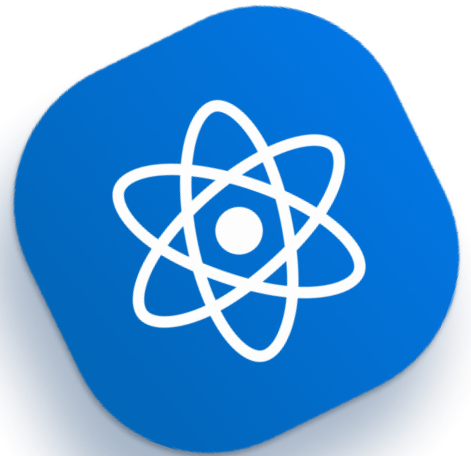


# 12 React Exercises for Beginners

---



12 beginner-friendly projects to help you  
build your **React** skills

Capela.dev

Version 0.1

This ebook is designed to help you learn the basics of frontend web development with React through a series of hands-on exercises.

Each exercise is a different project, with multiple steps that gradually increase in difficulty. By working through these exercises, you'll have a chance to train your foundation React skills and be well on your way to building your own web applications!

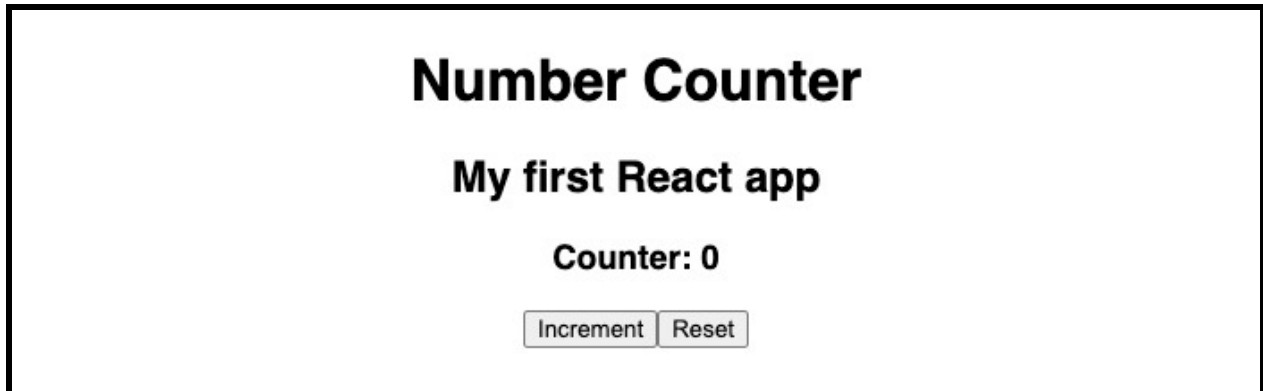
## React Exercises

1. Number Counter
2. Color Picker
3. Character Counter
4. Active Doors
5. Word Counter
6. Rock Paper Scissors
7. Fruit Search Filter
8. Email List
9. React Pokédex
10. Random Dog Images
11. Weather App
12. Reactflix

## Exercise 1: Number Counter

JSX / useState

**Let's build a React app to count numbers!** Start with [this code](#) as a boilerplate for the exercise.



### Step 1

Use the useState hook to keep the state of the counter.

### Step 2

Implement the logic of the increment button. Every time the user clicks the increment button, the counter should be incremented by 1.

### Step 3

Implement the reset button. Every time the user clicks the reset button, the counter should reset to 0.

### Step 4

Create a new decrement button and add the logic to decrement the counter by 1 when clicked.

## Step 5

Create a button with the text "Random" that, when clicked, sets the counter to a random value between 1 and 100. You can use the following function to generate a random number:

```
function getRandomNumberBetween(min, max) {  
  return Math.floor(Math.random() * (max - min + 1) + min);  
}
```

## Step 6

Don't allow the increment button to set the counter to more than 100.

## Step 7

Don't allow the decrement button to set the counter to less than 0.

