The Magic of React-Based Multi-Step Forms

Author
Nathan Sebhastian

9 Comments
Join the Conversation

Published
Feb 15, 2019

Updated
Feb 15, 2019

One way to deal with long, complex forms is to break them up into multiple steps. You know, answer one set of questions, move on to another, then maybe another, and so on and so forth. We often refer to these as **multi-step forms** (for obvious reasons), but others also take to calling it a "wizard" form.

Multi-step forms can be a great idea! By only showing a few inputs on a screen at a time, the form may feel more digestible and prevent users from feeling overwhelmed by a sea of form fields. Although I haven't looked it up, I'm willing to say no one enjoys completing a ginormous form — that's where multiple steps can come in handy.

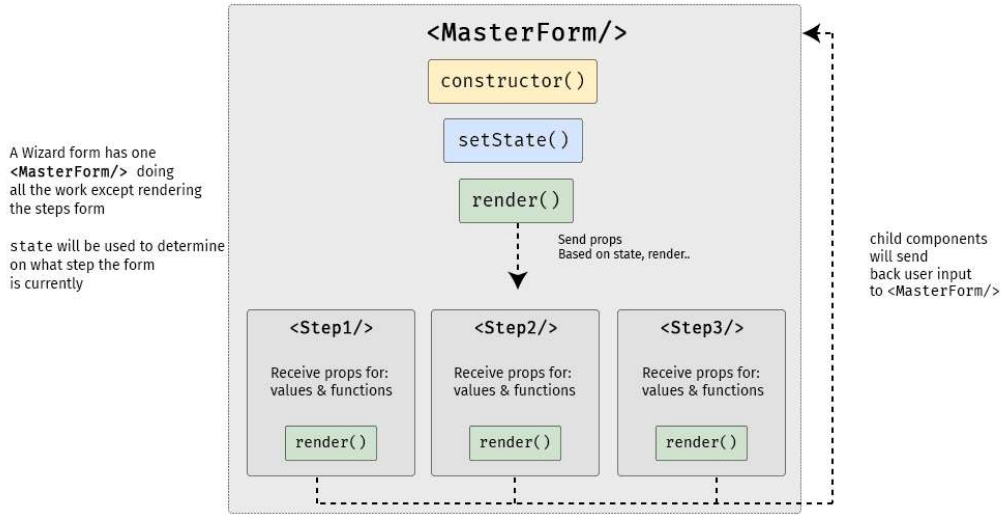
The problem is that multi-step forms — while reducing perceived complexity on the front end — can feel complex and overwhelming to develop. But, I'm here to tell you that it's not only achievable, but relatively straightforward using React as the base. So, that's what we're going to build together today!

Here's the final product:

Embedded Pen Here

Let's build it!

The *easiest* way to create a multi-step form is to create a **container** form element that contains all the steps inside of it as components. Here's a visual showing that container (`<MasterForm/>`), the components inside of it (`<Step1/>` , `<Step2/>` , `<Step3/>`) and the way states and props are passed between them.



`<MasterForm/>` serves as the container while three child components inside of it act as each step of the form.

Although it seems to be more complex than a regular form, a multi-step form still uses the same principles as a React form:

- **State** is used for storing data and user inputs.
- **Component** is used for writing methods and the interface.
- **Props** are used for passing data and function into elements.

Instead of having *one* form component, we will have one *parent* component and three *child* components. In the diagram above, `<MasterForm/>` will send data and functions to the child components via props, and in turn, the child components will trigger a `handleChange()` function to set values in the state of `<MasterForm/>` . It's one big happy family over here!

We'll need a function to move the form from one step to another as well, and we'll get to that a little later.

The step child (get it?) components will receive props from the `<MasterForm/>` parent component for `value` and `onChange` props.

- `<Step1/>` component will render an email address input
- `<Step2/>` will render a username input
- `<Step3/>` will render a password input and a submit button

`<MasterForm/>` will supply both data and function into child components, and child components will pass user inputs back to the parent using its `props`.

Creating the step (child) components

First, we'll create the form's child components. We're keeping things pretty barebones for this example by only using one input per step, but each step could really be as complex as we'd like. Since the child components look almost similar between one another, I'm just gonna show one of them here. But be sure to take a look at the demo (https://codepen.io/nathansebhastian/pen/RvrOYq) for the full code.

JavaScript

```
class Step1 extends React.Component {
  render() {
    if (this.props.currentStep !== 1) { // Prop: The current step
      return null
    }
    // The markup for the Step 1 UI
    return(
      <div className="form-group">
        <label htmlFor="email">Email address</label>
        <input
          className="form-control"
          id="email"
          name="email"
          type="text"
          placeholder="Enter email"
          value={this.props.email} // Prop: The email input data
```

```
            onChange={this.props.handleChange} // Prop: Puts data into state
          />
        </div>
      )
    }
  }
```

Now we can put this child component into the form's `render()` function and pass in the necessary props. Just like in React's form documentation (https://reactjs.org/docs/forms.html) , we can still use `handleChange()` to put the user's submitted data into state with `setState()` . A `handleSubmit()` function will run on form submit.

