# Forms

### Preparation

Inside the code download that came with this course, navigate to forms:

$ cd forms

This folder contains all the code examples for this chapter. To view them in your browser install the dependencies by running npm install (or npm i for short):

$ npm i

Once that’s finished, you can start the app with npm start:

$ npm start

You should expect to see the following in your terminal:

$ npm start

Compiled successfully! The app is running at:

http://localhost:3000/

You should now be able to see the app in your browser if you go to http://localhost:3000.

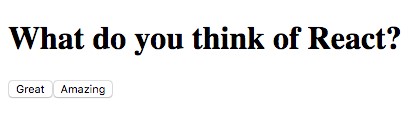
### 1 The Basic Button

At their core, forms are a conversation with the user. Fields are the app’s questions, and the values that the user inputs are the answers.

Let’s ask the user what they think of React.

We could present the user with a text box, but we’ll start even simpler. In this example, we’ll constrain the response to just one of two possible answers. We want to know whether the user thinks React is either “great” or “amazing”, and the simplest way to do that is to give them two buttons to choose from.

Here’s what the first example looks like:



Basic Buttons

To get our app to this stage we create a component with a render() method that returns a div with three child elements: an h1 with the question, and two button elements for the answers. This will look like the following:

**forms/src/01-basic-button.js**

render() {

**return** (

<div>

<h1>What **do** you think **of** React?</h1>

<button

name='button-1'

value='great'

onClick={**this**.onGreatClick}

>

Great

</button>

<button

name='button-2'

value='amazing'

onClick={**this**.onAmazingClick}

>

Amazing

</button>

</div>

);

}

So far this looks a lot like how you’d handle a form with vanilla HTML. The important part to pay attention to is the onClickprop of the button elements. When a button is clicked, if it has a function set as its onClickprop, that function will be called. We’ll use this behavior to know what our user’s answer is.

To know what our user’s answer is, we pass a different function to each button. Specifically, we’ll create function onGreatClick() and provide it to the “Great” button and create function onAmazingClick() and provide it to the “Amazing” button.

Here’s what those functions look like:

**forms/src/01-basic-button.js**

onGreatClick = (evt) => {

console.log('The user clicked button-1: great', evt);

};

onAmazingClick = (evt) => {

console.log('The user clicked button-2: amazing', evt);

};

When the user clicks on the “Amazing” button, the associated onClick function will run (onAmazingClick() in this case). If, instead, the user clicked the “Great” button, onGreatClick() would be run instead.

Notice that in the onClick handler we pass this.onGreatClick and not this.onGreatClick(). What’s the difference?

In the first case (without parens), we’re passing the function onGreatClick, whereas in the second case we’re passing the result of calling the function onGreatClick (which isn’t what we want right now).

This becomes the foundation of our app’s ability to respond to a user’s input. Our app can do different things depending on the user’s response. In this case, we log different messages to the console.

### 2 Back to the Button

In the previous section, we were able to perform different actions (log different messages) depending on the action of the user. However, the way that we set it up, we’d need to create a separate function for each action. Instead, it would be much cleaner if we provided the same event handler to both buttons, and used information from the event itself to determine our response.

To do this, we replace the two event handlers onGreatClick() and onAmazingClick() with a new single event handler, onButtonClick().

**forms/src/02-basic-button.js**

onButtonClick = (evt) => {

**const** btn = evt.target;

console.log(`The user clicked **${**btn.name**}**: **${**btn.value**}**`);

};

Our click handler function receives an event object, evt. **evt** has an attribute target that is a reference to the button that the user clicked. This way we can access the button that the user clicked without creating a function for each button. We can then log out different messages for different user behavior.

Next we update our render() function so that our button elements both use the same event handler, our new onButtonClick() function.

**forms/src/02-basic-button.js**

render() {

**return** (

<div>

<h1>What **do** you think **of** React?</h1>

<button

name='button-1'

value='great'

onClick={**this**.onButtonClick}

>

Great

</button>

<button

name='button-2'

value='amazing'

onClick={**this**.onButtonClick}

>

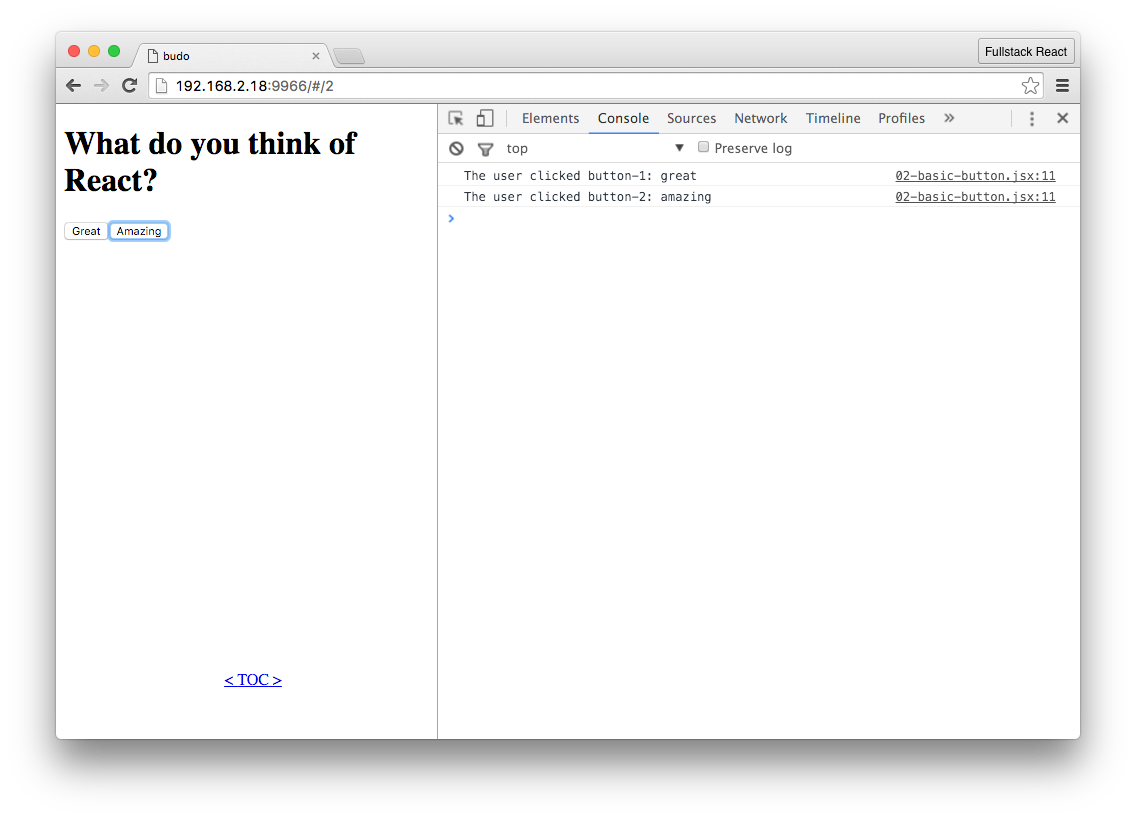
Amazing

</button>

</div>

);

}



One Event Handler for Both Buttons

By taking advantage of the event object and using a shared event handler, we could add 100 new buttons, and we wouldn’t have to make any other changes to our app.

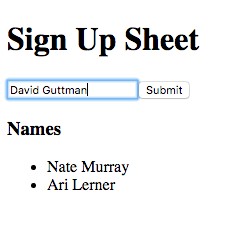
## 3 Text Input

In the previous example, we constrained our user’s response to only one of two possibilities. Now that we know how to take advantage of event objects and handlers in React, we’re going to accept a much wider range of responses and move on to a more typical use of forms: text input.