■ ■ ■

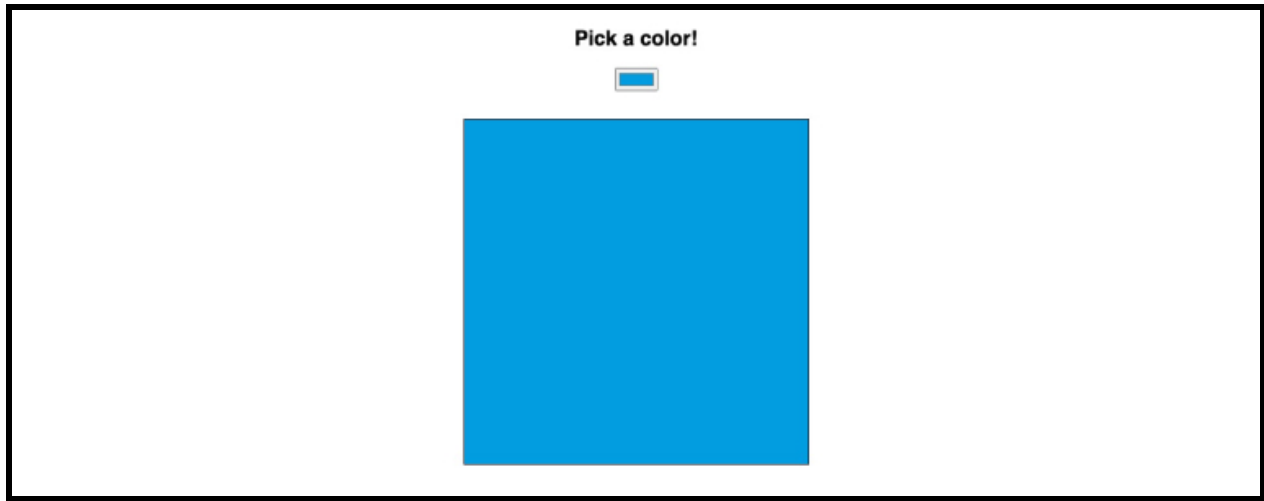
## Exercise 1 Solution

<https://codesandbox.io/s/twu9li?file=/src/App.jsx>

## Exercise 2: Color Picker

JSX / useState

**Let's build a simple color picker app using React!** Start with [this code](#) as a boilerplate for the exercise.



### Step 1

Take a look at the boilerplate code.

### Step 2

Show on the page the HEX value of the chosen color.

### Step 3

Add an input and a button to the page, and every time the user enters a HEX value on the input and clicks the button, the app sets the color entered on the input field as the chosen color.

## Step 4

When the user clicks the button, if the value on the input field is not a valid HEX color, show an error message informing the user that the value entered is incorrect.

For simplicity reasons, assume that a valid value for a HEX color is a string that starts with a # and has 7 characters total.

## Step 5

What about if the user selected 2 colors instead of just 1? Could we generate a gradient instead of a solid color in this case? 🤔



## Exercise 2 Solution

<https://codesandbox.io/s/wtt7p2?file=/src/App.jsx>

## Exercise 3: Character Counter

JSX / useState

**Let's build a React app to count characters!** Start with [this code](#) as a boilerplate for the exercise.

### Character counter

**Word:**  

---

Number of characters: 0

### Step 1

Switch the input border color to green when the entered word has more than 10 characters, and switch it back to black when it has 10 characters or less.

### Step 2

Every time the number of characters of the entered word is 0, show a proper warning message. When the number of characters of the entered word is 1 or more, remove this warning message. Use the `WarningMessage` component located at `/src/components/WarningMessage.jsx`.

### Step 3

Do not allow the user to enter spaces on the input. To remove spaces from a string you can use the following function:

```
function removeSpaces(string) {  
  return string.replaceAll(' ', '');  
}
```

### Step 4

Every time the entered word includes numbers, show an error message in red. Use the component ErrorMessage located at /src/components/ErrorMessage.jsx which contains the error message. You can use the following function to check if a string includes at least one number.

```
function hasNumber(string) {  
  return /\d/.test(string);  
}
```

### Step 5

Add a reset button with the text "Reset" which clears the input field when clicked.

### Step 6

Add a button with the text "Uppercase" which, when clicked, causes the input text to be all in UPPERCASE.

### Step 7

Change the app's background color to one of your choice.



## Step 8

Add a button with the text "Lowercase" which, when clicked, causes the input text to be all lowercase.

## Step 9

Create a component called SuccessMessage identical to the other error and alert message components, but with a green background and with the text "Valid word!". This component, which represents a success alert, must appear whenever the word entered does not contain numbers and has a number of characters greater than 0.

## Step 10

Add a square-shaped div, with height and width equal to the number of characters in the word times 10. For example, if the word is "windows" it has 7 characters, that is, multiplying by 10, we have 70, so your div will be a 70px by 70px square (this div must have a black background).

## Step 11

Add a color picker that allows the user to select the background color of the div you added in step 1. To select the color, you can use an `input[type="color"]` element as in the example below:

```
<input
  type="color"
  value={color}
  onChange={(event) => {
    console.log(`New color value: ${event.target.value}`);
  }}
/>
```

## Step 12

Remove the UPPERCASE and lowercase buttons. After removing them, add a single button that will act as a toggle between UPPERCASE and lowercase. If the last time you clicked it changed the text to UPPERCASE, now it will change it to lowercase, and vice versa. The text of the button must vary between "Uppercase" and "Lowercase" if it goes by the text UPPERCASE or lowercase, correspondingly.

## Step 13

If you haven't already, move the color picker code to a separate component called ColorPicker, and place it in the following path `/src/components/ColorPicker.jsx`.

