and the bottom as well as removed the default border and set it to be rounded.

## MetricsCard

The `MetricsCard` component will present detailed information about the weather conditions. It will be used as a template to display conditions like `Humidity`, `Wind` and `Visibility`, as well as `Sunrise` and `Sunset` times.

Include the following code in the `MetricsCard.js` file:

```
import Image from "next/image";
import styles from "./MetricsCard.module.css";

export const MetricsCard = ({ title, iconSrc, metric, unit }) => {
```

```

return (
  <div className={styles.wrapper}>
    <p>{title}</p>
    <div className={styles.content}>
      <Image width="100px" height="100px" src={iconSrc} alt="weatherIcon" />
      <div>
        <h1>{metric}</h1>
        <p>{unit}</p>
      </div>
    </div>
  </div>
);
};

```

Each card will contain information about `title`, `metric` value, and the `unit` used. Similar to what we did for the `MainCard` component, we also included a weather `icon`, and we'll use the NextJS built-in image component for that.

Content will be passed into props once we include `MetricsCard` components in `MetricsBox.js`.

Then switch to `MetricsCard.module.css` and include the following styles:

```

.wrapper {
  background: rgba(255, 255, 255, 0.95);
  padding: 20px;
  text-align: right;
  border-radius: 20px;
}

.content {
  display: grid;
  grid-template-columns: 1fr 1fr;
}

```

The card wrapper will use a specific background color and some padding. Included text will be aligned right. The wrapper will also have a rounded border.

Included content will be divided into two columns, each with the same width.

## MetricsBox

The `MetricsBox` component will be a wrapper for the `MetricCard` component we created in the previous step. We'll include the cards directly in this component just to keep the `index.js` file less cluttered.

Include the following code in the `MetricsBox.js` file:

```
import { degToCompass } from "../services/converters";
import {
  getTime,
  getAMPM,
  getVisibility,
  getWindSpeed,
} from "../services/helpers";
import { MetricsCard } from "./MetricsCard";
import styles from "./MetricsBox.module.css";

export const MetricsBox = ({ weatherData, unitSystem }) => {
  return (
    <div className={styles.wrapper}>
      <MetricsCard
        title="Humidity"
        iconSrc="/icons/humidity.png"
        metric={weatherData.main.humidity}
        unit={"%"}
      />
      <MetricsCard
        title="Wind speed"
        iconSrc="/icons/wind.png"
        metric={getWindSpeed(unitSystem, weatherData.wind.speed)}
        unit={unitSystem == "metric" ? "m/s" : "m/h"}
      />
      <MetricsCard
        title="Wind direction"
        iconSrc="/icons/compass.png"
        metric={degToCompass(weatherData.wind.deg)}
      />
      <MetricsCard
        title="Visibility"
        iconSrc="/icons/binocular.png"
        metric={getVisibility(unitSystem, weatherData.visibility)}
        unit={unitSystem == "metric" ? "km" : "miles"}
      />
      <MetricsCard
        title="Sunrise"
        iconSrc="/icons/sunrise.png"
        metric={getTime(
          unitSystem,
          weatherData.sys.sunrise,
          weatherData.timezone
        )}
        unit={getAMPM(
          unitSystem,
          weatherData.sys.sunrise,
```



```

        weatherData.timezone
      )}
    />
    <MetricsCard
      title={"Sunset"}
      iconSrc={"/icons/sunset.png"}
      metric={getTime(
        unitSystem,
        weatherData.sys.sunset,
        weatherData.timezone
      )}
      unit={getAMPM(unitSystem, weatherData.sys.sunset, weatherData.timezone)}
    />
  </div>
);
};

```

We've passed in all the required props for `MetricsCard`. Notice we've also used converter function `degToCompass()` and helper functions, such as `getWindSpeed()`, `getVisibility()`, `getTime()`, `getAMPM()` to assist in the data processing.

Then switch to `MetricsBox.module.css` and include the following styles:

```

.wrapper {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
  gap: 20px;
  margin-bottom: 20px;
}

```

We've set the wrapper to use grid layout with three columns, each the same width. Then we've added a gap between the columns and set a margin below the wrapper.

## UnitSwitch

The `UnitSwitch` component will allow user interaction by letting them select their preferred unit system (metric or imperial) to view the weather conditions.