To showcase text input we’ll create a “sign-up sheet” app. The purpose of this app is to allow a user to record a list of names of people who want to sign up for an event.

The app presents the user a text field where they can input a name and hit “Submit”. When they enter a name, it is added to a list, that list is displayed, and the text box is cleared so they can enter a new name.

Here’s what it will look like:



Sign-Up Adding to a List

### Accessing User Input With refs

We want to be able to read the contents of the text field when the user submits the form. A simple way to do this is to wait until the user submits the form, find the text field in the DOM, and finally grab its value.

To begin we’ll start by creating a form element with two child elements: a text input field and a submit button:

**forms/src/03-basic-input.js**

render() {

**return** (

<div>

<h1>Sign Up Sheet</h1>

<form onSubmit={**this**.onFormSubmit}>

<input

placeholder='Name'

ref='name'

/>

<input type='submit' />

</form>

</div>

);

}

This is very similar to the previous example, but instead of two button elements, we now have a form element with two child elements: a text field and a submit button.

There are two things to notice. First, we’ve added an onSubmit event handler to the form element. Second, we’ve given the text field a refprop of 'name'.

By using an onSubmit event handler on the form element this example will behave a little differently than before. One change is that the handler will be called either by clicking the “Submit” button, or by pressing “enter”/”return” while the form has focus. This is more user-friendly than forcing the user to click the “Submit” button.

However, because our event handler is tied to the form, the event object argument to the handler is less useful than it was in the previous example. Before, we were able to use the targetprop of the event to reference the button and get its value. This time, we’re interested in the text field’s value. One option would be to use the event’s target to reference the form and from there we could find the child input we’re interested in, but there’s a simpler way.

In React, if we want to easily access a DOM element in a component we can use refs (references). Above, we gave our text field a ref property of 'name'.

Later when the onSubmit handler is called, we have the ability to access this.refs.name to get a reference to that text field. Here’s what that looks like in our onFormSubmit() event handler:

**forms/src/03-basic-input.js**

onFormSubmit = (evt) => {

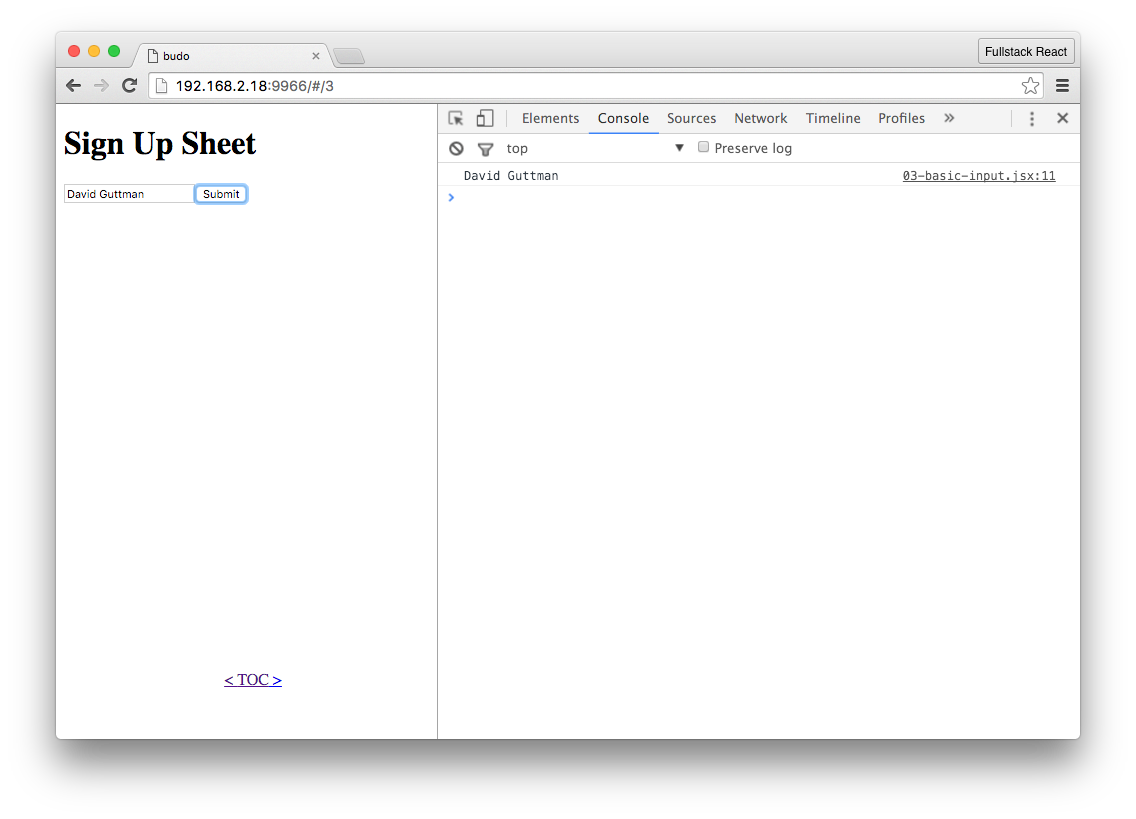
evt.preventDefault();

console.log(**this**.refs.name.value);

};

As you can see, by using this.refs.name we gain a reference to our text field element and we can access its value property.

That value property contains the text that was entered into the field.



Logging The Name

With just the two functions render() and onFormSubmit(), we should now be able to see the value of the text field in our console when we click “Submit”. In the next step we’ll take that value and display it on the page.

### 4 Using User Input

Now that we’ve shown that we can get user submitted names, we can begin to use this information to change the app’s state and UI.

The goal of this example is to show a list with all of the names that the user has entered. React makes this easy. We will have an array in our state to hold the names, and in render() we will use that array to populate a list.

When our app loads, the array will be empty, and each time the user submits a new name, we will add it to the array. To do this, we’ll make a few additions to our component.

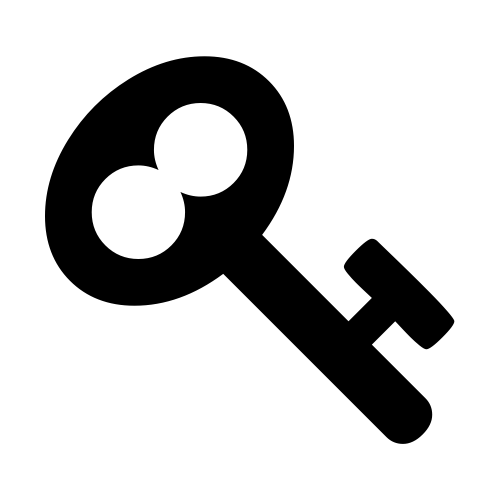
First, we’ll create a names array in our state. In React, when we’re using ES6 component classes we can set the initial value of our state object by defining a property of state. Here’s what that looks like:

**forms/src/04-basic-input.js**

module.exports = **class extends** React.Component {

**static** displayName = "04-basic-input";

state = { names: [] }; *// <-- initial state*



**static**

belongs to the class

Notice

in

this

component

we

have

the

line:

**static**

displayName

=

"04-basic-input"

;

This means that this component class has a static property displayName. When a property is static, that means it is a class property (instead of an instance property). In this case, we’re going to use this displayName when we show the list of examples on the demo listing page.

