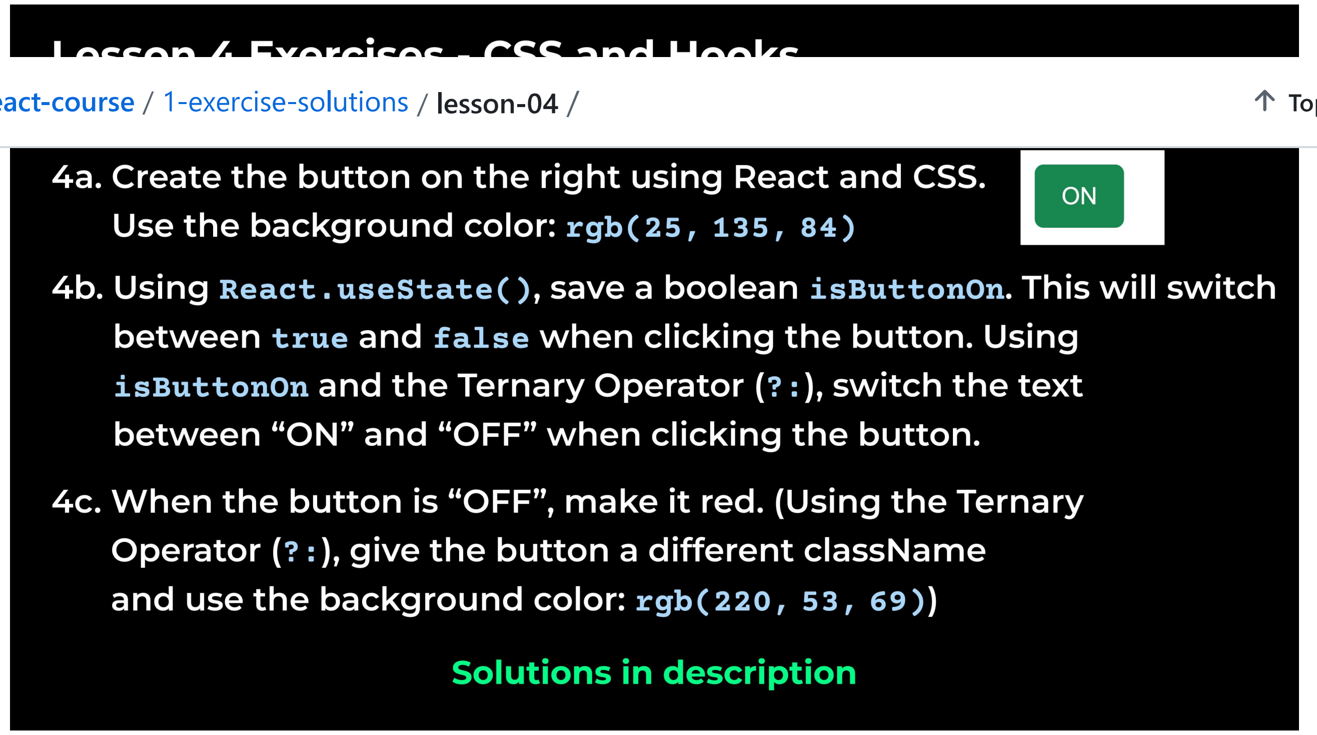


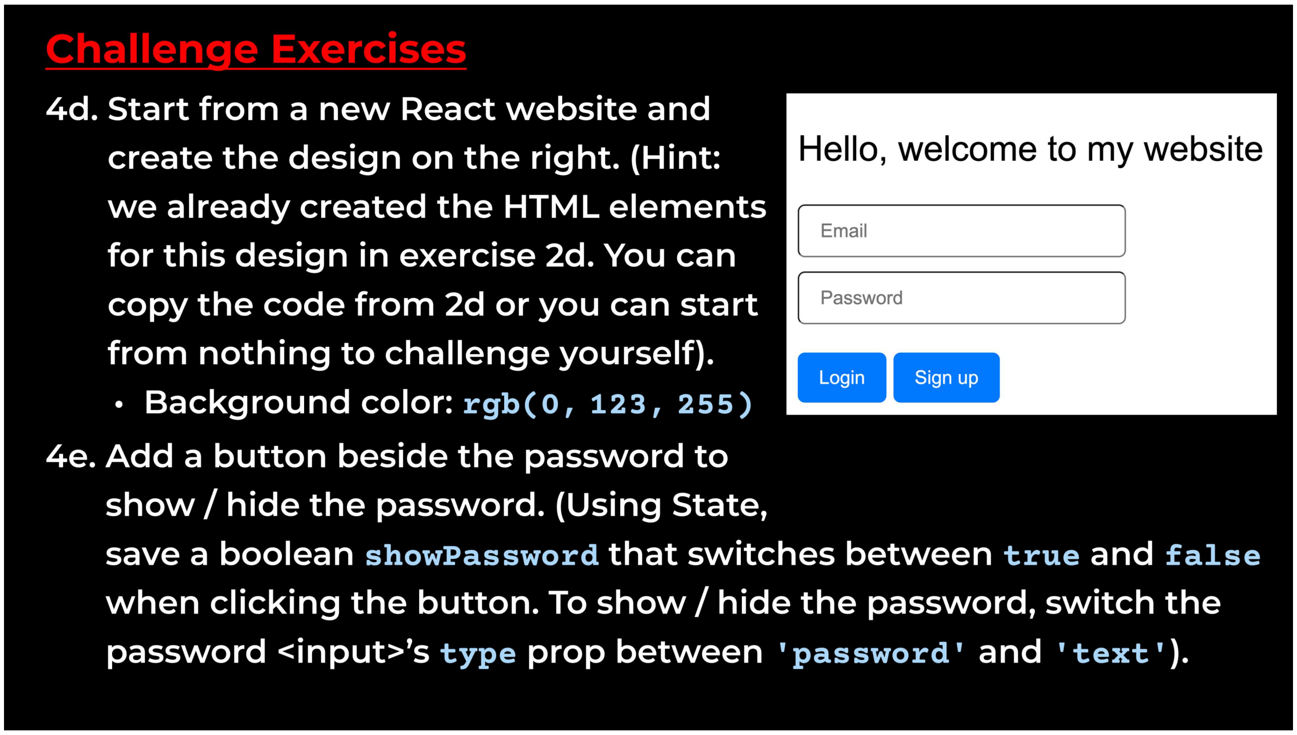
Lesson 4 Exercises - CSS and Hooks

4a. Create the button on the right using React and CSS. Use the background color: `rgb(25, 135, 84)`

4b. Using `React.useState()`, save a boolean `isButtonOn`. This will switch between `true` and `false` when clicking the button. Using `isButtonOn` and the Ternary Operator (`?:`), switch the text between “ON” and “OFF” when clicking the button.

4c. When the button is “OFF”, make it red. (Using the Ternary Operator (`?:`), give the button a different className and use the background color: `rgb(220, 53, 69)`)

Solutions in description



Challenge Exercises

4d. Start from a new React website and create the design on the right. (Hint: we already created the HTML elements for this design in exercise 2d. You can copy the code from 2d or you can start from nothing to challenge yourself).

- Background color: `rgb(0, 123, 255)`

Hello, welcome to my website
Email
Password
Login
Sign up

4e. Add a button beside the password to show / hide the password. (Using State, save a boolean `showPassword` that switches between `true` and `false` when clicking the button. To show / hide the password, switch the password <input>’s `type` prop between `'password'` and `'text'`).

