How hooks work

You've gotten an overview of hooks and focused on the most popular and crucial one, useState(). Let's see it in action.

Studying useState

1. Edit repository.ts. In it you'll find exported functions to communicate with the data server. Read through them to understand how they're working. Especially note that they're returning promises.

2. Edit Orders.tsx. Notice how it called getOrders() and, in the .then(), it sets the orders in state by calling setOrder() with the return value from getOrders().

3. Edit Register.tsx. Note all of the state variables. We could have made one big object the only state variable here, keeping the useState()s much cleaner. But what would have to be done in each <input>'s onChange?

Keeping ids unique with useId

HTML tags often use the id attribute, particularly when there's a <label> involved because the label points to its target by id. We literally need that to be compliant with WCAG guidelines for a11y. But React components always have the potential to be placed multiple times in the same scene. That's a problem if any HTML tag in it has an id. Why? Because ids must be unique on a page.

Let's use useId to make our ids unique.

4. Edit Register.tsx Note that there are a lot of inputs and a lot of labels.

Frankly, this would be fine because we know Register.tsx will never be used again. But let's use rigor and write clean code. Let's make the ids unique.

5. Import useId from react.

6. Add this to the component.

```
const id = useId();
```

We have a unique id. Let's use it.

7. Change the username label and input like this. Add ${id} in a JavaScript template.

```
<div>
  <label htmlFor={`username${id}`}>Username</label>
  <input id={`username${id}`} onChange={e => setUsername(e.target.value)}
         value={username} />
</div>
```

8. Do the same for every form field in the component.

9. Run and test. In the browser, inspect the form fields and examine the unique id.

10. Now do the same in the Login form on Login.tsx

useRef to get a reference to a DOM element

useRef is great for getting a reference to a DOM element and then remembering it across re-renders.

Try to register a new user. It works fine but let's say that the toast message is considered insufficient. Let's say that we want to pop up a big <dialog>. We can create that dialog in the JSX, but that dialog won't appear until we get a reference to it and call its showModal() method. React isn't very strong with low-level DOM manipulation.

But useRef will allow us to get a reference to that <dialog> that will be remembered across renders. We can then respond to the registration event by showing the <dialog>.

11. Edit Register.tsx.

12. Add a <dialog> to the JSX. Here's a start:

```
<dialog>
  <p>
    Welcome, {first}!
  </p>
  <button onClick={_ => dialogRef.current?.close()}>
    Close
  </button>
</dialog>
```

13. Near the top, add a useRef:

```
const dialogRef = useRef<HTMLDialogElement>(null);
```

14. Add this ref to your dialog:

```
<dialog ref={dialogRef}>
```

15. Find where we're registering the user. After it has returned successfully, open your dialog with

```
dialogRef.current?.showModal()
```

16. Run and test. When you register a new user, you should see your dialog pop up and should be able to dismiss it with the close button.

17. Bonus! If you're ahead of schedule, fill the dialog out with some detail; a message about the new registration. Put some styling in as well.