Next, we’ll modify render() to show this list. Below our form element, we’ll create a new div. This new container div will hold a heading, h3, and our names list, a ul parent with a li child for each name. Here’s our updated render() method:

**forms/src/04-basic-input.js**

render() {

**return** (

<div>

<h1>Sign Up Sheet</h1>

<form onSubmit={**this**.onFormSubmit}>

<input

placeholder='Name'

ref='name'

/>

<input type='submit' />

</form>

<div>

<h3>Names</h3>

<ul>

{ **this**.state.names.map((name, i) => <li key={i}>{name}</li>) }

</ul>

</div>

</div>

);

}

ES2015 gives us a compact way to insert li children. Since this.state.names is an array, we can take advantage of its map() method to return a li child element for each name in the array. Also, by using “arrow” syntax, for our iterator function in map(), the li element is returned without us explicitly using return.

One other thing to note here is that we provide a keyprop to the li element. React will complain when we have children in an array or iterator (like we do here) and they don’t have a keyprop. React wants this information to keep track of the child and make sure that it can be reused between render passes.

We won’t be removing or reordering the list here, so it is sufficient to identify each child by its index. If we wanted to optimize rendering for a more complex use-case, we could assign an immutable id to each name that was not tied to its value or order in the array. This would allow React to reuse the element even if its position or value was changed.

Now that render() is updated, the onFormSubmit() method needs to update the state with the new name. To add a name to the names array in our state we might be tempted to try to do something like this.state.names.push(name). However, React relies on this.setState() to mutate our state object, which will then trigger a new call to render().

The way to do this properly is to:

1. create a new variable that copies our current names
2. add our new name to that array, and finally 3. use that variable in a call to this.setState().

We also want to clear the text field so that it’s ready to accept additional user input. It would not be very user friendly to require the user to delete their input before adding a new name. Since we already have access to the text field via refs, we can set its value to an empty string to clear it.

This is what onFormSubmit() should look like now:

**forms/src/04-basic-input.js**

onFormSubmit = (evt) => {

**const** name = **this**.refs.name.value;

**const** names = [ ...**this**.state.names, name ];

**this**.setState({ names: names });

**this**.refs.name.value = '';

evt.preventDefault();

};

At this point, our sign-up app is functional. Here’s a rundown of the application flow:

* 1. User enters a name and clicks “Submit”.
  2. onFormSubmit is called.
  3. this.refs.name is used to access the value of the text field (a name).
  4. The name is added to our names list in the state.
  5. The text field is cleared so that it is ready for more input.
  6. render is called and displays the updated list of names. So far so good!

In the next sections we’ll improve it further.

### 5 Accessing User Input With state

Converting an uncontrolled input component to a controlled one requires three things. First, we need a place in state to store its value. Second, we provide that location in state as its valueprop. Finally, we add an onChange handler that will update its value in state. The flow for a controlled component looks like this:

1. The user enters/changes the input.
2. The onChange handler is called with the “change” event.
3. Using event.target.value we update the input element’s value in state.
4. render() is called and the input is updated with the new value in state.

Here’s what our render() looks like after converting the input to a controlled component:

**forms/src/05-state-input.js**

render() {

**return** (

<div>

<h1>Sign Up Sheet</h1>

<form onSubmit={**this**.onFormSubmit}>

<input

placeholder='Name'

value={**this**.state.name}

onChange={**this**.onNameChange}

/>

<input type='submit' />

</form>

<div>

<h3>Names</h3>

<ul>

{ **this**.state.names.map((name, i) => <li key={i}>{name}</li>) }

</ul>

</div>

</div>

);

}

The only difference in our input is that we’ve removed the refprop and replaced it with both a value and an onChangeprop.

Now that the input is “controlled”, its value will always be set equal to a property of our state. In this case, that property is name, so the value of the input is this.state.name.

While not strictly necessary, it’s a good habit to provide sane defaults for any properties of state that will be used in our component.

Because we now use state.name for the value of our input, we’ll want to choose what value it will have before the user has had a chance to provide one. In our case, we want the field to be empty, so the default value will be an empty string, '':

**forms/src/05-state-input.js**

state = {

name: '',

names: [],

};

If we had just stopped there, the input would effectively be disabled. No matter what the user types into it, its value wouldn’t change. In fact, if we left it like this, React would give us a warning in our console.

To make our input operational, we’ll need to listen to its onChange events and use those to update the state. To achieve this, we’ve created an event handler for onChange. This handler is responsible for updating our state so that state.name will always be updated with what the user has typed into the field. We’ve created the method onNameChange() for that purpose.

Here’s what onNameChange() looks like now:

**forms/src/05-state-input.js**

onNameChange = (evt) => {

**this**.setState({ name: evt.target.value });

};

onNameChange() is a very simple function. Like we did in a previous section, we use the event passed to the handler to reference the field and get its value. We then update state.name with that value.

Now the controlled component cycle is complete. The user interacts with the field. This triggers an onChange event which calls our onNameChange() handler. Our onNameChange() handler updates the state, and this in turn triggers render() to update the field with the new value.

Our app still needs one more change, however. When the user submits the form, onFormSubmit() is called, and we need that method to add the entered name (state.name) to the names list (state.names). When we last saw onFormSubmit() it did this using this.refs.

Since we’re no longer using refs, we’ve updated it to the following:

**forms/src/05-state-input.js**

onFormSubmit = (evt) => {

**const** names = [ ...**this**.state.names, **this**.state.name ];

**this**.setState({ names: names, name: '' });

evt.preventDefault();

};

Notice that to get the current entered name, we simply access this.state.name because it will be continually updated by our onNameChange() handler. We then append that to our names list, this.state.names and update the state. We also clear this.state.name so that the field is empty and ready for a new name.

While our app didn’t gain any new features in this section, we’ve both paved the way for better functionality (like validation and persistence) while also taking greater advantage of

**6 Multiple Fields**

Our sign-up sheet is looking good, but what would happen if we wanted to add more fields? If our sign-up sheet is like most projects, it’s only a matter of time before we want to add to it. With forms, we’ll often want to add inputs.

If we continue our current approach and create more controlled components, each with a corresponding state property and an onChange handler, our component will become quite verbose. Having a one-to-one-to-one relationship between our inputs, state, and handlers is not ideal.

Let’s explore how we can modify our app to allow for additional inputs in a clean, maintainable way. To illustrate this, let’s add email address to our sign-up sheet.

In the previous section our input field has a dedicated property on the root of our state object. If we were to do that here, we would add another property, email.

To avoid adding a property for each input on state, let’s instead add a fields object to store the values for all of our fields in one place. Here’s our new initial state:

**forms/src/06-state-input-multi.js**

state = {

fields: {

name: '',

email: ''