Include the following code in the `UnitSwitch.js` file:

```

import styles from "./UnitSwitch.module.css";

export const UnitSwitch = ({ onClick, unitSystem }) => {
  return (

```

```

<div className={styles.wrapper}>
  <p
    className={` ${styles.switch} ${
      unitSystem == "metric" ? styles.active : styles.inactive
    }`}
    onClick={onClick}
  >
    Metric System
  </p>
  <p
    className={` ${styles.switch} ${
      unitSystem == "metric" ? styles.inactive : styles.active
    }`}
    onClick={onClick}
  >
    Imperial System
  </p>
</div>
);
};

```

The `UnitSwitch` component will present unit system switch options and set either `active` or `inactive` classes based on the user interaction.

Then switch to `UnitSwitch.module.css` and include the following styles:

```

.wrapper {
  text-align: right;
}

.switch {
  display: inline;
  margin: 0 10px;
  cursor: pointer;
}

.active {
  color: green;
}

.inactive {
  color: black;
}

```

We've set all the text in the component to be aligned right. For the switch options, we've set them to be inline to display them in the same line and also add a margin to the top and bottom as well as improve the UX by changing mouse cursor to `pointer`.

Based on the explained logic on the `UnitSwitch.js` file, we've set the text of the selected unit system to be displayed in green, while the `inactive` will be black.

## LoadingScreen

The `LoadingScreen` component will be a simple component, returning just a loading message of our choice. We'll use conditional rendering in `index.js` to detect when it will be shown.

Include the following code in the `LoadingScreen.js` file:

```
export const LoadingScreen = ({ loadingMessage }) => <h1>{loadingMessage}</h1>;
```

## ErrorScreen

The `ErrorScreen` component will display the error message if the search request doesn't return any results. Similar to `LoadingScreen`, we'll use conditional rendering in `index.js` to detect when it will be shown.

Include the following code in the `ErrorScreen.js` file:

```
import styles from "./ErrorScreen.module.css";

export const ErrorScreen = ({ errorMessage, children }) => {
  return (
    <div className={styles.wrapper}>
      <h1 className={styles.message}>{errorMessage}</h1>
      {children}
    </div>
  );
};
```

As you'll notice, we've also allowed `children` components for the `ErrorScreen` component. We'll include the `Search` component in there once we've imported `ErrorScreen` component in the `index.js` file. This way, users will be able to perform the next search once the error message is shown on the screen.

Then switch to `ErrorScreen.module.css` and include the following styles:

```
.wrapper {
  max-width: 260px;
```

```
    text-align: center;
  }

  .message {
    margin-bottom: 30px;
  }
}
```

We've set the wrapper of `ErrorScreen` to not exceed a specific width and align the text in it. For the `message` presented, we've added a margin at the bottom so there's a space between it and the next `children` component.

## Implementing the Logic

To display the components we just created, we first need to import them into `index.js` and set the rendering logic as well as provide all the props necessary for the functionality.

Add the following code to the `index.js` file:

```
import { useState, useEffect } from "react";

import { MainCard } from "../components/MainCard";
import { ContentBox } from "../components/ContentBox";
import { Header } from "../components/Header";
import { DateAndTime } from "../components/DateAndTime";
import { Search } from "../components/Search";
import { MetricsBox } from "../components/MetricsBox";
import { UnitSwitch } from "../components/UnitSwitch";
import { LoadingScreen } from "../components/LoadingScreen";
import { ErrorScreen } from "../components/ErrorScreen";

import styles from "../styles/Home.module.css";

export const App = () => {
  // states and data fetch...

  const changeSystem = () =>
    unitSystem == "metric"
      ? setUnitSystem("imperial")
      : setUnitSystem("metric");

  return weatherData && !weatherData.message ? (
    <div className={styles.wrapper}>
      <MainCard
        city={weatherData.name}
        country={weatherData.sys.country}
        description={weatherData.weather[0].description}
      />
    </div>
  ) : <ErrorScreen />
}
```

```

        iconName={weatherData.weather[0].icon}
        unitSystem={unitSystem}
        weatherData={weatherData}
      />
    <ContentBox>
      <Header>
        <DateAndTime weatherData={weatherData} unitSystem={unitSystem} />
        <Search
          placeholder="Search a city..."
          value={cityInput}
          onFocus={(e) => {
            e.target.value = "";
            e.target.placeholder = "";
          }}
          onChange={(e) => setCityInput(e.target.value)}
          onKeyDown={(e) => {
            e.keyCode === 13 && setTriggerFetch(!triggerFetch);
            e.target.placeholder = "Search a city...";
          }}
        />
      </Header>
      <MetricsBox weatherData={weatherData} unitSystem={unitSystem} />
      <UnitSwitch onClick={changeSystem} unitSystem={unitSystem} />
    </ContentBox>
  </div>
) : weatherData && weatherData.message ? (
  <ErrorScreen errorMessage="City not found, try again!">
    <Search
      onFocus={(e) => (e.target.value = "")}
      onChange={(e) => setCityInput(e.target.value)}
      onKeyDown={(e) => e.keyCode === 13 && setTriggerFetch(!triggerFetch)}
    />
  </ErrorScreen>
) : (
  <LoadingScreen loadingMessage="Loading data..." />
);
};

export default App;