4f. We'll create the clock from lesson 1 exercises using hooks:

- First, load the DayJS external library: `https://unpkg.com/supersimpledev/dayjs.js`
- Using `React.useState()` save a string `time`. To get the current time as a string, you can use `dayjs().format('HH:mm:ss')`
- Using `React.useEffect()` run this code once after the component is created: `setInterval(() => { }, 1000)` (this code runs a function every 1000 milliseconds or 1 second).
- In the function that we give to `setInterval`, get the time again using `dayjs().format('HH:mm:ss')` and update the State.

4g. Add `console.log('run code');` inside `setInterval` and check how many times it runs. Then, remove the dependency array (`[]`) from `React.useEffect()` and check how many times it runs.

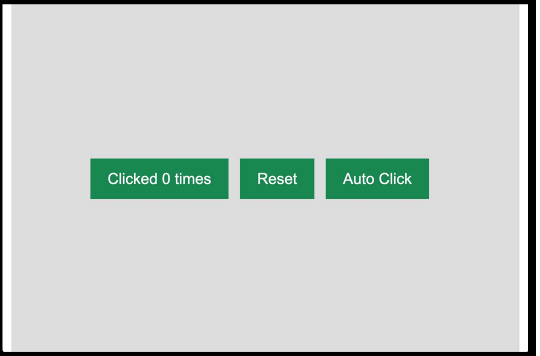
4h. Create the counter app below (you can use the code for exercise 3c as a starting point or start from an empty React website).

- Note: do not put the counter in a separate component for now.
- Create a "Reset" button that resets the count to 0.
- Create an "Auto Click" button. When clicked, your code will click the counter every second (use `setInterval`). To click a button with code, first save the button element in your JavaScript (use `useRef`). Once you have the button element, run `buttonElem.click()`.