■ ■ ■

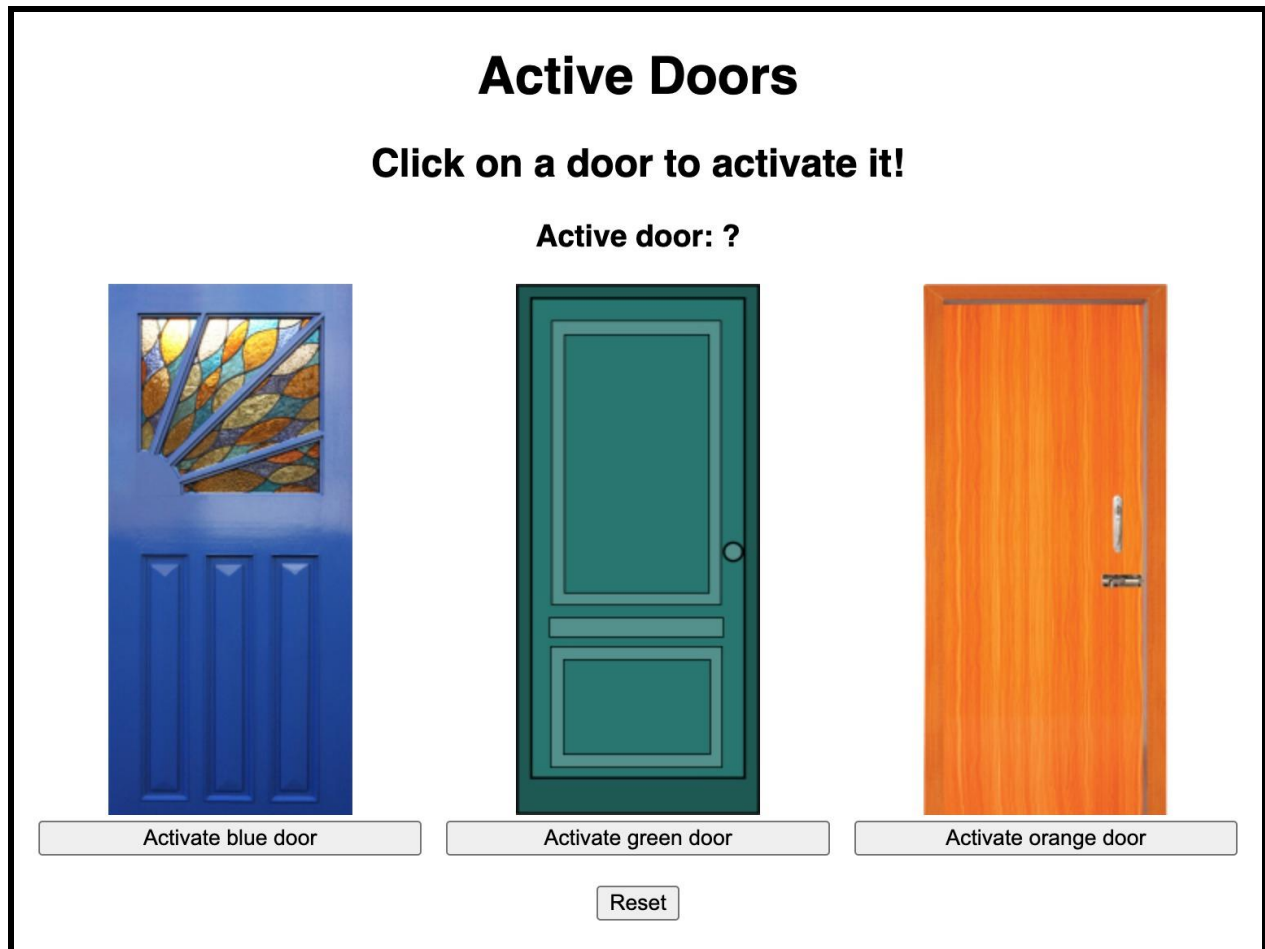
## Exercise 3 Solution

<https://codesandbox.io/s/301124?file=/src/App.jsx>

## Exercise 4: Active Doors

JSX / useState

Let's build a simple app that shows us which door is active using **React**! Start with [this code](#) as a boilerplate for the exercise.



### Step 1

Use the useState hook to keep the state of the active door.

## Step 2

Show which door is active depending on the state. If no door is active the text should say "Active door: none".

## Step 3

Modify the background color of the li element of the active door to the same color as the door.

## Step 4

Implement a reset button which when clicked, eliminates the active door, returning the text to "Active door: none".

## Step 5

Create an array with the 3 color strings "blue", "green", and "orange" and instead of repeating the li element 3 times (one for each door), iterate over the array and have it written in the code only once.



