- ⏱️ 28 minutes
- ✅ Completed

React Forms

As you've learned in earlier lessons, HTML forms are an essential and ubiquitous part of the web. Forms are used to search, create resources (i.e. account, posts), update resources, and more. Learning how to create forms using React is an invaluable skill for you to learn and practice.

When you finish this article, you should be able to:

- Create a React function component containing a simple form
- Define controlled inputs with the `useState` hook for different form inputs
- Implement form validations

Set up for following along

If you want to follow along, create a React application using `create-react-app` with the `@appacademy/simple` template:

```
npx create-react-app contact-us-form --template @appacademy/simple --use-npm
```

Creating a simple form

To learn how to create an HTML form in React, you'll create a `ContactUs` function component that'll contain a simple "Contact Us" form. The form will initially contain just three fields:

- Name - The name of the user filling out the form
- Email - The user's email
- Phone - The user's phone number

Defining the `render` method

To start, add a function component named `ContactUs` and render the HTML form:

```
// ./src/components/ContactUs/index.js
```

```
function ContactUs() {
  return (
    <div>
      <h2>Contact Us</h2>
      <form>
        <div>
          <label htmlFor='name'>Name:</label>
          <input id='name' type='text' />
        </div>
        <div>
          <label htmlFor='email'>Email:</label>
          <input id='email' type='text' />
        </div>
        <div>
          <label htmlFor='phone'>Phone:</label>
          <input id='phone' type='text' />
        </div>
        <button>Submit</button>
      </form>
    </div>
  );
}

export default ContactUs;
```

So far, there's nothing particularly interesting about this form. The only thing that looks different from regular HTML is that the `<label>` element's `for` attribute is `htmlFor` in React.

If you're following along, be sure to update your React application's entry point to render the `ContactUs` component:

```
// ./src/index.js
import React from 'react';
import ReactDOM from 'react-dom';
import ContactUs from './components/ContactUs';

ReactDOM.render(
  <React.StrictMode>
    <ContactUs />
  </React.StrictMode>,
  document.getElementById('root')
);
```

At this point, you can run your application (`npm start`) and view the form in the browser. You can even fill out the form, but currently the component doesn't know what the form input values are. To keep track of each of the input values, you will need to initialize and maintain component state.

Adding state to the component

To add state to the `ContactUs` component, import `useState` from React. Initialize three state variables, `name`, `email`, and `phone` as empty strings. Then use them to set the `value` attributes on the corresponding form field `<input>` elements:

```
// ./src/components/ContactUs/index.js
import { useState } from 'react';

function ContactUs() {
  const [name, setName] = useState('');
  const [email, setEmail] = useState('');
  const [phone, setPhone] = useState('');

  return (
    <div>
      <h2>Contact Us</h2>
      <form>
        <div>
          <label htmlFor='name'>Name:</label>
          <input id='name' type='text' value={name} />
        </div>
        <div>
          <label htmlFor='email'>Email:</label>
          <input id='email' type='text' value={email} />
        </div>
        <div>
          <label htmlFor='phone'>Phone:</label>
          <input id='phone' type='text' value={phone} />
        </div>
        <button>Submit</button>
      </form>
    </div>
  );
}

export default ContactUs;
```

If you try navigating to the browser and refreshing now, then you will get a warning in your browser's dev tools console saying, "You provided a `value` prop to a form field without an `onChange` handler. This will render a read-only field." Also, if you try typing in any of the fields, they don't update anymore. This is because the `value` attribute for each input will always be an empty string.

For example, the name field will always be an empty string because the `value` attribute on that field is set to the `name` state variable. The `name` state variable is initialized to an empty string and is never updated. To fix this, you need to update the `name` state variable whenever the user types into the field.

When a form field element value is changed, the associated component state variable needs to be updated. Adding or removing a character within an `<input>` element raises the `onChange` event, which makes it a natural choice for keeping the component state in sync:

```
<input id='name' type='text' onChange={(e) =>
setName(e.target.value)} value={name} />
```

Remember that when an event is raised, the associated event handler method is called and passed an instance of `event` object type. A reference to the element that raised the event is available through the `event` object's `target` property. Using the reference to the form field element, you can retrieve the current value as the `value` property on the `target` object.