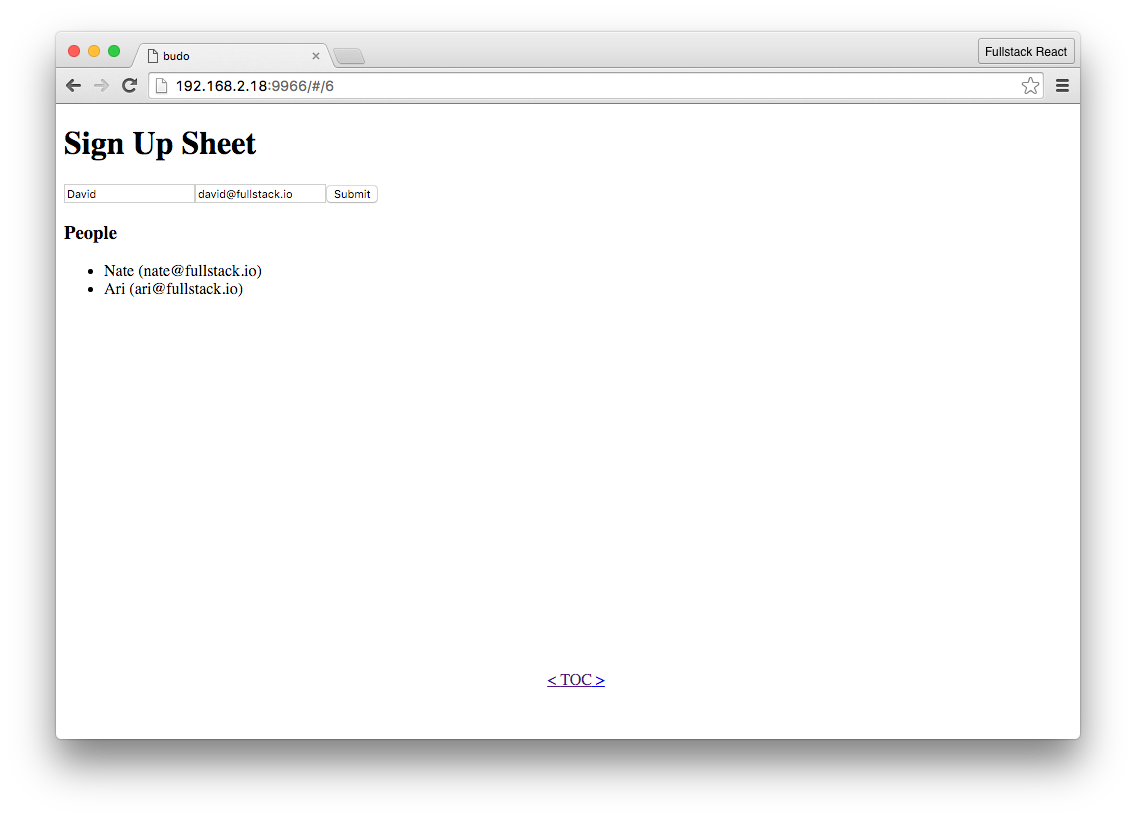
},

people: [],

};

This fields object can store state for as many inputs as we’d like. Here we’ve specified that we want to store fields for name and email.

Now, we will find those values at state.fields.name and state.fields.email instead of state.name and state.email.



Name and Email Fields

Of course, those values will need to be updated by an event handler. We could create an event handler for each field we have in the form, but that would involve a lot of copy/paste and needlessly bloat our component. Also it would make maintainability more difficult, as any change to a form would need to be made in multiple places.

Instead of creating an onChange handler for each input, we can create a single method that accepts change events from all of our inputs. The trick is to write this method in such a way that it updates the correct property in state depending on the input field that triggered the event. To pull this off, the method uses the event argument to determine which input was changed and update our state.fields object accordingly. For example, if we have an input field and we were to give it a nameprop of "email", when that field triggers an event, we would be able to know that it was email field, because event.target.name would be "email".

To see what this looks like in practice, here’s the updated render():

**forms/src/06-state-input-multi.js**

render() {

**return** (

<div>

<h1>Sign Up Sheet</h1>

<form onSubmit={**this**.onFormSubmit}>

<input

placeholder='Name'

name='name'

value={**this**.state.fields.name}

onChange={**this**.onInputChange}

/>

<input

placeholder='Email'

name='email'

value={**this**.state.fields.email}

onChange={**this**.onInputChange}

/>

<input type='submit' />

</form>

<div>

<h3>People</h3>

<ul>

{ **this**.state.people.map(({ name, email }, i) =>

<li key={i}>{name} ({ email })</li>

)}

</ul>

</div>

</div>

);

}

There are a several things to note: first, we’ve added a second input to handle email addresses.

Second, we’ve changed the valueprop of the input fields so that they don’t access attributes on the root of the state object. Instead they access the attributes of state.fields. Looking at the code above, the input for name now has its value set to this.state.fields.name.

Third, both input fields have their onChangeprop set to the same event handler, onInputChange(). We’ll see below how we modified onNameChange() to be a more general event handler that can accept events from any field, not just “name”.

Fourth, our input fields now have a nameprop. This is related to the last point. To allow our general event handler, onInputChange(), to be able to tell where the change event came from and where we should store it in our state (e.g. if the change comes from the “email” input the new value should be stored at state.fields.email), we provide that nameprop so that it can be pulled off of the event via its target attribute.

Finally, we modify how our people list is rendered. Because it’s no longer just a list of names, we modify our li element to display both the previous name attribute as well as the new email data we plan to have.

To make sure that all the data winds up in the right place, we’ll need to make sure that our event handlers are properly modified. Here’s what the onInputChange() event handler (that gets called when any field’s input changes) should look like:

**forms/src/06-state-input-multi.js**

onInputChange = (evt) => {

**const** fields = **this**.state.fields;

fields[evt.target.name] = evt.target.value;

**this**.setState({ fields });

};

At its core this is similar to what we did before in onNameChange() in the last section. The two key differences are that:

* 1. we are updating a value nested in the state (e.g. updating state.fields.email instead of state.email), and
  2. we’re using evt.target.name to inform which attribute of state.fields needs to be updated.

To properly update our state, we first grab a local reference to state.fields. Then, we use information from the event (evt.target.name and evt.target.value) to update the local reference. Lastly, we setState() with the modified local reference.

To get concrete, let’s go through what would happen if the user enters “someone@somewhere.com” into the “email” field.

First, onInputChange() would be called with the evt object as an argument. evt.target.name would be "email" (because "email" is set as its nameprop in render()) and evt.target.value would be "someone@somewhere.com" (because that’s what they entered into the field).

Next, onInputChange() would grab a local reference to state.fields. If this is the first time there was input, state.fields and our local reference will be the default fields in state, { name: '', email: '' }. Then, the local reference would be modified so that it becomes { name: '', email:

"someone@somewhere.com" }.

And finally, setState() is called with that change.

At this point, this.state.fields will always be in sync with any text in the input fields. However, onFormSubmit() will need to be changed to get that information into the list of people who have signed up. Here’s what the updated onFormSubmit() looks like:

**forms/src/06-state-input-multi.js**

onFormSubmit = (evt) => {

**const** people = [

...**this**.state.people,

**this**.state.fields,

];

**this**.setState({

people,

fields: {

name: '',

email: ''

}

});

evt.preventDefault();

};

In onFormSubmit() we first obtain a local reference to the list of people who have signed up, this.state.people. Then, we add our this.state.fields object (an object representing the name and email currently entered into the fields) onto the people list. Finally, we use this.setState() to simultaneously update our list with the new information and clear all the fields by returning state.fields to the empty defaults, { name: '', email: '' }.

The great thing about this is that we can easily add as many input fields as we want with very minimal changes. In fact, only the render() method would need to change. For each new field, all we would have to do is add another input field and change how the list is rendered to display the new field.

For example, if we wanted to add a field for phone number, we would add a new input with appropriate name and valueprops: name would be phone and value would be this.state.fields.phone. onChange, like the others, would be our existing onInputChange() handler.

After doing that our state will automatically keep track of the phone field and will add it to the state.people array and we could change how the view (e.g. the li) displays the information.

At this point we have a functional app that’s well situated to be extended and modified as requirements evolve. However, it is missing one crucial aspect that almost all forms need: validation.

### 7 On Validation

Validation is so central to building forms that it’s rare to have a form without it. Validation can be both on the level of the individual field and on the form as a whole.

When you validate on an individual field, you’re making sure that the user has entered data that conforms to your application’s expectations and constraints as it relates to that piece of data.