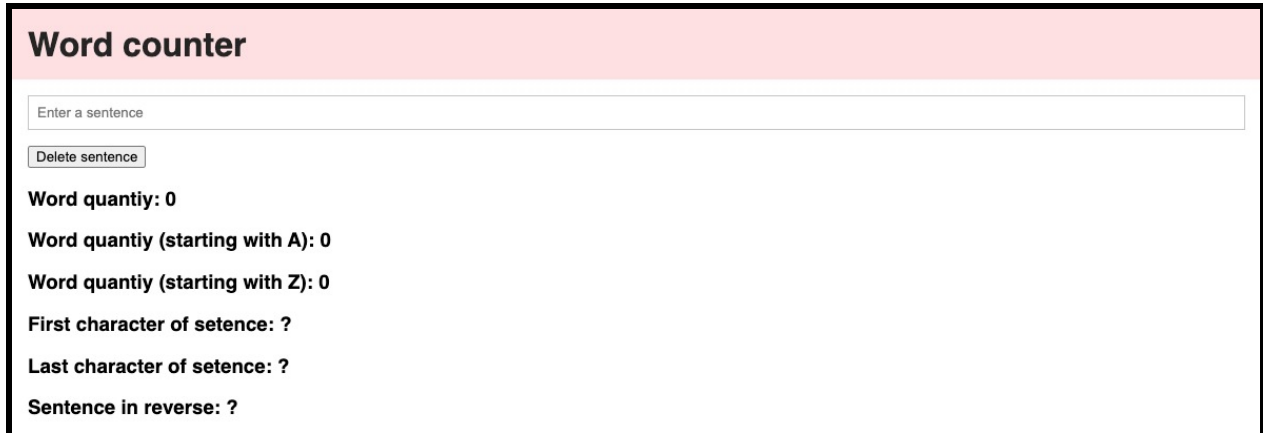
## Exercise 4 Solution

<https://codesandbox.io/s/gxo01q?file=/src/App.jsx>

## Exercise 5: Word Counter

JSX / useState

**Let's build a React app to count words!** Start with [this code](#) as a boilerplate for the exercise.



The screenshot shows a web application titled "Word counter" with a pink header. Below the header is a text input field with the placeholder "Enter a sentence". To the right of the input field is a button labeled "Delete sentence". Below the input field, there are several lines of text: "Word quanti: 0", "Word quanti (starting with A): 0", "Word quanti (starting with Z): 0", "First character of setence: ?", "Last character of setence: ?", and "Sentence in reverse: ?".

### Step 1

Make that when the user clicks on the "Delete sentence" button, all the text in the input is deleted.

### Step 2

Show the number of words in the entered sentence. Whenever the user writes a new sentence in the input, the number of words must be updated in real-time. To get the individual words of a string, you can try using the split method, for example:

```
const words = "This is my dog".split(" ");  
// words = ["This", "is", "my", "dog"]
```

### Step 3

Show the number of words starting with the letter "A" or "a" (uppercase or lowercase).

### Step 4

Show the number of words that end with the letter "Z" or "z" (uppercase or lowercase).

### Step 5

Show the first letter of the sentence.

### Step 6

Show the last letter of the sentence.

### Step 7

Show the sentence in reverse. For example for the phrase "I am learning React" it would show "tcaeR gninrael ma I".



## Exercise 5 Solution

<https://codesandbox.io/s/e2jj28?file=/src/App.jsx>

## Exercise 6: Rock Paper Scissors

JSX / useState / Components

**Let's build a rock-paper-scissors game with React!** Start with [this code](#) as a boilerplate for the exercise.



### Step 1

Whenever the user clicks the "Rock", "Paper", or "Scissors" button, it shows the image rock.png, paper.png, or scissors.png respectively. The images are in the /public/images/ folder.

### Step 2

Whenever the user clicks the "Random" button, it shows the rock, paper, or scissors image at random. Perhaps the following function will be useful:

```
function getRandomNumberBetween(min, max) {  
  return Math.floor(Math.random() * (max - min + 1) + min);  
}
```

### Step 3

Whenever someone clicks one of the blue "Rock", "Paper", or "Scissors" buttons, show a browser prompt for the user to confirm their action. If the user confirms, the button changes the icon in the game, if the user cancels, the button will have no effect. To open the browser confirmation prompt you can use the following method:

```
window.confirm('Do you really want to do that?');
```

### Step 4

Duplicate the div with `className="player"` from "Player 1" and create one for "Player 2". The "Random" buttons must work independently for each player. The blue "Rock", "Paper", or "Scissors" buttons should change both Players to rock, paper, or scissors, respectively.

### Step 5

Add a header under the blue buttons where we can see which of the two players is currently winning the game, or if they are both tied.

### Step 6

Whenever a player is losing, change the background color of the div with `className="player"` of that same player to "red".

### Step 7

Whenever a player is winning, change the background color of the div with `className="player"` of that same player to "green".



## Step 8

If you haven't done it yet, move the two divs with `className="player"` into a single component and use it twice, once for each player. This new component should be called `Player` and located in the following path `/src/components/Player.jsx`.