Using the same approach to add an `onChange` event handler to the "Email" and "Phone" form fields gives you this:

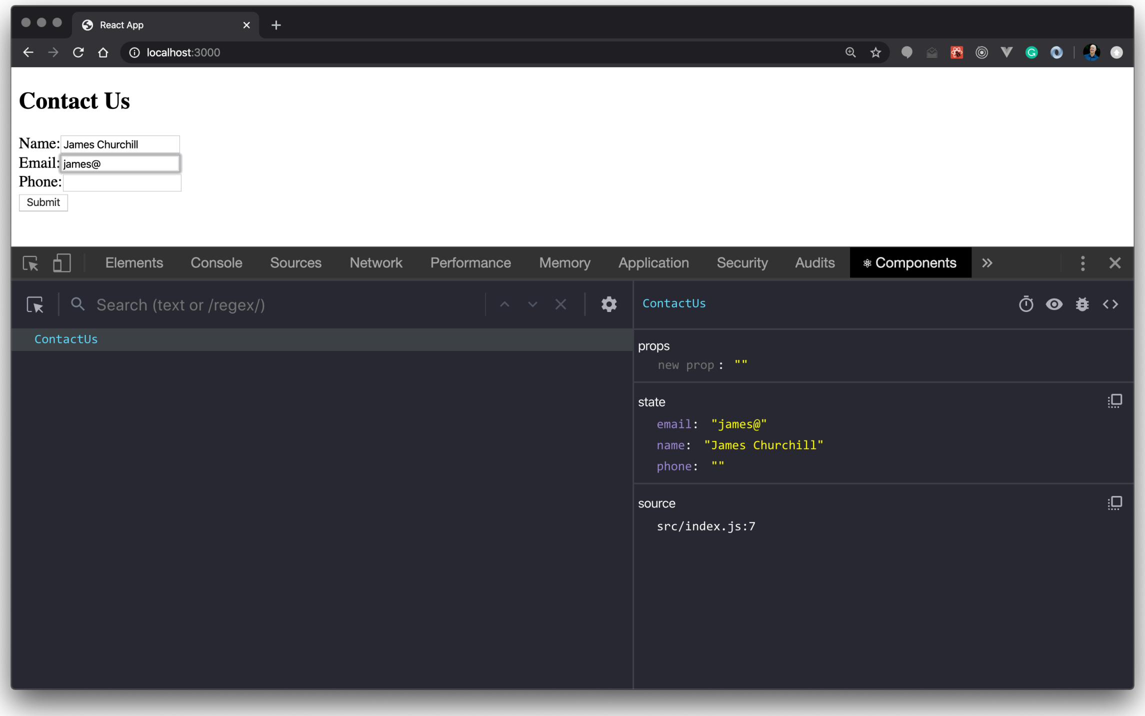
```
// ./src/components/ContactUs/index.js
import { useState } from 'react';

function ContactUs() {
  const [name, setName] = useState('');
  const [email, setEmail] = useState('');
  const [phone, setPhone] = useState('');

  return (
    <div>
      <h2>Contact Us</h2>
      <form>
        <div>
          <label htmlFor='name'>Name:</label>
          <input
            id='name'
            type='text'
            onChange={(e) => setName(e.target.value)}
            value={name}
          />
        </div>
        <div>
          <label htmlFor='email'>Email:</label>
          <input
            id='email'
            type='text'
            onChange={(e) => setEmail(e.target.value)}
            value={email}
          />
        </div>
        <div>
          <label htmlFor='phone'>Phone:</label>
          <input
            id='phone'
            type='text'
            onChange={(e) => setPhone(e.target.value)}
            value={phone}
          />
        </div>
        <button>Submit</button>
      </form>
    </div>
  );
}

export default ContactUs;
```

If you view the form again in the browser and open the React Developer Tools, you can see the component's state update as you type within each of the form fields (i.e the `<input>` elements).



```
ContactUs
props
  new prop : ""
state
  email: "james@"
  name: "James Churchill"
  phone: ""
source
  src/index.js:7
```