(Normally, you can just update the count directly using State, instead of clicking the button with code. This is just to practice `useRef`)

4i. Style the counter using this design:

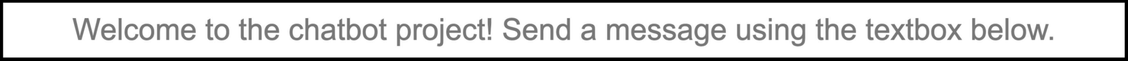
- Background color: `rgb(25, 135, 84)`
- To center the buttons horizontally, put the buttons in a flexbox and use `justify-content: center;`
- To center the buttons vertically, first, make the flexbox the same height as the browser (use `height: 100vh;`)
- Add `background-color: rgb(222, 222, 222);` to the flexbox. Notice the default margin of 8px around the `<body>` element. Remove this default margin from the top and bottom.
- Now that the flexbox is the same height as the browser, center the elements vertically using `align-items: center;`



<u>Do the next exercises in the Chatbot Project</u>

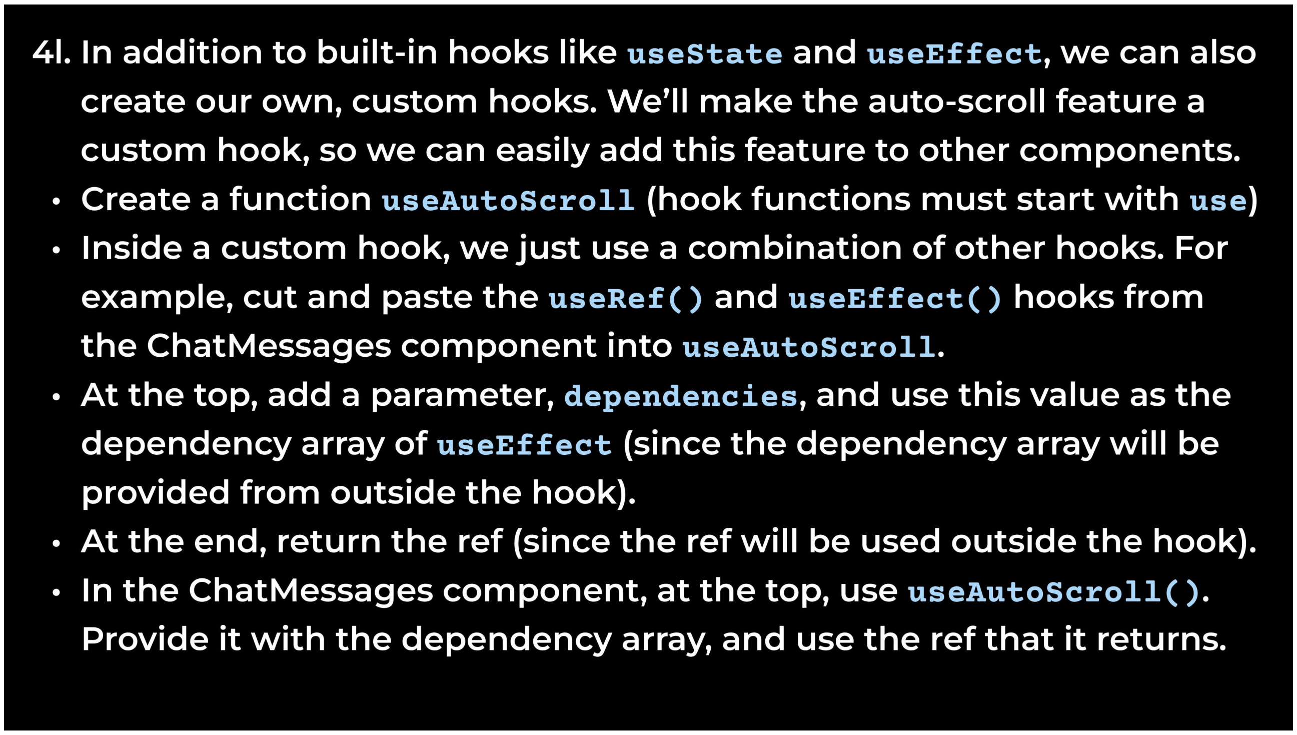
4j. Remove the default chat messages at the start of the app.

- If there are no chat messages (check `chatMessages.length === 0`), display a welcome message (to center it, use `text-align: center`)


Welcome to the chatbot project! Send a message using the textbox below.

4k. Add a loading spinner when the Chatbot is loading a response.

- First, make the Chatbot display a "Loading..." message before giving a response (see solutions for exercise 3k and 3l if you haven't done this).
- Download the loading spinner image: `https://supersimple.dev/images/loading-spinner.gif`
- In the loading message, make the `message` property an `<img>` instead of `'Loading...'`
- Style this `<img>` with `height: 40px; margin: -15px;`



4l. In addition to built-in hooks like `useState` and `useEffect`, we can also create our own, custom hooks. We'll make the auto-scroll feature a custom hook, so we can easily add this feature to other components.

- Create a function `useAutoScroll` (hook functions must start with `use`)
- Inside a custom hook, we just use a combination of other hooks. For example, cut and paste the `useRef()` and `useEffect()` hooks from the ChatMessages component into `useAutoScroll`.
- At the top, add a parameter, `dependencies`, and use this value as the dependency array of `useEffect` (since the dependency array will be provided from outside the hook).
- At the end, return the ref (since the ref will be used outside the hook).
- In the ChatMessages component, at the top, use `useAutoScroll()`. Provide it with the dependency array, and use the ref that it returns.