Introduction

In this lab we are going to get comfortable working with forms and public APIs.

An HTML form element represents a document section containing interactive controls for submitting information. In the example below we created a simple form containing only one input representing a Name.

```
<form>
  <label>
    Name:
    <input type="text" name="name" />
  </label>
  <input type="submit" value="Submit" />
</form>
```

Controlled Components

In React, Controlled Components are those in which form's data is handled by the component's state. It takes its current value through props and makes changes through callbacks like onClick,onChange, etc. A parent component manages its own state and passes the new values as props to the controlled component.

In the form elements are either the typed ones like textarea, input or the selected one like radio buttons or checkboxes. Whenever there is any change made it is updated accordingly through some functions that update the state as well.

```
export default function Form() {
    const [name, setName] = React.useState('');

    function handleChange(event) {
        setName(name => event.target.value);
    }

    function handleSubmit(event) {
        alert('A name was submitted: ' + name);
        event.preventDefault();
    }

    return (
        <form onSubmit={handleSubmit}>
            <label>
                Name:
                <input type="text" value={name} onChange={handleChange} />
            </label>
            <button type="submit" value="Submit" />
        </form>
    );
}
```

In most cases it is recommended to use Controlled Components to implement forms.

preventDefault()

By default, the browser will refresh the page when a form submission event is triggered. We generally want to avoid this in React.js applications because it would cause us to lose our state.

To prevent the default browser behavior, we have to use the preventDefault() method on the event object.

```
function handleSubmit(event) {
  alert('A name was submitted: ' + name);
  event.preventDefault();
}
```

Text Area

A <textarea> element in HTML:

```
<textarea>
  DSS rules!
</textarea>
```

In React, a <textarea> uses a value attribute instead. This way, a form using a <textarea> can be written very similarly to a form that uses a single-line input:

```
<form onSubmit={handleSubmit}>
  <label>
  Feedback Laborator:
    <textarea value={value} onChange={handleChange} />
  </label>
  <button type="submit" value="Submit" />
</form>
```

Select

In HTML, <select> creates a drop-down list. For example, this HTML creates a drop-down list of our previous labs:

```
export default function Form() {
    const [value, setValue] = React.useState('lab3')


    return (
            <select value={value} onChange={handleChange}>
                <option value="lab1">Lab 1</option>
                <option value="lab2">Lab 2</option>
                <option value="lab3">Lab 3</option>
                <option value="lab4">Lab 4</option>
            </select>
    );
}
```

Note that the “Lab 3” option is initially selected, because of its initial state.

Handling Multiple Inputs

When you need to handle multiple controlled input elements, you can add a name attribute to each element and let the handler function choose what to do based on the value of event.target.name.

```
const MyComponent = () => {
   const [inputs, setInputs] = useState({});
   const handleChange = e => setInputs(prevState => ({ ...prevState, [e.target.name]: e.target.value }));

   return (
     <div>
      <input name="field1" value={inputs.field1 || ''} onChange={handleChange} />
      <input name="field2" value={inputs.field2 || ''} onChange={handleChange} />
     </div>
   )
}
```

Form Validation

Form Validation is the process used to check if the information provided by the user is correct or not (eg: if an email containts '@'). There are two types of validation:

- Client Side: Validation is done in the browser
- Server Side: Validation is done on the server

Client-side validation is further categorized as:

- Built-in: Uses HTML-based attributes like required, type, minLength, maxLength, pattern, etc.
- JavaScript-based: Validation that's coded with JavaScript.

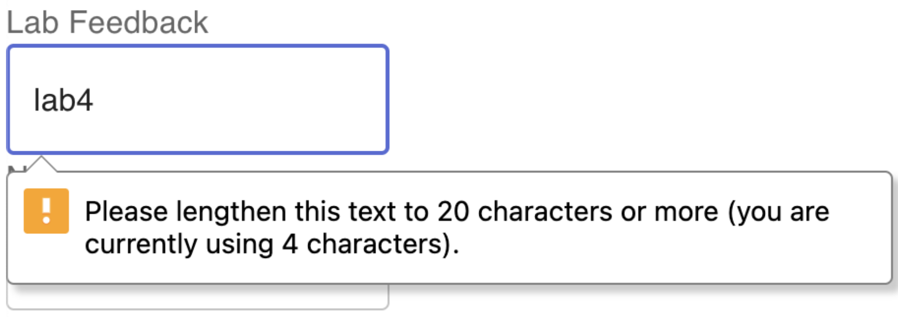
Validation HTML-based attributes:

- required: the fields with this attribute must be filled.
- type: i.e a number, email address, string, etc.
- minLength: minimum length for the text data string.
- maxLength: maximum length for the text data string.

