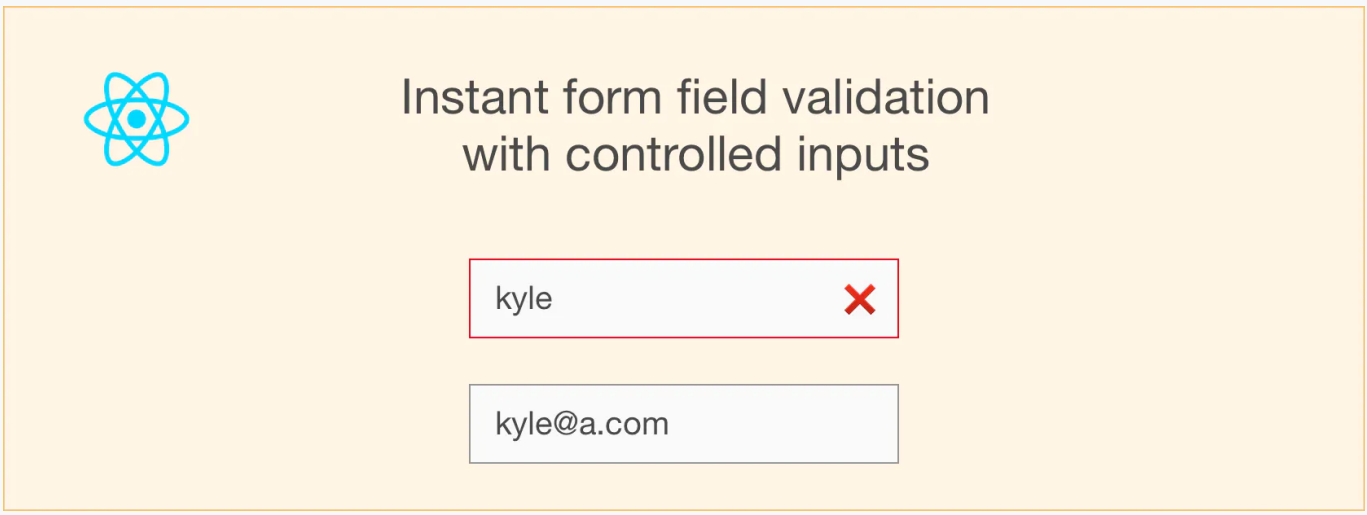
Instant form field validation with React's controlled inputs



If you've followed along, you know about controlled inputs and simple things they enable, like disabling the Submit button when some fields are missing or invalid.

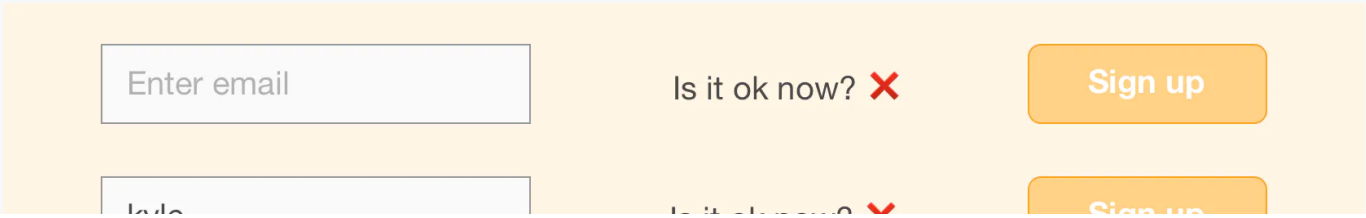
We're not stopping there, of course!