Handling form submissions

Now that the `ContactUs` component is initializing and updating state when form field values are changed, it's time to handle form submissions! To start, create an `onSubmit` function and attach it to the `onSubmit` event listener for the form. Within the `onSubmit` event handler prevent the default behavior so that the page doesn't reload:

```
const onSubmit = (e) => {
  // Prevent the default form behavior
  // so the page doesn't reload.
  e.preventDefault();
}
<form onSubmit={onSubmit}>
```

Then use the `name`, `email`, and `phone` values from state to create a new `contactUsInformation` object literal:

```
const onSubmit = (e) => {
  // Prevent the default form behavior
  // so the page doesn't reload.
  e.preventDefault();

  // Create a new object for the contact us information.
  const contactUsInformation = {
    name,
    email,
    phone,
    submittedOn: new Date()
  };

  // For now, just log the contact us information to the
console
  // though ideally, you'd persist this information to a
database
  // using a REST API.
  console.log(contactUsInformation);
}
```

Notice that an additional property, `submittedOn`, is being added to the `contactUsInformation` object literal to indicate the date/time that the information was submitted. Ideally, the `contactUsInformation` object would be persisted to a database using a REST API, but for now, you'll just log the object to the console.
Now that the form submission has been processed, reset the `name`, `email`, and `phone` values to empty strings:

```
const onSubmit = (e) => {
  // Prevent the default form behavior
  // so the page doesn't reload.
  e.preventDefault();

  // Create a new object for the contact us information.
  const contactUsInformation = {
    name,
    email,
    phone,
    submittedOn: new Date()
  };

  // For now, just log the contact us information to the
console
  // though ideally, you'd persist this information to a
database
  // using a REST API.
  console.log(contactUsInformation);

  // Reset the form state.
  setName('');
  setEmail('');
  setPhone('');
}
```

Putting all of that together gives you this:

```
// ./src/components/ContactUs/index.js
import { useState } from 'react';

function ContactUs() {
  const [name, setName] = useState('');
  const [email, setEmail] = useState('');
  const [phone, setPhone] = useState('');

  const onSubmit = (e) => {
    e.preventDefault();
    const contactUsInformation = {
      name,
      email,
      phone,
      submittedOn: new Date()
    };

    console.log(contactUsInformation);
    setName('');
    setEmail('');
    setPhone('');
  };

  return (
    <div>
      <h2>Contact Us</h2>
      <form onSubmit={onSubmit}>
        <div>
          <label htmlFor='name'>Name:</label>
          <input
            id='name'
            type='text'
            onChange={(e) => setName(e.target.value)}
            value={name}
          />
        </div>
        <div>
          <label htmlFor='email'>Email:</label>
          <input
            id='email'
            type='text'
            onChange={(e) => setEmail(e.target.value)}
            value={email}
          />
        </div>
        <div>
          <label htmlFor='phone'>Phone:</label>
          <input
            id='phone'
            type='text'
            onChange={(e) => setPhone(e.target.value)}
            value={phone}
          />
        </div>
        <button>Submit</button>
      </form>
    </div>
  );
}

export default ContactUs;
```

If you run your application again and view the form in the browser, you can fill out each form field and click "Submit" to submit the form. Notice that the page doesn't reload! And if you look in the developer tool's console, you'll see an object containing your contact information.

Controlled components

Congrats! You've completed your first simple React form! In doing so, you used what's known as *controlled components*.
HTML form elements naturally maintain their own state. For example, an `input` element will track the state of the value that's typed within it (without any help from libraries like React). But a React component keeps track of its own internal state. To keep a component's state as the "one source of truth", `onChange` event handlers are used on form field elements to update the component's state when a form element's state has changed.
This approach of making the component's state the "one source of truth" is called *controlled components*. Inputs in a controlled component are called *controlled inputs*.

To help understand how this works, here's an overview of the flow:

- A user types a character within a form `<input>` element;
- The `<input>` element's `onChange` event is raised;
- The event handler method associated with the `<input>` element's `onChange` event is called;

- The event handler method updates the form field's value in state;
- Updating the component's state causes React to re-render the component (i.e. the `render` method is called); and
- The form `<input>` element is rendered with its `value` attribute set to the updated value from the state.

While all of the above steps might *feel* like a lot, in reality, the entire process happens very quickly. You can test this yourself by playing around with the `ContactUs` component. Typing within each of the form fields feels completely natural and you won't notice the difference!