Next up, the parent component

Let's make the parent component — which we're all aware by now, we're calling `<MasterForm/>` — and initialize its state and methods.

We're using a `currentStep` state that will be initialized with a default value of 1, indicating the first step (`<Step1/>`) of the form. We'll update the state as the form progresses to indicate the current step.

JavaScript

```
class MasterForm extends Component {
  constructor(props) {
    super(props)
    // Set the initial input values
    this.state = {
      currentStep: 1, // Default is Step 1
      email: '',
      username: '',
      password: '',
```

```
    }
    // Bind the submission to handleChange()
    this.handleChange = this.handleChange.bind(this)
  }

  // Use the submitted data to set the state
  handleChange(event) {
    const {name, value} = event.target
    this.setState({
      [name]: value
    })
  }

  // Trigger an alert on form submission
  handleSubmit = (event) => {
    event.preventDefault()
    const { email, username, password } = this.state
    alert(`Your registration detail: \n
      Email: ${email} \n
      Username: ${username} \n
      Password: ${password}`)
  }

  // Render UI will go here...
}
```

OK, that's the baseline functionality we're looking for. Next, we want to create the shell UI for the actual form add call the child components in it, including the required state props that will be passed from `<MasterForm/>` via `handleChange()`.

JavaScript

```
render() {
return (
  <React.Fragment>
  <h1>A Wizard Form!</h1>
  <p>Step {this.state.currentStep} </p>
```

```
    <form onSubmit={this.handleSubmit}>

      // Render the form steps and pass in the required props
      <Step1
        currentStep={this.state.currentStep}
        handleChange={this.handleChange}
        email={this.state.email}
      />
      <Step2
        currentStep={this.state.currentStep}
        handleChange={this.handleChange}
        username={this.state.username}
      />
      <Step3
        currentStep={this.state.currentStep}
        handleChange={this.handleChange}
        password={this.state.password}
      />

    </form>
    </React.Fragment>
  )
  }
```

One step at a time

So far, we've allowed users to fill the form fields, but we've provided no actual way to proceed to the next step or head back to the previous one. That calls for next and previous functions that check if the current step has a previous or next step; and if it does, push the `currentStep` prop up or down accordingly.

JavaScript

```
class MasterForm extends Component {
  constructor(props) {
    super(props)
    // Bind new functions for next and previous
    this._next = this._next.bind(this)
    this._prev = this._prev.bind(this)
  }

  // Test current step with ternary
  // _next and _previous functions will be called on button click
  _next() {
    let currentStep = this.state.currentStep
    // If the current step is 1 or 2, then add one on "next" button click
    currentStep = currentStep >= 2? 3: currentStep + 1
    this.setState({
      currentStep: currentStep
    })
  }

  _prev() {
    let currentStep = this.state.currentStep
    // If the current step is 2 or 3, then subtract one on "previous" butto
    currentStep = currentStep <= 1? 1: currentStep - 1
    this.setState({
      currentStep: currentStep
    })
  }
}
```

We'll use a `get` function that will check whether the current step is 1 or 3. This is because we have three-step form. Of course, we can change these checks as more steps are added to the form. We also want to display the next and previous buttons only if there actually are next and previous steps to navigate to, respectively.

JavaScript

```
// The "next" and "previous" button functions
get previousButton(){
  let currentStep = this.state.currentStep;
  // If the current step is not 1, then render the "previous" button
  if(currentStep !==1){
    return (
      <button
        className="btn btn-secondary"
        type="button" onClick={this._prev}>
      Previous
      </button>
    )
  }
  // ...else return nothing
  return null;
}

get nextButton(){
  let currentStep = this.state.currentStep;
  // If the current step is not 3, then render the "next" button
  if(currentStep <3){
    return (
      <button
        className="btn btn-primary float-right"
        type="button" onClick={this._next}>
      Next
      </button>
    )
  }
  // ...else render nothing
  return null;
}
```

All that's left is to render those buttons:

JavaScript

```
// Render "next" and "previous" buttons
render(){
  return(
    <form onSubmit={this.handleSubmit}>
      {/*
        ... other codes
      */}

      {this.previousButton}
      {this.nextButton}

    </form>
  )
}
```

Congrats, you're a form wizard! 🧙

That was the last step in this multi-step tutorial on multi-step forms. Whoa, how meta! While we didn't go deep into styling, hopefully this gives you a solid overview of how to go about making complex forms less… complex!

Here's that final demo again so you can see all the code in it's full and glorious context:

Embedded Pen Here

HEY!

React was made for this sort of thing considering it makes use of states, property changes, reusable components and such. I know that React may seem like a high barrier to entry for some folks, but I've written a book that makes it a much lower hurdle (https://sebhastian.com/react-distilled/) . I hope you check it out!