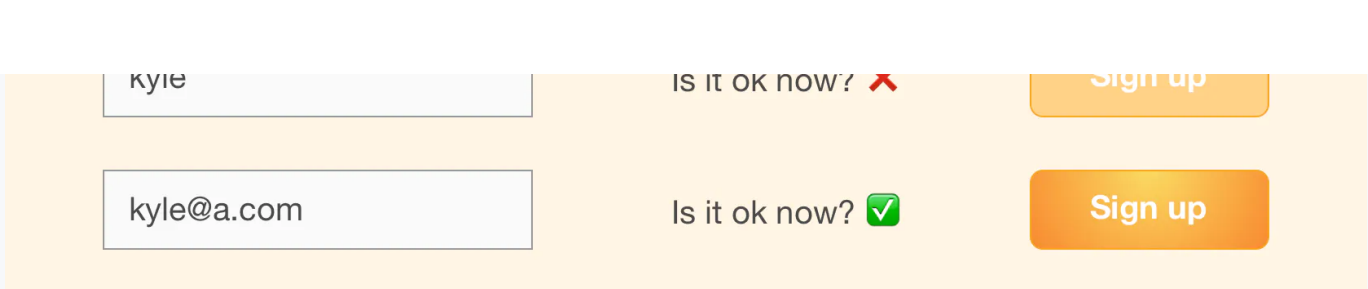
While a disabled button is nice, the reason for that is not immediately apparent to the users. They have no idea what's their mistake here. They don't even know which field is causing that…

And that ain't pretty. We absolutely have to fix that!

As a little refresher from before:

Using controlled inputs implies we are storing all the input values in our state. We can then evaluate a particular condition on every value change, and do something based on it. Previously all we did was disable the button.





We used a simple expression to compute whether the button should be disabled (aka when either of email or password was empty):

```
const { email, password } = this.state;
const isEnabled = email.length > 0 && password.length > 0;

<button disabled={!isEnabled}>Sign up</button>;
```

It got the job done. Now, to mark the bad inputs, we need to ask ourselves a couple of questions.

How are the errors going to be shown?

This is an important question to ask yourself, as different requirements might warrant different error representations.

There are many ways to show input errors. For example, you could:

- display a ❌
- mark red the inputs that contain bad data
- display errors right next to the relevant inputs

- display a list of errors at the top of the form
- any combination of the above, or something else!

Which one should you use? Well, it's all about the experience you want to provide. Pick what you want.

For the purpose of this post, I'm going to do with the simplest one — marking red the bad inputs, without anything else.

How to represent errors?

The way you want to display errors influences how you might represent them.

To indicate whether a particular input is valid, without any additional information as to *why* it is invalid, something like this will suffice:

```
errors: {
  name: false,
  email: true,
}
```

`false` means no errors aka entirely valid; `true` means a field is invalid.

In future, if we decide we need to store the reason something was invalid, we can replace true/false here with a string containing an error message.

But how is this error object created?

Now that we know how we want to display the errors AND know how to represent them, there's something crucial missing.

> How to **actually** get errors?

Or, another way to put it is, how to take existing inputs, validate then and get the error object we need?

We are going to need a **validation function** for that. It will accept the current values of the fields and returns us the `errors` object.

Continuing the sign up form example, recall we had this:

```
const { email, password } = this.state;
const isEnabled = email.length > 0 && password.length > 0;
```

We can, in fact, turn that piece of logic into a validation function that will:

- have `email: true` if email is empty; and
- have `password: true` if password is empty.

```
function validate(email, password) {
  // true means invalid, so our conditions got reversed
  return {
    email: email.length === 0,
    password: password.length === 0,
  };
}
```

The remaining piece

There's one piece of the puzzle remaining.

We have a validation function and know how we want to show errors. We also have a form.

It's time to connect the dots now.