For example, if we want a user to enter an email address, we expect their input to look like a valid email address. If the input does not look like an email address, they might have made a mistake and we’re likely to run into trouble down the line (e.g. they won’t be able to activate their account). Other common examples are making sure that a zip code has exactly five (or nine) numerical characters and enforcing a password length of at least some minimum length.

Validation on the form as a whole is slightly different. Here is where you’ll make sure that all required fields have been entered. This is also a good place to check for internal consistency. For example you might have an order form where specific options are required for specific products.

Additionally, there are trade-offs for “how” and “when” we validate. On some fields we might want to give validation feedback in real-time. For example, we might want to show password strength (by looking at length and characters used) while the user is typing. However, if we want to validate the availability of a username, we might want to wait until the user has finished typing before we make a request to the server/database to find out.

We also have options for how we display validation errors. We might style the field differently (e.g. a red outline), show text near the field (e.g. “Please enter a valid email.”), and/or disable the form’s submit button to prevent the user from progressing with invalid information.

As for our app, we can begin with validation of the form as a whole and

1. make sure that we have both a name and email and

2. make sure that the email is a valid address.

### Adding Validation to Our App

To add validation to our sign-up app we’ve made some changes. At a high level these changes are

1. add a place in state to store validation errors if they exist,
2. change our render() method so it will show validation error messages (if they exist) with red text next to each field,
3. add a new validate() method that takes our fields object as an argument and returns a fieldErrors object, and
4. onFormSubmit() will call the new validate() method to get the fieldErrors object, and if there are errors it will add them to the state (so that they can be shown in render()) and return early without adding the “person” to the list, state.people.

First, we’ve changed our initial state:

**forms/src/07-basic-validation.js**

state = {

fields: {

name: '',

email: '',

},

fieldErrors: {},

people: [],

};

The only change here is that we’ve created a default value for the fieldErrors property. This is where we’ll store errors for each of the field if they exist.

Next, here’s what the updated render() method looks like:

**forms/src/07-basic-validation.js**

render() {

**return** (

<div>

<h1>Sign Up Sheet</h1>

<form onSubmit={**this**.onFormSubmit}>

<input

placeholder='Name'

name='name'

value={**this**.state.fields.name}

onChange={**this**.onInputChange}

/>

<span style={{ color: 'red' }}>{ **this**.state.fieldErrors.name }</span>

<br />

<input

placeholder='Email'

name='email'

value={**this**.state.fields.email}

onChange={**this**.onInputChange}

/>

<span style={{ color: 'red' }}>{ **this**.state.fieldErrors.email }</span>

<br />

<input type='submit' />

</form>

<div>

<h3>People</h3>

<ul>

{ **this**.state.people.map(({ name, email }, i) =>

<li key={i}>{name} ({ email })</li>

) }

</ul>

</div>

</div>

);

}

The only differences here are two new span elements, one for each field. Each span will look in the appropriate place in state.fieldErrors for an error message. If one is found it will be displayed in red next to the field. Next up, we’ll see how those error messages can get into the state.

It is after the user submits the form that we will check the validity of their input. So the appropriate place to begin validation is in the onFormSubmit() method.

However, we’ll want to create a standalone function for that method to call. We’ve created the pure function, validate() method for this:

**forms/src/07-basic-validation.js**

validate = (person) => {

**const** errors = {};

**if** (!person.name) errors.name = 'Name Required';

**if** (!person.email) errors.email = 'Email Required';

**if** (person.email && !isEmail(person.email)) errors.email = 'Invalid Email';

**return** errors;

};

Our validate() method is pretty simple and has two goals. First, we want to make sure that both name and email are present. By checking their truthiness we can know that they are defined and not empty strings. Second, we want to know that the provided email address looks valid. This is actually a bit of a thorny issue, so we rely on [validator](http://npm.im/validator)[[1]](#footnote-1) to let us know. If any of these conditions are not met, we add a corresponding key to our errors object and set an error message as the value.

Afterwards, we’ve updated our onFormSubmit() to use this new validate() method and act on the returned error object:

**forms/src/07-basic-validation.js**

onFormSubmit = (evt) => {

**const** people = [ ...**this**.state.people ];

**const** person = **this**.state.fields;

**const** fieldErrors = **this**.validate(person);

**this**.setState({ fieldErrors });

evt.preventDefault();

**if** (Object.keys(fieldErrors).length) **return**;

**this**.setState({

people: people.concat(person),

fields: {

name: '',

email: '',

},

});

};

To use the validate() method, we get the current values of our fields from this.state.fields and provide it as the argument. validate() will either return an empty object if there are no issues, or if there are issues, it will return an object with keys corresponding to each field name and values corresponding to each error message. In either case, we want to update our state.fieldErrors object so that render() can display or hide the messages as necessary.

If the validation errors object has any keys (Object.keys(fieldErrors).length > 0) we know there are issues. If there are no validation issues, the logic is the same as in previous sections – we add the new information and clear the fields. However, if there are any errors, we return early.

This prevents the new information from being added to the list.



Email Required



Email Invalid

At this point we’ve covered the fundamentals of creating a form with validation in React. In the next section we’ll take things a bit further and show how we can validate in real-time at the field level and we’ll create a Field component to improve the maintainability when an app has multiple fields with different validation requirements.

### 8 Creating the Field Component

In the last section we added validation to our form. However, our form component is responsible for running the validations on the form as a whole as well as the individual validation rules for each field.

It would be ideal if each field was responsible for identifying validation errors on its own input, and the parent form was only responsible for identifying issues at the form-level. This comes with several advantages:

1. an email field created in this way could check the format of its input while the user types in real-time.
2. the field could incorporate its validation error message, freeing the parent form from having to keep track of it.

To do this we’re first going to create a new separate Field component, and we will use it instead of input elements in the form. This will let us combine a normal input with both validation logic and error messaging.

Before we get into the creation of this new component, it will be useful to think of it at a high level in terms of inputs and outputs. In other words, “what information do we need to provide this component?”, and “what kinds of things would we expect in return?”

These inputs are going to become this component’s props and the output will be used by any event handlers we pass into it.

Because our Field component will contain a child input, we’ll need to provide the same baseline information so that it can be passed on. For example, if we want a Field component rendered with a specific placeholderprop on its child input, we’ll have to provide it as a prop when we create the Field component in our form’s render() method.

Two other props we’ll want to provide are name, and value. name will allow us to share an event handler between components like we’ve done before, and value allows the parent form to prepopulate Field as well as keep it updated.

Additionally, this new Field component is responsible for its own validation. Therefore we’ll need to provide it rules specific to data it contains. For example, if this is the “email” Field, we’ll provide it a function as its validateprop. Internally it will run this function to determine if its input is a valid email address.

Lastly, we’ll provide an event handler for onChange events. The function we provide as the onChange prop will be called every time the input in the Field changes, and it will be called with an event argument that we get to define. This event argument should have three properties that we’re interested in: (1) the name of the Field, (2) the current value of the input, and (3) the current validation error (if present).

To quickly review, for the new Field component to do its job it will need the following:

* placeholder: This will be passed straight through to the input child element. Similar to a label, this tells the user what data to the Field expects.
* name: We want this for the same reason we provide name to input elements: we’ll use this in the event handler decide where to store input data and validation errors.
* value: This is how our parent form can initialize the Field with a value, or it can use this to update the Field with a new value. This is similar to how the valueprop is used on an input. • validate: A function that returns validation errors (if any) when run.
* onChange: An event handler to be run when the Field changes. This function will accept an event object as an argument.