## Exercise 6 Solution

<https://codesandbox.io/s/kuyyeg?file=/src/App.jsx>

## Exercise 7: Fruit Search Filter

JSX / useState / Components / Lists

Let's build a React app that searches and filters different fruits! Start with [this code](#) as a boilerplate for the exercise.

### Fruit search filter

Search to filter fruits

Apple	Apricot	Banana	Blackberry
Blueberry	Cherry	Coconut	Fig
Grape	Kiwi	Lemon	Lime
Mango	Orange	Papaya	Peach

### Step 1

Make it so that whenever the user enters text in the search input, the fruits are filtered by name, and only the ones that match are shown. For example, for the search text "ap", the app should show "Apple", "Apricot", "Grape" and "Papaya".

### Step 2

Change the FruitCard component so that each card has the background color of the corresponding color. The fruit array is in a JSON file at `/src/data/fruits.json`.

### Step 3

Create a component called `FruitList` in `/src/components/FruitList.jsx`, responsible for receiving an array of fruits as a prop and rendering a `ul[className="fruitList"]` grid.

### Step 4

What happens when what the user searches for doesn't match any fruit in our array? Create a component called `EmptyList` in `/src/components/EmptyList.jsx` and render it whenever the user's search doesn't match any fruit. This component should have an `h2` element with the text "Whoops, no results!", and a `p` element with the text "No results found for: {searchQuery}", where `searchQuery` is the search string of the user.

### Step 5

Add a button with the text "Reset" that erases the text entered by the user.

### Step 6

Whenever the search text has more than 20 characters, show a browser prompt to the user with the message "Try with a smaller search query!". To display a browser prompt with a custom message you can use the following function:

```
const message = "Hello world!";  
window.confirm(message);
```

### Step 7

Whenever the user clicks on a fruit card, it shows the description of that fruit in a browser prompt, identical to the previous step, but with the format "{fruitName}: {fruitDescription}", where `fruitName` is the name of the fruit that was clicked and `fruitDescription` is the description of that same fruit.

## Step 8

Whenever the user clicks on a fruit, instead of displaying its info on a browser prompt, display it in a custom modal component named `FruitModal`. This component must be in `/src/components/FruitModal.jsx` and have 3 elements: an `h1` with the name of the fruit, a `p` with the description of the fruit, and a button with the text "Close" which will close the modal whenever it is clicked.

## Step 9

In the same modal, add a link with the text "Search on Google". When this link is clicked, it opens the search page for that same fruit on Google in a new tab. To search for a specific fruit on Google you can use the following URL passing the desired `q` query param:

`https://google.com/search?q=banana`

■ ■ ■

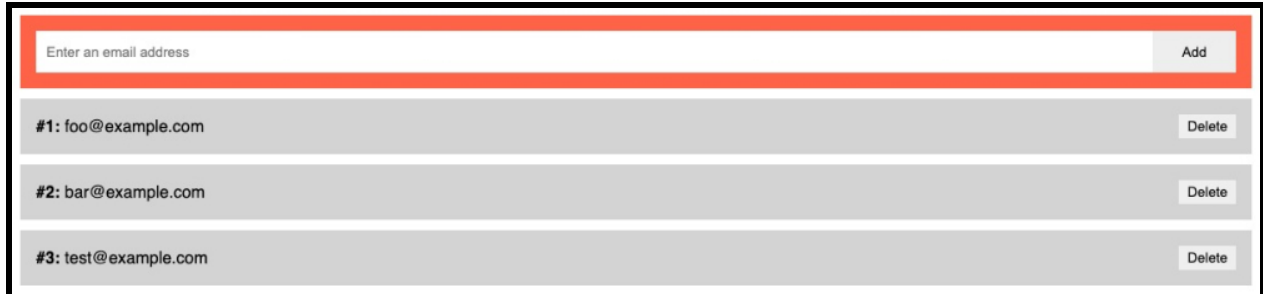
## Exercise 7 Solution

<https://codesandbox.io/s/tz39re?file=/src/App.jsx>

## Exercise 8: Email List

[JSX](#) / [useState](#) / [Components](#) / [Lists](#) / [useEffect](#)

**Let's build a React app to manage a list of emails!** Start with [this code](#) as a boilerplate for the exercise.



### Step 1

Develop an app that allows the user to manage email addresses, by adding and deleting them from a list.

### Step 2

Use [localStorage](#) to persist the data between page refreshes.