Adding a text area

In a regular HTML form, the value for a `<textarea>` element is defined by its inner content:

```
<textarea>This is the value for the text area element.</textarea>
```

In React, the `<textarea>` element uses a `value` attribute instead of its inner content to define its value. This allows the `<textarea>` element to be handled in the same way as `<input>` elements.

To see this in action, add a `comments` state variable and add a "Comments" field to the form:

```
<div>
  <label htmlFor='comments'>Comments:</label>
  <textarea
    id='comments'
    name='comments'
    onChange={(e) => setComments(e.target.value)}
    value={comments}
  />
</div>
```

To support this new form field, you'll also need to update the `onSubmit` function:

```
// ./src/components/ContactUs/index.js
import { useState } from 'react';

function ContactUs() {
  const [name, setName] = useState('');
  const [email, setEmail] = useState('');
  const [phone, setPhone] = useState('');
  const [comments, setComments] = useState('');

  const onSubmit = (e) => {
    e.preventDefault();
    const contactUsInformation = {
      name,
      email,
      phone,
      comments,
      submittedOn: new Date()
    };

    console.log(contactUsInformation);
    setName('');
    setEmail('');
    setPhone('');
    setComments('');
  };

  return (
    <div>
      <h2>Contact Us</h2>
      <form onSubmit={onSubmit}>
        <div>
          <label htmlFor='name'>Name:</label>
          <input
            id='name'
            type='text'
            onChange={(e) => setName(e.target.value)}
            value={name}
          />
        </div>
        <div>
          <label htmlFor='email'>Email:</label>
          <input
            id='email'
            type='text'
            onChange={(e) => setEmail(e.target.value)}
            value={email}
          />
        </div>
        <div>
          <label htmlFor='phone'>Phone:</label>
          <input
            id='phone'
            type='text'
            onChange={(e) => setPhone(e.target.value)}
            value={phone}
          />
        </div>
        <div>
          <label htmlFor='comments'>Comments:</label>
          <textarea
            id='comments'
            name='comments'
            onChange={(e) => setComments(e.target.value)}
            value={comments}
          />
        </div>
        <button>Submit</button>
      </form>
    </div>
  );
}

export default ContactUs;
```

Adding a select list

To maintain symmetry across React form element types, the `<select>` element also uses a `value` attribute to get and set the element's selected option. To see this in action, add a `<select>` element to the right of the `<input>` element for the "Phone" form field, to give the user a way to specify what type of phone number they're providing:

```
<div>
  <label htmlFor='phone'>Phone:</label>
  <input
    id='phone'
    name='phone'
    type='text'
    onChange={e => setPhone(e.target.value)}
    value={phone}
  />
  <select
    name='phoneType'
    onChange={e => setPhoneType(e.target.value)}
    value={phoneType}
  >
    <option value='' disabled>Select a phone type...</option>
    <option>Home</option>
    <option>Work</option>
    <option>Mobile</option>
  </select>
</div>
```

In the above `<select>` list, the `<option>` elements are statically rendered, but it's also possible to dynamically render them from an array of values. For the array of phone type option values, define a default value for a prop named `phoneTypes`:

```
ContactUs.defaultProps = {
  phoneTypes: [
    'Home',
    'Work',
    'Mobile',
  ],
};
```

Then render the `<select>` list options using the `props.phoneTypes` array:

```
<div>
  <label htmlFor='phone'>Phone:</label>
  <input
    id='phone'
    name='phone'
    type='text'
    onChange={e => setPhone(e.target.value)}
    value={phone}
  />
  <select
    name='phoneType'
    onChange={e => setPhoneType(e.target.value)}
    value={phoneType}
  >
    <option value='' disabled>Select a phone type...</option>
    {props.phoneTypes.map(phoneType =>
      <option key={phoneType}>{phoneType}</option>
    )}
  </select>
</div>
```

Notice that you can leave the first "Select a phone type..." `<option>` element as a static element before rendering the dynamic `<option>` elements.