Following this, we’re able to set up propTypes on our new Field component:

**forms/src/08-field-component-field.js**

**static** propTypes = {

placeholder: PropTypes.string,

name: PropTypes.string.isRequired,

value: PropTypes.string,

validate: PropTypes.func,

onChange: PropTypes.func.isRequired,

};

Next, we can think about the state that Field will need to keep track of. There are only two pieces of data that Field will need, the current value and error. Like in previous sections where our form component needed that data for its render() method, so too does our Field component. Here’s how we’ll set up our initial state:

**forms/src/08-field-component-field.js**

state = {

value: **this**.props.value,

error: **false**,

};

One key difference is that our Field has a parent, and sometimes this parent will want to update the value prop of our Field.

To allow this, we’ll need to create a new lifecycle method, componentWillReceiveProps() to accept the new value and update the state. Here’s what that looks like:

**forms/src/08-field-component-field.js**

componentWillReceiveProps(update) {

**this**.setState({ value: update.value });

}

The render() method of Field should be pretty simple. It’s just the input and the corresponding span that will hold the error message:

**forms/src/08-field-component-field.js**

render() {

**return** (

<div>

<input

placeholder={**this**.props.placeholder}

value={**this**.state.value}

onChange={**this**.onChange}

/>

<span style={{ color: 'red' }}>{ **this**.state.error }</span>

</div>

);

}

For the input element, the placeholder will be passed in from the parent and is available from this.props.placeholder. As mentioned above, the value of the input and the error message in the span will both be stored in the state. value comes from this.state.value and the error message is at this.state.error.

And lastly, we’ll set an onChange event handler that will be responsible for accepting user input, validating, updating state, and calling the parent’s event handler as well. The method that will take care of that is this.onChange:

onChange (evt) {

**const** name = **this**.props.name;

**const** value = evt.target.value;

**const** error = **this**.props.validate ? **this**.props.validate(value) : **false**;

**this**.setState({value, error});

**this**.props.onChange({name, value, error});

}

this.onChange is a pretty efficient function. It handles four different responsibilities in as many lines. As in previous sections, the event object gives us the current text in the input via its target.value property. Once we have that, we see if it passes validation. If Field was given a function for its validateprop, we use it here. If one was not given, we don’t have to validate the input and error sets to false. Once we have both the value and error we can update our state so that they both appear in render(). However, it’s not just the Field component that needs to be updated with this information.

When Field is used by a parent component, it passes in its own event handler in as the onChange prop. We call this function so that we can pass information up the parent. Here in this.onChange(), it is available as this.props.onChange(), and we call it with three pieces of information: the name, value, and error of the Field.

This might be a little confusing since “onChange” appears in multiple places. You can think of it as carrying information in a chain of event handlers. The form contains the Field which contains an input. Events occur on the input and the information passes first to the Field and finally to the form.

At this point our Field component is ready to go, and can be used in place of the input and error message combos in our app.

### Using our new Field Component

Now that we’re ready to use our brand new Field component, we can make some changes to our app. The most obvious change is that Field will take the place of both the input and error message span elements in our render() method. This is great because Field can take care of validation at the field level. But what about at the form level?

If you remember, we can employ two different levels of validation, one at the field level, and one at the form level. Our new Field component will let us validate the format of each field in real-time. What they won’t do, however, is validate the entire form to make sure we have all the data we need. For that, we also want form-level validation.

Another nice feature we’ll add here is disabling/enabling the form submit button in real-time as the form passes/fails validation. This is a nice bit of feedback that can improve a form’s UX and make it feel more responsive.

Here’s how our update render() looks:

**forms/src/08-field-component-form.js**

render() {

**return** (

<div>

<h1>Sign Up Sheet</h1>

<form onSubmit={**this**.onFormSubmit}>

<Field

placeholder='Name'

name='name'

value={**this**.state.fields.name}

onChange={**this**.onInputChange}

validate={(val) => (val ? **false** : 'Name Required')}

/>

<br />

<Field

placeholder='Email'

name='email'

value={**this**.state.fields.email}

onChange={**this**.onInputChange}

validate={(val) => (isEmail(val) ? **false** : 'Invalid Email')}

/>

<br />

<input type='submit' disabled={**this**.validate()} />

</form>

<div>

<h3>People</h3>

<ul>

{ **this**.state.people.map(({ name, email }, i) =>

<li key={i}>{name} ({email})</li>

) }

</ul>

</div>

</div>

);

}

You can see that Field is a drop-in replacement for input. All the props are the same as they were on input, except we have one additional prop this time: validate.

Above in the Field component’s onChange() method, we make a call to the this.props.validate() function. What we provide as the validateprop to Field, will be that function. Its goal is to take user provided input as its argument and give a return value that corresponds to the validity of that input. If the input is not valid, validate should return an error message. Otherwise, it should return false.

For the “name” Field the validateprop is pretty simple. We’re just checking for a truthy value. As long as there are characters in the box, validation will pass, otherwise we return the 'Name Required' error message.

For the “email” Field, we’re going to use the isEmail() function that we imported from the validator module. If that function returns true, we know it’s a valid-looking email and validation passes. If not, we return the 'Invalid Email' message.

Notice that we left their onChangeprop alone, it is still set to this.onInputChange. However, since Field uses the function differently than input, we must update onInputChange().

Before we move on, notice the only other change that we’ve made to render(): we conditionally disable the submit button. To do this, we set the value of the disabledprop to the return value of this.validate(). Because this.validate() will have a truthy return value if there are validation errors, the button will be disabled if the form is not valid. We’ll show what the this.validate() function looks like in a bit.



Disabled Submit Button

As mentioned, both Field components have their onChangeprops set to this.onInputChange.

We’ve had to make some changes to match the difference between input and our Field. Here’s the updated version:

**forms/src/08-field-component-form.js**

onInputChange = ({ name, value, error }) => {

**const** fields = **this**.state.fields;

**const** fieldErrors = **this**.state.fieldErrors;

fields[name] = value;

fieldErrors[name] = error;

**this**.setState({ fields, fieldErrors });

};

Previously, the job of onInputChange() was to update this.state.fields with the current user input values. In other words, when a text field was edited, onInputChange() would be called with an event object. That event object had a target property that referenced the input element. Using that reference, we could get the name and value of the input, and with those, we would update state.fields.

This time around onInputChange() has the same responsibility, but it is our Field component that calls this function, not input. In the previous section, we show the onChange() method of Field, and that’s where this.props.onChange() is called. When it is called, it’s called like this: ‘ this.props.onChange({name, value, error})‘.

This means that instead of using evt.target.name or evt.target.value as we did before, we get name and value directly from the argument object. In addition, we also get the validation error for each field. This is necessary – for our form component to prevent submission, it will need to know about field-level validation errors.

Once we have the name, value, and error, we can update two objects in our state, the state.fields object we used before, and a new object, state.fieldErrors. Soon, we will show how state.fieldErrors will be used to either prevent or allow the form submit depending on the presence or absence of field-level validation errors.

With both render() and onInputChange() updated, we again have a nice feedback loop set up for our Field components:

* First, the user types into the Field.
* Then, the event handler of the Field is called, onInputChange().
* Next, onInputChange() updates the state.
* After, the form is rendered again, and the Field passed an updated valueprop.
* Then, componentWillReceiveProps() is called in Field with the new value, and its state is updated.
* Finally, Field.render() is called again, and the text field shows the appropriate input and (if applicable) validation error.