```

First, we've imported all the components we created, except `MetricsCard`, which we've already imported into `MetricsBox` to keep this file less cluttered.

Then we've created a handler function `ChangeSystem()`, which we've assigned to the `onClick` prop of the `UnitSwitch` component. Once the inactive unit system is clicked, the `unitSystem` state variable is changed, becoming the active system in use.

Next, we've created conditional rendering blocks for the app. The code block inside `weatherData`

`&& !weatherData.message` condition gets rendered if the response from the server is received and there's no error message.

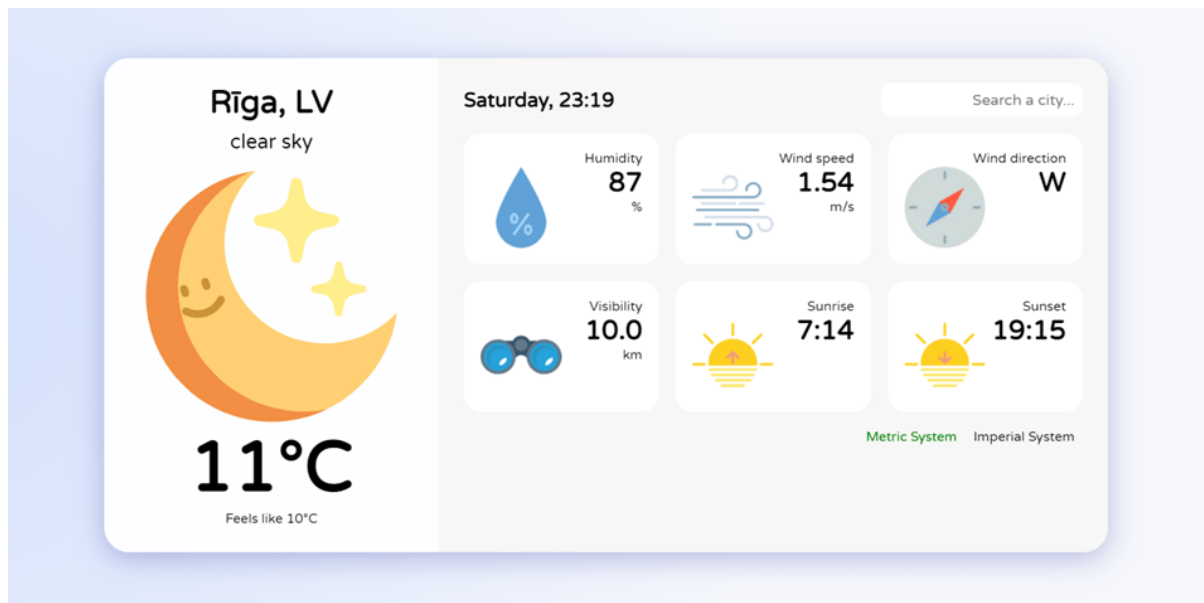
The code block inside `weatherData && weatherData.message` condition gets rendered if the response is received, but there's an error message, signaling that the user search query didn't return any results. In this state, the `ErrorScreen` component is presented to the user.

The third condition renders its contained code if none of the above conditions is true, meaning that the application is in a loading state when the request is sent, but the response is not yet received. While in this state, the `LoadingScreen` component is presented to the user.

Notice that we've also created a separate function `(e) => e.keyCode === 13 && setTriggerFetch(!triggerFetch)` for the `onKeyDown` prop of the `Search` component. This will change the `triggerFetch` state variable to the opposite value (the default was `true`) each time the `Enter` key is hit on the keyboard (the `Enter` key code is `13`, which you can check it out [here](#)).