To complete this new field, update the `onSubmit` function just like you did when adding the "Comments" form field:

```
// ./src/components/ContactUs/index.js
import { useState } from 'react';

function ContactUs(props) {
  const [name, setName] = useState('');
  const [email, setEmail] = useState('');
  const [phone, setPhone] = useState('');
  const [comments, setComments] = useState('');
  const [phoneType, setPhoneType] = useState('');

  const onSubmit = (e) => {
    e.preventDefault();
    const contactUsInformation = {
      name,
      email,
      phone,
      phoneType,
      comments,
      submittedOn: new Date()
    };
    console.log(contactUsInformation);
    setName('');
    setEmail('');
    setPhone('');
    setPhoneType('');
    setComments('');
  };

  return (
    <div>
      <h2>Contact Us</h2>
      <form onSubmit={onSubmit}>
        <div>
          <label htmlFor='name'>Name:</label>
          <input
            id='name'
            type='text'
            onChange={(e) => setName(e.target.value)}
            value={name}
          />
        </div>
        <div>
          <label htmlFor='email'>Email:</label>
          <input
            id='email'
            type='text'
            onChange={(e) => setEmail(e.target.value)}
            value={email}
          />
        </div>
        <div>
          <label htmlFor='phone'>Phone:</label>
          <input
            id='phone'
            type='text'
            onChange={(e) => setPhone(e.target.value)}
            value={phone}
          />
          <select
            name='phoneType'
            onChange={e => setPhoneType(e.target.value)}
            value={phoneType}
          >
            <option value='' disabled>Select a phone
type...</option>
            {props.phoneTypes.map(phoneType =>
              <option key={phoneType}>{phoneType}</option>
            )}
          </select>
        </div>
        <div>
          <label htmlFor='comments'>Comments:</label>
          <textarea
            id='comments'
            name='comments'
            onChange={(e) => setComments(e.target.value)}
            value={comments}
          />
        </div>
        <button>Submit</button>
      </form>
    </div>
  );
}

ContactUs.defaultProps = {
  phoneTypes: [
    'Home',
    'Work',
    'Mobile',
  ],
};

export default ContactUs;
```

Implementing validations

One last feature needs to be added before the simple "Contact Us" form is done: form validation. Without validation, a user can submit the form without providing a single bit of data. To implement form validation, you'll use vanilla JS to validate that the "Name" and "Email" form fields have values before allowing the form to be submitted.

To do that, add a function to your component named `validate`. Use conditional statements to check if the `name` and `email` state variables are empty. If they are empty, add an appropriate validation error message to a `validationErrors` array and return the array from the function:

```
const validate = () => {
  const validationErrors = [];

  if (!name) validationErrors.push('Please provide a Name');

  if (!email) validationErrors.push('Please provide an Email');

  return validationErrors;
}
```

Create a state variable for `validationErrors` initialized to an empty array.
Within the `onSubmit` event handler method, call the `validate` method and check the length of the returned array to see if there are any validation errors. If there are validation errors, then call the `setValidationErrors` function to update the component errors state variable, otherwise process the form submission and reset the `validationErrors` array to an empty array:

```
// Get validation errors.
const errors = validate();

// If we have validation errors...
if (errors.length > 0) {
  // Update the state to display the validation errors.
  setValidationErrors(errors);
} else {
  // Process the form submission...
  // Reset the validation errors
  setValidationErrors([]);
}
```

In the return of the function component, use an inline conditional expression with a logical `&&` operator to conditionally render an unordered list of validation messages if the `validationErrors` array has a `length` greater than `0`:

```
{validationErrors.length > 0 && (
  <div>
    The following errors were found:
    <ul>
      {validationErrors.map(error => <li
key={error}>{error}</li>)}
    </ul>
  </div>
)}
```