Example:

```
<form>
  <label for="feedback">Lab Feedback</label>
  <input
    type="text"
    id="feedback"
    name="feedback"
    required
    minLength="20"
    maxLength="40"
  />
  <label for="name">Name:</label>
  <input type="text" id="name" name="name" required />
  <button type="submit">Submit</button>
</form>
```

With these validation checks in place, when a user tries to submit an empty field for Name, it gives an error that pops right in the form field. Similarly, the feedback must be between 20 and 40 characters long.



Please lengthen this text to 20 characters or more (you are currently using 4 characters).

JavaScript-based Form Validation

JavaScript offers an additional level of validation along with HTML native form attributes on the client side.

```
    function validateFormWithJS() {
        const name = nameRef.current.value;
        const feedback = feedbackRef.current.value;

        if (!name) {
            alert('Please enter your name.')
            return false
        }

        if (feedback.length < 20) {
            alert('Feedback should be at least 20 characters long.')
            return false
        }
    }

    return(
    <form onSubmit={validateFormWithJS}>
      <label for="feedback">Lab Feedback</label>
      <input
        type="text"
        id="feedback"
        name="feedback"
        ref={feedbackRef}
      />
      <label for="name">Name:</label>
      <input ref={nameRef} type="text" id="name" name="name" required />
      <button type="submit">Submit</button>
    </form>
    )
```

React useRef() hook

In most cases, we recommend using controlled components to implement forms. In a controlled component, form data is handled by a React component. The alternative is uncontrolled components, where form data is handled by the DOM itself.

To write an uncontrolled component, instead of writing an event handler for every state update, you can use a ref to get form values from the DOM.

useRef returns a mutable ref object whose .current property is initialized to the passed argument (initialValue). The returned object will persist for the full lifetime of the component.

You might be familiar with refs primarily as a way to access the DOM. If you pass a ref object to React with <div ref={myRef} />, React will set its .current property to the corresponding DOM node whenever that node changes.

```
function GetTextAreaDataUsingUseRef() {
  const inputEl = React.useRef();
  const onButtonClick = () => {
    // `current` points to the mounted text input element
    console.log(inputEl.current.value);
  };
  return (
    <div>
      <input ref={inputEl} type="text" />
      <button onClick={onButtonClick}>Focus the input</button>
    </div>
  );
}
```

Promises

The Promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

Here, we create a promise that will resolve after 300ms.

```
const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("foo");
  }, 300);
});
```

The Promise.resolve() method “resolves” a given value to a Promise. If the value is a promise, that promise is returned; if the value is a thenable, Promise.resolve() will call the then() method with two callbacks it prepared; otherwise the returned promise will be fulfilled with the value.

A Promise that is resolved with the given value, or the promise passed as value, if the value was a promise object. It may be either fulfilled or rejected — for example, resolving a rejected promise will still result in a rejected promise.

```
const promise1 = Promise.resolve('lab4');

promise1.then((value) => {
  console.log(value);
  // expected output: lab4
});
```

Async/Await

An async function is a function declared with the async keyword, and the await keyword is permitted within it. The async and await keywords enable asynchronous, promise-based behavior to be written in a cleaner style, avoiding the need to explicitly configure promise chains.

```
async function foo() {
  await 'lab4';
}
```

Is similar to:

```
function foo() {
  return Promise.resolve('lab4').then(() => undefined);
}
```

The async and await keywords are tools to manage promises.

If you mark a function as async then:

- its normal return value is replaced with a promise that resolves to its normal return value.
- you may use await inside it

If you use await on the left hand side of a promise, then the containing function will go to sleep until the promise settles. Execution outside the async function will continue while it sleeps (i.e. is is not halted).

The await keyword is only valid inside async functions within regular JavaScript code. If you use it outside of an async function's body, you will get a SyntaxError.

Axios

Axios is a promise-based HTTP Client for node.js and the browser. It can run in the browser and nodejs with the same codebase. On the server-side it uses the native node.js http module, while on the client (browser) it uses XMLHttpRequests.

We are gonna use Axios to make HTTP requests.

Example GET Request using AXIOS:

```
axios.get('https://dummyjson.com/users')
  .then(function (response) {
    // handle success
    console.log(response);
  });
```

Example POST Request using AXIOS:

```
axios.post('https://dummyjson.com/users/add', {
    firstName: 'Prenume',
    lastName: 'Dumitrescu',
    age: 30,
    /* other user data */
  })
  .then(function (response) {
    console.log(response);
  });
```