Now check your terminal and see if your dev server is still running. If it isn't, run the command `npm run dev` again. Then open your browser.

You should be presented with a fully rendered and functional app.



## Adding Responsiveness

At this point, our app looks great, but it has one major drawback. It isn't adapted for use on

different devices, meaning it lacks responsiveness.

To fix that, we'll use [CSS media queries](#), which define the looks of the rendered elements for different resolutions, making it look great on any screen.

We'll revisit some CSS files and add some additional styling rules. Keep in mind that the media queries are usually added at the bottom of each stylesheet.

## Main App Wrapper

Add the following styling rules in the `Home.module.css` file:

```
@media only screen and (max-width: 950px) {
  .wrapper {
    grid-template-columns: 1fr;
    max-width: 600px;
    margin: 20px auto;
  }
}

@media only screen and (max-width: 600px) {
  .wrapper {
    margin: 0;
    border-radius: 0;
  }
}
```

This will ensure that the main wrapper of the entire app switches to the one-column layout for all the devices that are smaller than `950px`, meaning that the directly included children—`MainCard` and `ContentBox`—will be shown directly below each other.

If the screen width is smaller than `600px`, it will remove the top and bottom margin, as well as the border radius.

## Header

Add the following styling rules in the `Header.module.css` file:

```
@media only screen and (max-width: 520px) {
  .wrapper {
    grid-template-columns: 1fr;
    place-items: center;
  }
}
```

```
}
```

For any screen width smaller than `520px`, the `header` will use the one-column layout, meaning that the included children— `DateAndTime` and `Search` —will be shown directly below each other. We also set the rule that both children will be centered.

## Search

Add the following styling rules to the `Search.module.css` file:

```
@media only screen and (max-width: 520px) {  
  .search {  
    width: 100%;  
    text-align: center;  
  }  
}
```

For the device screens smaller than `520px`, we've set the search bar to be extended to the available width of the parent element `Header`. We've also set the rule so that the text is centered inside the search bar.

## MetricsCard

Add the following styling rules in the `MetricsCard.module.css` file:

```
@media only screen and (max-width: 475px) {  
  .content {  
    grid-template-columns: 1fr 2fr;  
  }  
}
```

For the device screens smaller than `475px`, we've adjusted the content so that the included children elements are shown in a two-column layout with a 1:2 width ratio.

## MetricsBox

Add the following styling rules in the `MetricBox.module.css` file:

```
@media only screen and (max-width: 600px) {  
  .wrapper {  
    grid-template-columns: 1fr 1fr;  
  }  
}
```



```
    }  
  }  
  
  @media only screen and (max-width: 475px) {  
    .wrapper {  
      grid-template-columns: 1fr;  
    }  
  }  
}
```

For the device screens smaller than `600px`, the `MetricBox` wrapper will use the two-column layout, meaning it will hold two `MetricCard` components side by side.

If the device screen is smaller than `475px`, the wrapper will then switch to the one-column layout, meaning that all the `MetricCard` elements will be shown directly below each other.

## UnitSwitch

Add the following styling rules in the `UnitSwitch.module.css` file:

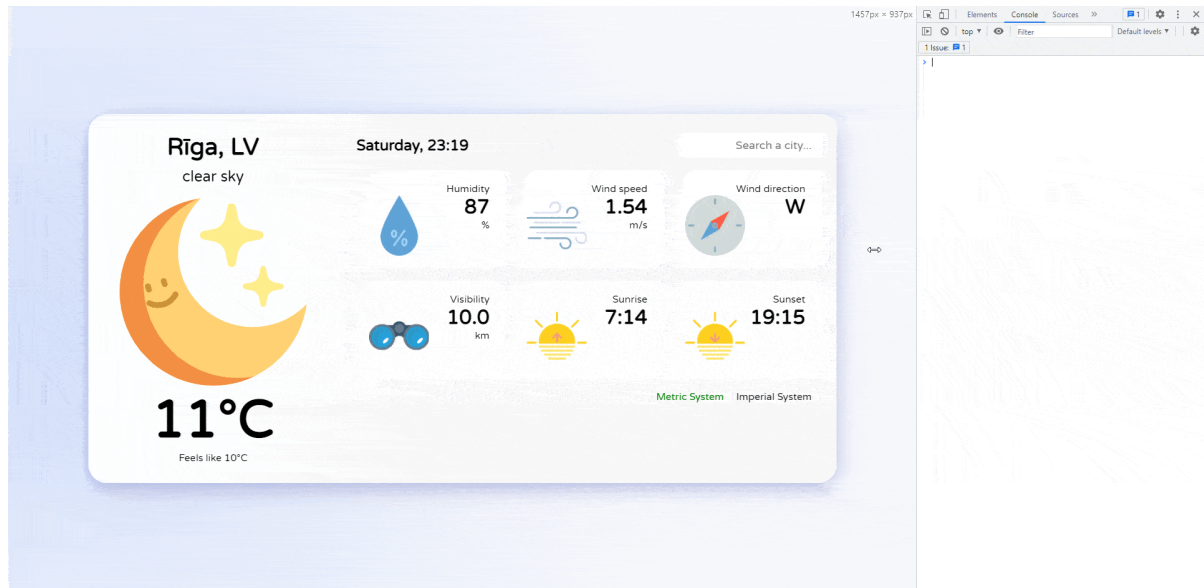
```
@media only screen and (max-width: 475px) {  
  .wrapper {  
    text-align: center;  
  }  
}  
  
@media only screen and (max-width: 335px) {  
  .wrapper {  
    display: grid;  
    grid-template-columns: 1fr;  
  }  
  
  .switch {  
    margin: 10px 0;  
  }  
}
```

For device screens smaller than `475px`, both of the unit switches will be centered horizontally while still being side by side.

For tiny screens that are less than `335px` wide, the wrapper will switch to a one-column layout, meaning both `switch` options will be shown directly below each other. We've also set some margin, so that there's some space between both `switch` elements.

At this point, you have made a fully functional app. Let's test it!

Open your browser and enter the developer tools (by pressing `F12` or manually). Now drag the developer tools sidebar and you should see the app readjusting to the width of the screen.



## Deploying the App

The last step is to deploy the app so that it can be accessed online. Since we're working with NextJS, the recommended approach would be to use Vercel.

In order to deploy it, we firstly need to push our code to [GitHub](#).

Log in to GitHub (or create a new account if you don't already have one). Select **Create a new repository** from the menu, choose a repository name (it could be "weather-app" or anything else you want), and click **Create repository**.

To push the app to the newly created repository, switch back to your terminal/code editor and run the following commands (replace `<username>` with your GitHub username and `<reponame>` with the name of your repository):

```
git remote add origin https://github.com/<username>/<reponame>.git
git push -u origin main
```

Then switch back to GitHub and check if the files of your project have appeared in the repository you created. If so, you've successfully committed your code.

The screenshot shows a GitHub repository page for 'madzadev edit comp degree'. At the top, there are navigation buttons: 'Go to file', 'Add file', 'Code', and 'Gitpod'. Below this is a table of files and folders with their commit messages and dates. On the right side, there are sections for 'About', 'Releases', and 'Packages'.

| File/Folder       | Commit Message                      | Time Ago     |
|-------------------|-------------------------------------|--------------|
| .vscode           | edit cards and switch               | 3 days ago   |
| components        | rename helpers                      | 3 days ago   |
| pages             | format code                         | 3 days ago   |
| public            | img rename                          | 6 days ago   |
| services          | edit comp degree                    | yesterday    |
| styles            | add box sizing                      | 3 days ago   |
| .env.example      | add weather icons                   | 2 months ago |
| .eslintrc         | Initial commit from Create Next App | 2 months ago |
| .gitignore        | fix add .env to .gitignore          | 2 months ago |
| README.md         | edit readme                         | 4 days ago   |
| package-lock.json | fix: add package-lock.json          | 2 months ago |
| package.json      | Initial commit from Create Next App | 2 months ago |

**About**  
No description, website, or topics provided.  
[Readme](#)

**Releases**  
No releases published  
[Create a new release](#)

**Packages**  
No packages published  
[Publish your first package](#)

Next, head to Vercel, create a new account (if you don't have one yet) and log in.

Then create a new project. You'll need to install Vercel for GitHub (access rights), so that Vercel can view your GitHub repositories.

Then find your project in the **Import Git Repository** panel, click **Import**, and you should be presented with the configuration wizard.