Putting all of that together, here's what the updated `ContactUs` function component should look like now:

```
// ./src/components/ContactUs/index.js
import { useState } from 'react';

function ContactUs(props) {
  const [name, setName] = useState('');
  const [email, setEmail] = useState('');
  const [phone, setPhone] = useState('');
  const [comments, setComments] = useState('');
  const [phoneType, setPhoneType] = useState('');
  const [validationErrors, setValidationErrors] = useState([]);

  const validate = () => {
    const validationErrors = [];

    if (!name) validationErrors.push('Please provide a Name');
    if (!email) validationErrors.push('Please provide an Email');

    return validationErrors;
  }

  const onSubmit = (e) => {
    e.preventDefault();
    const errors = validate();

    if (errors.length > 0) return setValidationErrors(errors);

    const contactUsInformation = {
      name,
      email,
      phone,
      phoneType,
      comments,
      submittedOn: new Date()
    };

    console.log(contactUsInformation);
    setName('');
    setEmail('');
    setPhone('');
    setPhoneType('');
    setComments('');
    setValidationErrors([]);
  };

  return (
    <div>
      <h2>Contact Us</h2>
      {validationErrors.length > 0 && (
        <div>
          The following errors were found:
          <ul>
            {validationErrors.map(error => <li
key={error}>{error}</li>)}
          </ul>
        </div>
      )}
      <form onSubmit={onSubmit}>
        <div>
          <label htmlFor='name'>Name:</label>
          <input
            id='name'
            type='text'
            onChange={(e) => setName(e.target.value)}
            value={name}
          />
        </div>
        <div>
          <label htmlFor='email'>Email:</label>
          <input
            id='email'
            type='text'
            onChange={(e) => setEmail(e.target.value)}
            value={email}
          />
        </div>
        <div>
          <label htmlFor='phone'>Phone:</label>
          <input
            id='phone'
            type='text'
            onChange={(e) => setPhone(e.target.value)}
            value={phone}
          />
          <select
            name='phoneType'
            onChange={e => setPhoneType(e.target.value)}
            value={phoneType}
          >
            <option value='' disabled>Select a phone
type...</option>
            {props.phoneTypes.map(phoneType =>
              <option key={phoneType}>{phoneType}</option>
            )}
          </select>
        </div>
        <div>
          <label htmlFor='comments'>Comments:</label>
          <textarea
            id='comments'
            name='comments'
            onChange={(e) => setComments(e.target.value)}
            value={comments}
          />
        </div>
        <button>Submit</button>
      </form>
    </div>
  );
}

ContactUs.defaultProps = {
  phoneTypes: [
    'Home',
    'Work',
    'Mobile',
  ],
};

export default ContactUs;
```

If you run your application again, view the form in the browser, and attempt to submit the form without providing any form field values, you'll receive two validation error messages:

```
The following errors were found:

  * Please provide a Name
  * Please provide an Email
```

Overall, this approach to validating the form is relatively simple. You *could* validate the data as it changes so that the user would receive feedback sooner (i.e., not have to wait to submit the form to see the validation error messages). Sometimes it's helpful to receive feedback in real time, but sometimes it can be annoying to users. Consider each situation and use an approach that feels appropriate for your users.

Using a validation library

You can also use a validation library like Validator.js to add more sophisticated form validations.

First, install the `validator` npm package:

```
npm install validator
```

Then import the email validator into the `./src/components/ContactUs/index.js` module:

```
import isEmail from 'validator/es/lib/isEmail';
```

Now you can use the `isEmail` validator function to check if the provided `email` value is in fact a valid email address:

```
const validate = () => {
  const validationErrors = [];

  if (!name) validationErrors.push('Please provide a Name');

  if (!email) {
    validationErrors.push('Please provide an Email');
  } else if (!isEmail(email)) {
    validationErrors.push('Please provide a valid Email');
  }

  return validationErrors;
}
```

If you run your application again, view the form in the browser, and attempt to submit the form with an invalid email address, you'll receive the following validation error message:

```
The following errors were found:

  * Please provide a valid Email
```

Client-side vs server-side validation

As a reminder, client-side validation like the validations in the `ContactUs` function component, are optional to implement; **server-side validation is not optional**. This is because client side validations can be disabled or manipulated by savvy users. Sometimes the "best" approach is to skip implementing validations on the client-side and rely completely on the server-side validation. Using this approach, you'd simply call the API when the form is submitted and if the request returns a `400 BAD REQUEST` response, you'd display the validation error messages returned from the server.

If you do decide to implement client-side validations, do it with the end goal of improving your application's overall user experience, not as your only means of validating user provided data.

What you learned

In this article, you learned how to create a React function component containing a simple form. You also learned how to create a controlled component which means forms with controlled inputs using component state. You also learned how to use multiple form inputs in React. Lastly, you learned how to implement form validations and the difference between validating your inputs on the client vs. the server.