■ ■ ■

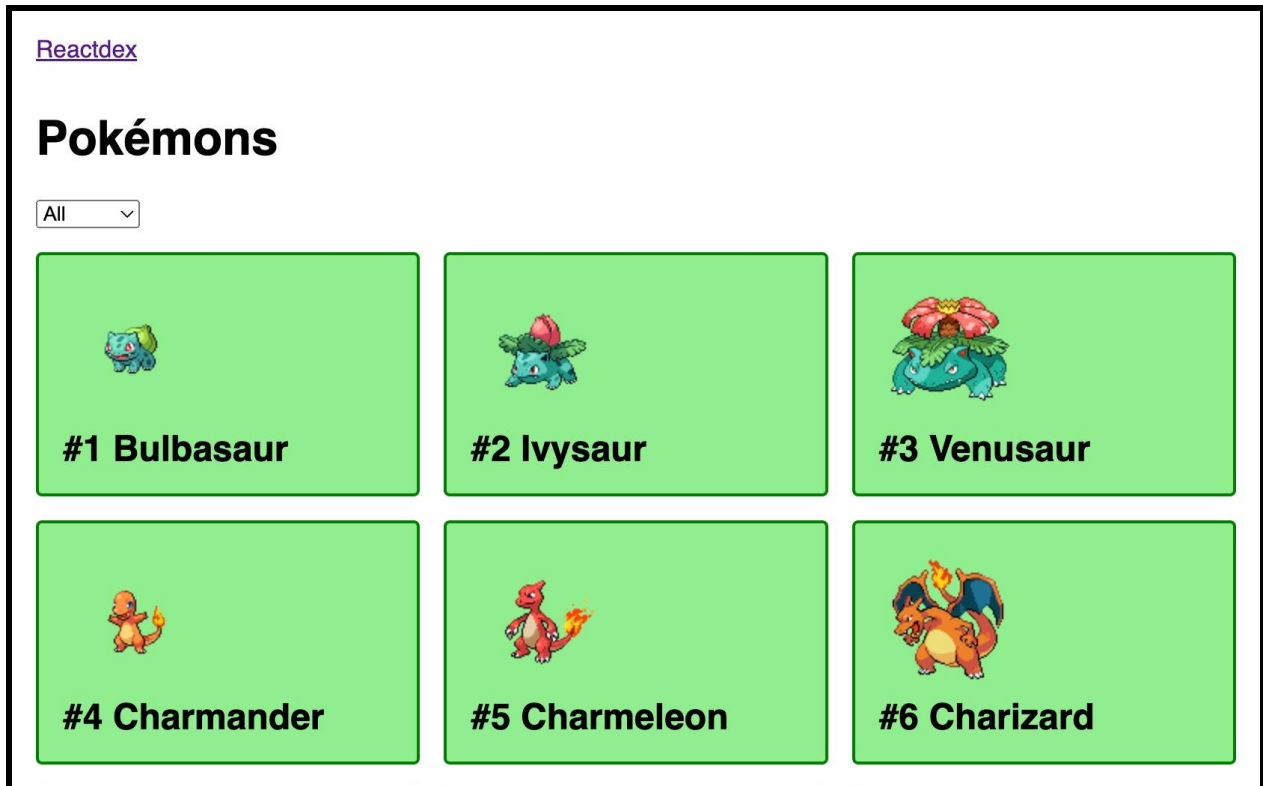
### Exercise 8 Solution

<https://codesandbox.io/s/er2zu8?file=/src/App.jsx>

## Exercise 9: React Pokédex

JSX / useState / Components / Lists / useEffect

Let's build a pokédex app using React! Start with [this code](#) as a boilerplate for the exercise.



### Details

Build a React app similar to [this one](#). Use this [pokemons.json](#) file as the database.

■ ■ ■

### Exercise 9 Solution

<https://codesandbox.io/s/jgq4rp?file=/src/App.jsx>

## Exercise 10: Random Dog Images

[JSX](#) / [useState](#) / [Components](#) / [Lists](#) / [useEffect](#) / [Async Requests](#)

**Let's build a React app that shows the user a random dog image when a button is clicked!** Start with [this code](#) as a boilerplate for the exercise.



### Details

To do this, we are going to use the [Dog API](#), which provides a collection of open-source dog pictures over HTTP endpoints. The endpoint you should use is this one and it returns a JSON structured as follows:

**GET `https://dog.ceo/api/breeds/image/random`**

```
{
  "message": "https://images.dog.ceo/breeds/spaniel-irish/n02102973_220.jpg",
  "status": "success"
}
```

## Step 1

Have a button that when clicked properly calls the API endpoint and shows the user the returned dog image.

## Step 2

Implement a loading state, for when the app is waiting for a response from the API.

- The loading state can be the string Loading... of a GIF image showing a loading spinner.

## Step 3

Use components to organize your app. You should have two components, a `<GenerateImageButton />` component for the button, and a `<DogImage />` component to show the image of the dog.

## Step 4

If we are waiting for an API response, we shouldn't be able to click the button again. Implement a disabled state on the button when we are waiting for an API response.

## Step 5

What should happen if the request fails? Implement a proper error message to handle this case.

- The error message should be the string "Whoops, something went wrong!" within a red box.
- You should write this in a separate component named `<ErrorMessage />`.