**Configure Project**

PROJECT NAME  
weather-app

FRAMEWORK PRESET  
Next.js

ROOT DIRECTORY  
./ Edit

► Build and Output Settings

▼ Environment Variables

| NAME                | VALUE (WILL BE ENCRYPTED) |                  |
|---------------------|---------------------------|------------------|
| OPENWEATHER_API_KEY | env_dot_local_value       | <span>Add</span> |

[Learn more about Environment Variables →](#)

Deploy

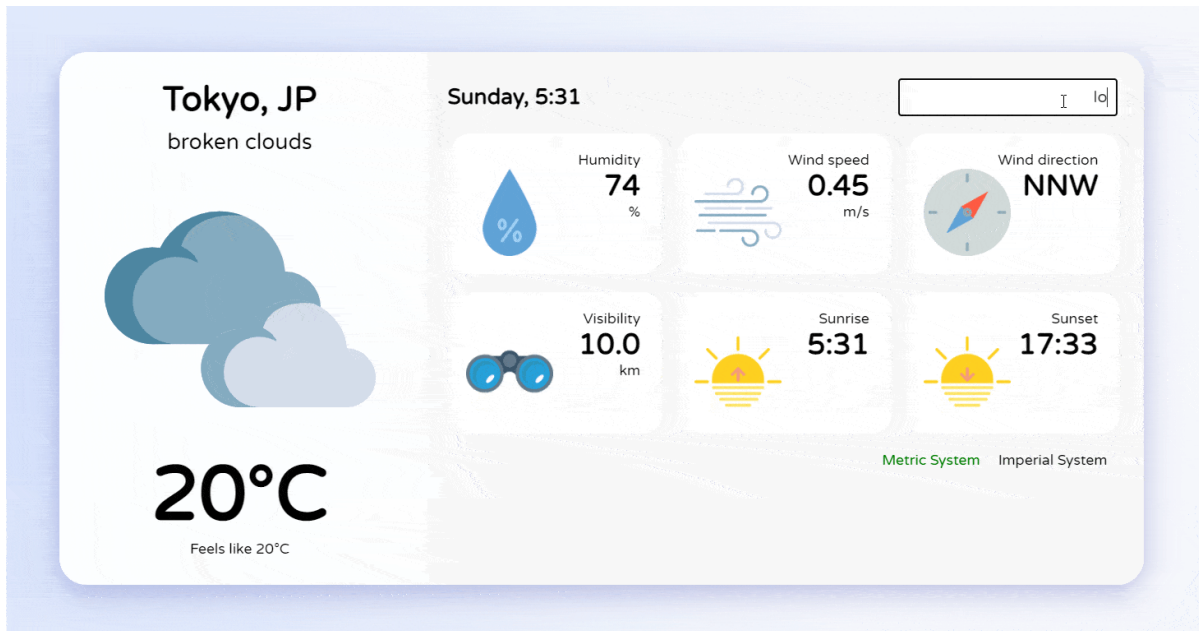
Vercel will detect the project name, build commands and root automatically, so you don't have to worry about that.

One thing you must do manually is to configure environment variables so that your deployed project on Vercel can access the OpenWeatherMap. Enter `OPENWEATHER_API_KEY` as a name and paste a key value from your local project's `.env.Local` file (see in the image above).

Once that's done, click **Deploy**. The build process shouldn't take longer than a minute.

After successful deployment, you'll be able to go to the dashboard of your project. Once there, click the **Visit** button, which will open the live URL of your project.

Congratulations! You've developed a fully functional weather app!



From now on, every time you push an update to GitHub, it will be automatically re-deployed on Vercel, meaning your live site will be in sync with the code on GitHub.

## Conclusion

In this tutorial, we've gone through the whole app creation process from planning to deployment. Hopefully you've learned a thing or two so you can further apply your skills in your future projects.

We picked the features, created the layout by constructing a wireframe, designed our own color scheme, and learned how to set up the project and configure the settings. We also made a fetch call to the external database and learned how to process and store the received data.

While working with components, we learned how to import and export them, use props and add local styling with CSS modules. We separated converters and helper functions from the main logic of the app and learned how to integrate them from the external files.

We also made sure the app looks great on every screen and deployed it, so it's accessible from anywhere, and the next time you want to check the weather you can use your own app on your desktop or mobile phone while you're on the go.

I've always believed that it's important to share knowledge around, so I encourage you to share this tutorial with your friends and give it a star in the [GitHub repo](#) if you liked it. Thanks for following along!