- **Run the validator in `render`.**

  It's no use having the `validate` function if we never call it. We want to validate the inputs every time (yes, every time) the form is re-rendered, which can be because of a new character in the input.

```
const errors = validate(this.state.email, this.state.password);
```

- **Disable the button.**

  This is a simple one. The button should be disabled if there are any errors, or, in other words, if any of `errors` values are `true`.

```
const isEnabled = !Object.keys(errors).some((x) => errors[x]);
```

- **Mark the inputs as erroneous.**

  This can be anything. For our case, adding an `error` class to the bad inputs will be just enough.

```
<input
  className={errors.email ? "error" : ""}
  ...
/>
```

  And we can add a simple CSS rule:

```
.error {
  border: 1px solid red;
}
```

```
import React from "react";
import ReactDOM from "react-dom";

function validate(email, password) {
  // true means invalid, so our conditions got reversed
  return {
    email: email.length === 0,
    password: password.length === 0
```

```
  };
}

export default class SignUpForm extends React.Component {
  constructor() {
    super();
    this.state = {
      email: "",
      password: "",

      everFocusedEmail: false,
```

```
  handlePasswordChange = evt => {
    this.setState({ password: evt.target.value });
  };

  handleSubmit = evt => {
    if (!this.canBeSubmitted()) {
      evt.preventDefault();
      return;
    }
    const { email, password } = this.state;
    alert(`Signed up with email: ${email} password: ${password}`);
  };
```

One more thing

If you look at the demo above, you may notice something odd. The fields are marked red by default, because empty fields are invalid but…

We never even gave a user a chance to type first! Also, the fields are still red while focused for the first time.

This is bad-ish for UX.

We are going to do that by adding the `error` class if the field was in focus at least once but has since been blurred.

This ensures that the first time a user focuses the field, the error won't appear right away, but instead, only when the field is blurred. On subsequent focuses, though, the error would be shown.

This is easily achievable by using the `onBlur` event, and state to keep track of what was blurred.

```
class SignUpForm extends React.Component {
  constructor() {
    super();
    this.state = {
      email: '',
      password: '',
      touched: {
        email: false,
        password: false,
      },
    };
  }

  // ...

  handleBlur = (field) => (evt) => {
    this.setState({
      touched: { ...this.state.touched, [field]: true },
    });
  }

  render()
    const shouldMarkError = (field) => {
      const hasError = errors[field];
      const shouldShow = this.state.touched[field];

      return hasError ? shouldShow : false;
    };

    // ...

    <input
      className={shouldMarkError('email') ? "error" : ""}
      onBlur={this.handleBlur('email')}

      type="text"
      placeholder="Enter email"
      value={this.state.email}
      onChange={this.handleEmailChange}
    />
```

```
  }
}
```

```
import React from "react";
import ReactDOM from "react-dom";

function validate(email, password) {
  // true means invalid, so our conditions got reversed
  return {
    email: email.length === 0,
    password: password.length === 0
  };
}

export default class SignUpForm extends React.Component {
  constructor() {
    super();
    this.state = {
      email: "",
      password: "",

      touched: {
        email: false
```

```
  };

  handlePasswordChange = evt => {
    this.setState({ password: evt.target.value });
  };

  handleBlur = field => evt => {
    this.setState({
      touched: { ...this.state.touched, [field]: true }
    });
  };

  handleSubmit = evt => {
```

Not so hard, right?

Final touches

Note that `shouldMarkError` only affects field presentation. The status of the submit button still depends only on validation `errors`.

A nice final touch might be to force display of errors on all fields, regardless of whether they have been in focus, when the user hovers or clicks a disabled submit button.

Implementing this is left as an exercise for you.

If you are digging this series on handling forms with React, subscribe below to get new posts straight in your inbox.

Think your friends would dig this article, too?

More about forms in React:

- Guide to forms
- Here's what you can do to make migrating your forms to Redux easier in the future
- How to handle validations involving several fields?
- Validating a React form upon submit
- Transitioning from uncontrolled inputs to controlled
- Why not field-level validations?
- How do you make a React form start out with some values prefilled when editing?
- Making dynamic form inputs with React
- Collecting data from a wizard form
- Form recipe: Conditionally disabling the Submit button
- Should you store your form state in Redux?
- Controlled and uncontrolled form inputs in React don't have to be complicated
- The Missing Forms Handbook of React
- My playlist on Egghead