## Step 6

When the app starts, there is no dog image, because no one has clicked the button yet. Write some code in order to show a random dog image when the app starts without the need for a button click.



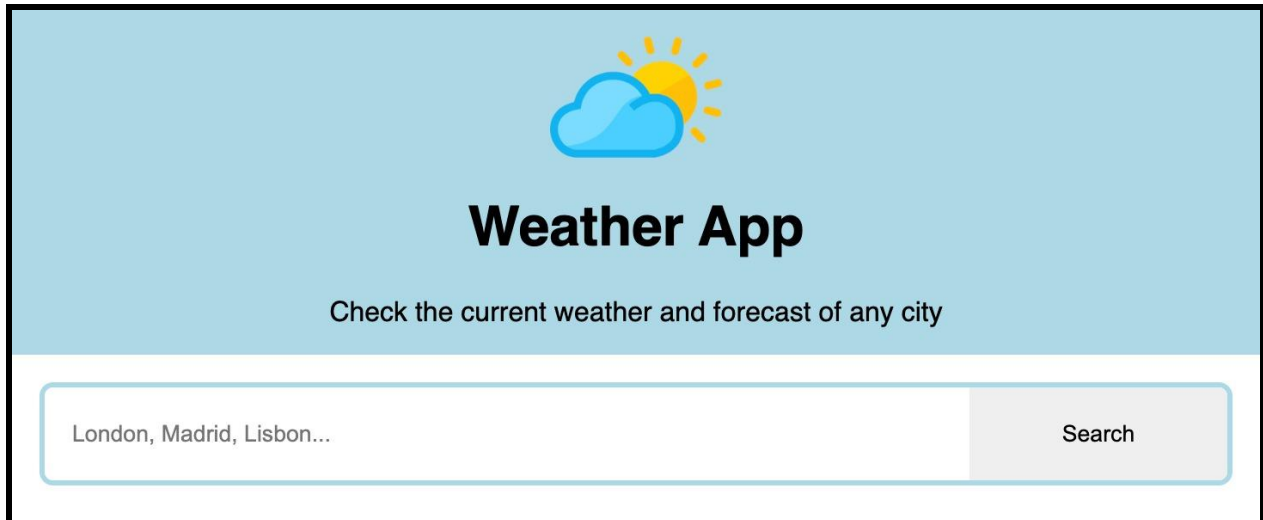
## Exercise 10 Solution

<https://codesandbox.io/s/9zndxw?file=/src/App.jsx>

## Exercise 11: Weather App

[JSX](#) / [useState](#) / [Components](#) / [Lists](#) / [useEffect](#) / [Async Requests](#)

**Let's build a weather app that checks the current weather and forecast for cities worldwide using React!** Start with [this code](#) as a boilerplate for the exercise.



### Details

To do this, we are going to use this API endpoint, which returns weather info for multiple world cities in a JSON format. The endpoint you should use is this one:

GET

`https://goweather.herokuapp.com/weather/${encodeURIComponent(city)}`

Lisbon:

`https://goweather.herokuapp.com/weather/Lisbon`

San Francisco:

`https://goweather.herokuapp.com/weather/San%20Francisco`

That returns a JSON structured as follows:

```

{
  "temperature": "19 °C",
  "wind": "22 km/h",
  "description": "Partly cloudy",
  "forecast": [
    {
      "day": "1",
      "temperature": "25 °C",
      "wind": "33 km/h"
    },
    {
      "day": "2",
      "temperature": "23 °C",
      "wind": "28 km/h"
    },
    {
      "day": "3",
      "temperature": "27 °C",
      "wind": "26 km/h"
    }
  ]
}

```

## Step 1

Build the app, so that when the user writes a city name on the input field and clicks the button, you properly call the API endpoint with the provided city name and show the current weather below for that city.

- Use state to manage the input text.
- Use state to store the weather data that is returned.
- When you have results, show the city name, temperature, wind speed, and weather description for the current day.

## Step 2

What does the API return when you input an invalid city name? Show an error message every time the API returns invalid data.

### Step 3

If you didn't do it yet, refactor your app to use components. You should have at least 3 components:

- `<Hero />`: Containing the header content.
- `<SearchForm />`: Containing the input text and search button.
- `<WeatherResult />`: Containing the current weather result.
- `<ErrorMessage />`: Containing the error message.

### Step 4

Show a loading state with the text Loading... when the app is waiting for an API response.

### Step 5

Disable the search button if the input text is empty.

### Step 6

Within the JSON result, the “forecast” node is an array that contains information about the weather for the next 3 days. Calculate and show on the `<WeatherResult />` component, which one of the next 3 days has the following:

- Highest temperature
- Lowest temperature
- Highest wind speed
- Lowest wind speed

### Step 7

Set your city as the default location, and fetch the weather forecast for that city every time the app starts (same as at every new full page reload).