At this point, our form’s state and appearance are in sync. Next, we need to change how we handle the submit event. Here’s the updated event handler for the form, onFormSubmit():

**forms/src/08-field-component-form.js**

onFormSubmit = (evt) => {

**const** people = **this**.state.people;

**const** person = **this**.state.fields;

evt.preventDefault();

**if** (**this**.validate()) **return**;

**this**.setState({

people: people.concat(person),

fields: {

name: '',

email: '',

},

});

};

The objective of onFormSubmit() hasn’t changed. It is still responsible for either adding a person to the list, or preventing that behavior if there are validation issues. To check for validation errors, we call this.validate(), and if there are any, we return early before adding the new person to the list.

Here’s what the current version of validate() looks like:

validate () {

**const** person = **this**.state.fields;

**const** fieldErrors = **this**.state.fieldErrors;

**const** errMessages = Object.keys(fieldErrors).filter((k) => fieldErrors[k])

**if** (!person.name) **return true**;

**if** (!person.email) **return true**;

**if** (errMessages.length) **return true**;

**return false**

},

Put simply, validate() is checking to make sure the data is valid at the form level. For the form to pass validation at this level it must satisfy two requirements: (1) neither field may be empty and (2) there must not be any field-level validation errors.

To satisfy the first requirement, we access this.state.fields and ensure that both state.fields.name and state.fields.email are truthy. These are kept up to date by onInputChange(), so it will always match what is in the text fields. If either name or email are missing, we return true, signaling that there is a validation error.

For the second requirement, we look at this.state.fieldErrors. onInputChange() will set any field-level validation error messages on this object. We use Object.keys and Array.filter to get an array of all present error messages. If there are any field-level validation issues, there will be corresponding error messages in the array, and its length will be non-zero and truthy. If that’s the case, we also return true to signal the existence of a validation error.

validate() is a simple method that can be called at any time to check if the data is valid at the formlevel. We use it both in onFormSubmit() to prevent adding invalid data to the list and in render() to disable the submit button, providing nice feedback in the UI.

And that’s it. We’re now using our custom Field component to do field-level validation on the fly, and we also use form-level validation to toggle the submit button in real-time.

## Remote Data Information Only

Our form app is coming along. A user can sign up with their name and email, and we validate their information before accepting the input. But now we’re going to kick it up a notch. We’re going to explore how to allow a user to select from hierarchical, asynchronous options.

The most common example is to allow the user to select a car by year, make, and model. First the user selects a year, then the manufacturer, then the model. After choosing an option in one select, the next one becomes available. There are two interesting facets to building a component like this.

First, not all combinations make sense. There’s no reason to allow your user to choose a 1965 Tesla Model T. Each option list (beyond the first) depends on a previously selected value.

Second, we don’t want to send the entire database of valid choices to the browser. Instead, the browser only knows the top level of choices (e.g. years in a specific range). When the user makes a selection, we provide the selected value to the server and ask for next level (e.g. manufacturers available for a given year). Because the next level of options come from the server, this is an asynchronous activity.

Our app won’t be interested in the user’s car, but we will want to know what they’re signing up for. Let’s make this an app for users to learn more JavaScript by choosing a [NodeSchool](http://nodeschool.io/)[[2]](#footnote-2) workshop to attend.

A NodeSchool workshop can be either “core” or “elective”. We can think of these as “departments” of NodeSchool. Therefore, depending on which department the user is interested in, we can allow them to choose a corresponding workshop. This is similar to the above example where a user chooses a car’s year before its manufacturer.

If a user chooses the core department, we would enable them to choose from a list of core workshops like learnyounode and stream-adventure. If instead, they choose the elective department, we would allow them to pick workshops like Functional JavaScript or Shader School. Similar to the car example, the course list is provided asynchronously and depends on which department was selected.

The simplest way to achieve this is with two select elements, one for choosing the department and the other for choosing the course. However, we will hide the second select until: (1) the user has selected a department, and (2) we’ve received the appropriate course list from the server.

Instead of building this functionality directly into our form. We’ll create a custom component to handle both the hierarchical and asynchronous nature of these fields. By using a custom component, our form will barely have to change. Any logic specific to the workshop selection will be hidden in the component.

### 9 Building the Custom Component

The purpose of this component is to allow the user to select a NodeSchool course. From now on we’ll refer to it as our CourseSelect component.

However, before starting on our new CourseSelect component, we should think about how we want it to communicate with its form parent. This will determine the component’s props.

The most obvious prop is onChange(). The purpose of this component is to help the user make a department/course selection and to make that data available to the form. Additionally, we’ll want to be sure that onChange() is called with the same arguments we’re expecting from the other field components. That way we don’t have to create any special handling for this component.

We also want the form to be able to set this component’s state if need be. This is particularly useful when we want to clear the selections after the user has submitted their info. For this we’ll accept two props. One for department and one for course.

And that’s all we need. This component will accept three props. Here’s how they’ll look in our new CourseSelect component:

**forms/src/09-course-select.js**

**static** propTypes = {

department: PropTypes.string,

course: PropTypes.string,

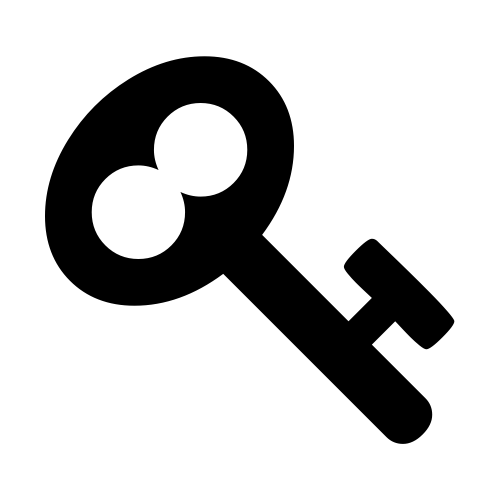
onChange: PropTypes.func.isRequired,

};

Next, we can think about the state that CourseSelect will need to keep track of. The two most obvious pieces of state are department and course. Those will change when the user makes selections, and when the form parent clears them on a submit.

CourseSelect will also need to keep track of available courses for a given department. When a user selects a department, we’ll asynchronously fetch the corresponding course list. Once we have that list we’ll want to store it in our state as courses.

Lastly, when dealing with asynchronous fetching, it’s nice to inform the user that data is loading behind the scenes. We will also keep track of whether or not data is “loading” in our state as \_loading.

The underscore prefix of \_loading is just a convention to highlight that it is purely presentational. Presentational state is only used for U I effects. In this case it will be used to hide/show the loading indicator image.

Here’s what our initial state looks like:

**forms/src/09-course-select.js**

state = {

department: **null**,

course: **null**,

courses: [],

\_loading: **false**,

};

As mentioned above, this component’s form parent will want to update the department and course props. Our componentWillReceiveProps() method will use the update to appropriately modify the state:

**forms/src/09-course-select.js**

componentWillReceiveProps(update) {

**this**.setState({

department: update.department,

course: update.course,

});

}

Now that we have a good idea of what our data looks like, we can get into how the component is rendered. This component is a little more complicated than our previous examples, so we take advantage of composition to keep things tidy.

You will notice that our render() method is mainly composed of two functions, renderDepartmentSelect() and renderCourseSelect():

**forms/src/09-course-select.js**

render() {

**return** (

<div>

{ **this**.renderDepartmentSelect() }

<br />

{ **this**.renderCourseSelect() }

</div>

);

}

Aside from those two functions, render() doesn’t have much. But this nicely illustrates the two “halves” of our component: the “department” half and the “course” half.