## Step 8

Refactor your app to use multiple pages. You should have a homepage, where only the input and search buttons are shown, and a result page where you can show the current weather for that city.

- `/`: Homepage containing the input and search button.
- `/about`: A page with some static text.
- `/result/:city`: Page showing the results for that specific city. For example `/result/Lisbon`, or `/result/San%20Francisco`.

Add links in the footer of the web app to the homepage, about page, and to the following cities: London, Lisbon, Madrid, and New York.

## Step 9

Add a cookie banner component saying that your app doesn't store any user data. Besides some text, your `<CookieBanner />` should have a button to accept. You should show the cookie banner to every user who hasn't clicked the accept button yet. To persist if a user has clicked it yet or not use the [localStorage web API](#).

■ ■ ■

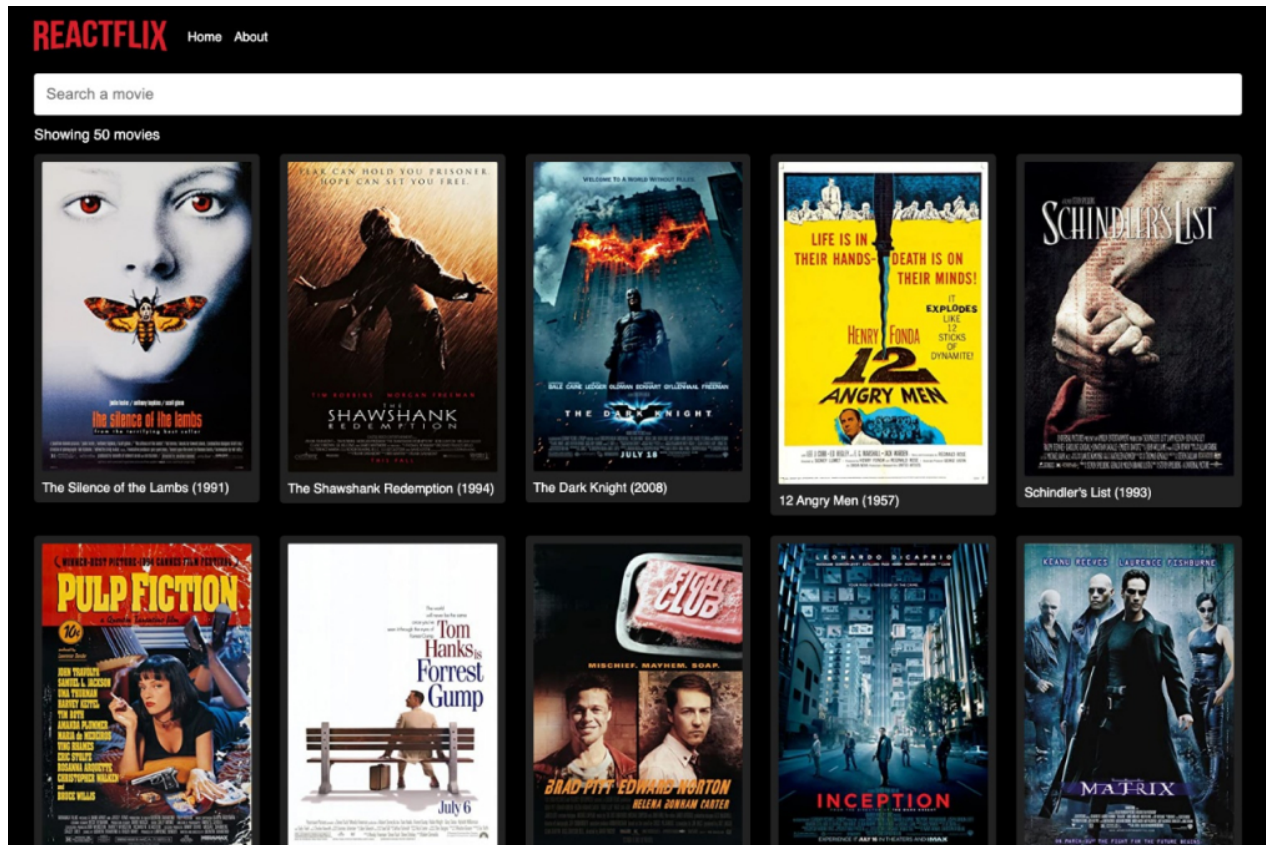
## Exercise 11 Solution

<https://codesandbox.io/s/mnm3qx?file=/src/App.jsx>

## Exercise 12: Reactflix

JSX / useState / Components / Lists / useEffect

Let's build a Netflix frontend clone app using React! Start with [this code](#) as a boilerplate for the exercise.



### Details

Build a React app similar to [this one](#). Use this [movies.json](#) file as the database.

■ ■ ■

### Exercise 12 Solution

<https://codesandbox.io/s/u5sifc?file=/src/App.jsx>