Let’s first take a look at the “department” half. Starting with renderDepartmentSelect():

**forms/src/09-course-select.js**

{ **this**.renderDepartmentSelect() }

This method returns a select element that displays one of three options. The currently displayed option depends on the valueprop of the select. The option whose value matches the select will be shown. The options are:

* “Which department?” (value: empty string)
* “NodeSchool: Core” (value: "core")
* “NodeSchool: Electives” (value: "electives")

The value of select is this.state.department || ''. In other words, if this.state.department is falsy (it is by default), the value will be an empty string and will match “Which department?”. Otherwise, if this.state.department is either "core" or "electives", it will display one of the other options.

Because this.onSelectDepartment is set as the onChangeprop of the select, when the user changes the option, onSelectDepartment() is called with the change event. Here’s what that looks like:

**forms/src/09-course-select.js**

onSelectDepartment = (evt) => {

**const** department = evt.target.value;

**const** course = **null**;

**this**.setState({ department, course });

**this**.props.onChange({ name: 'department', value: department });

**this**.props.onChange({ name: 'course', value: course });

**if** (department) **this**.fetch(department);

};

When the department is changed, we want three things to happen. First, we want to update state to match the selected department option. Second, we want to propagate the change via the onChange handler provided in the props of CourseSelect. Third, we want to fetch the available courses for the department.

When we update the state, we update it to the value of the event’s target, the select. The value of the select is the value of the chosen option, either '', "core", or "electives". After the state is set with a new value, render() and renderDepartmentSelect() are run and a new option is displayed.

Notice that we also reset the course. Each course is only valid for its department. If the department changes, it will no longer be a valid choice. Therefore, we set it back to its initial value, null.

After updating state, we propagate the change to the component’s change handler,this.props.onChange. Because we use the arguments as we have previously, this component can be used just like Field and can be given the same handler function. The only trick is that we need to call it twice, once for each input.

Finally, if a department was selected, we fetch the course list for it. Here’s the method it calls, fetch():

**forms/src/09-course-select.js**

fetch = (department) => {

**this**.setState({ \_loading: **true**, courses: [] });

apiClient(department).then((courses) => {

**this**.setState({ \_loading: **false**, courses: courses });

});

};

The responsibility of this method is to take a department string, use it to asynchronously get the corresponding course list, courses, and update the state with it. However, we also want to be sure to affect the state for a better user experience.

We do this by updating the state before the apiClient call. We know that we’ll be waiting for the response with the new course list, and in that time we should show the user a loading indicator. To do that, we need our state to reflect our fetch status. Right before the apiClient call, we set the state of \_loading to true. Once the operation completes, we set \_loading back to false and update our course list.

Previously, we mentioned that this component had two “halves” illustrated by our render() method:

**forms/src/09-course-select.js**

render() {

**return** (

<div>

{ **this**.renderDepartmentSelect() }

<br />

{ **this**.renderCourseSelect() }

</div>

);

}

We’ve already covered the “department” half. Let’s now take a look at the “course” half starting with renderCourseSelect():

**forms/src/09-course-select.js**

{ **this**.renderCourseSelect() }

The first thing that you’ll notice is that renderCourseSelect() returns a different root element depending on particular conditions.

If state.\_loading is true, renderCourseSelect() only returns a single img: a loading indicator. Alternatively, if we’re not loading, but a department has not been selected (and therefore state.department is falsy), an empty span is returned – effectively hiding this half of the component.

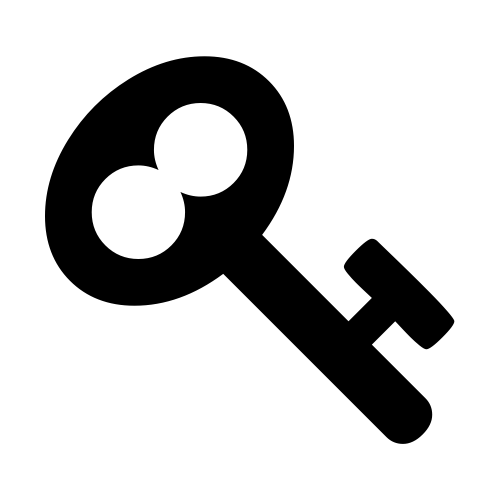
However, if we’re not loading, and the user has selected a department, renderCourseSelect() returns a select similar to renderDepartmentSelect().

The biggest difference between renderCourseSelect() and renderDepartmentSelect() is that renderCourseSelect() dynamically populates the option children of the select.

The first option in this select is “Which course?” which has an empty string as its value. If the user has not yet selected a course, this is what they should see (just like “Which department?” in the other select). The options that follow the first come from the course list stored in state.courses.

To provide all the child option elements to the select at once, the select is given a single array as its child. The first item in the array is an element for our “Which course?” option. Then, we use the spread operator combined with map() so that from the second item on, the array contains the course options from state.

Each item in the array is an option element. Like before, each element has text that it displays (like “Which course?”) as well as a value prop. If the value of the select matches the value of the option, that option will be displayed. By default, the value of the select will be an empty string, so it will match the “Which course?” option. Once the user chooses a course and we are able to update state.course, the corresponding course will be shown.

This is a dynamic collection, we must also provide a key prop to each option to avoid warnings from React.

Lastly, we provide a change handler function, onSelectCourse() to the select prop onChange. When the user chooses a course, that function will be called with a related event object. We will then use information from that event to update the state and notify the parent.

Here’s onSelectCourse():

**forms/src/09-course-select.js**

onSelectCourse = (evt) => {

**const** course = evt.target.value;

**this**.setState({ course });

**this**.props.onChange({ name: 'course', value: course });

};

Like we’ve done before, we get the value of the target element from the event. This value is the value of whichever option the user picked in the courses select. Once we update state.course with this value, the select will display the appropriate option.

After the state is updated, we call the change handler provided by the component’s parent. Same as with the department selection, we provide this.props.onChange() an object argument with the name/value structure the handler expects.

And that’s it for our CourseSelect component! As we’ll see in the next part, integration with the form requires very minimal changes.

### Adding CourseSelect

Now that our new CourseSelect component is ready, we can add it to our form. Only three small changes are necessary:

1. We add the CourseSelect component to render().
2. We update our “People” list in render() to show the new fields (department and course).
3. Since department and course are required fields, we modify our validate() method to ensure their presence.

Because we were careful to call the change handler from withinCourseSelect (this.props.onChange) with a {name, value} object the way that onInputChange() expects, we’re able to reuse that handler. When onInputChange() is called by CourseSelect, it can appropriately update state with the new department and course information – just like it does with calls from the Field components.

Here’s the updated render():

**forms/src/09-async-fetch.js**

render() {

**return** (

<div>

<h1>Sign Up Sheet</h1>

<form onSubmit={**this**.onFormSubmit}>

<Field

placeholder='Name'

name='name'

value={**this**.state.fields.name}

onChange={**this**.onInputChange}

validate={(val) => (val ? **false** : 'Name Required')}

/>

<br />

<Field

placeholder='Email'

name='email'

value={**this**.state.fields.email}

onChange={**this**.onInputChange}

validate={(val) => (isEmail(val) ? **false** : 'Invalid Email')}

/>

<br />

<CourseSelect

department={**this**.state.fields.department}

course={**this**.state.fields.course}

onChange={**this**.onInputChange}

/>

<br />

<input type='submit' disabled={**this**.validate()} />

</form>

<div>

<h3>People</h3>

<ul>

{ **this**.state.people.map(({ name, email, department, course }, i) =>

<li key={i}>{[ name, email, department, course ].join(' - ')}</li>

) }

</ul>

</div>

</div>

);

}

When adding CourseSelect we provide three props:

* + 1. The current department from state (if one is present)
    2. The current course from state (if one is present)
    3. The onInputChange() handler (same function used by Field) Here it is by itself:

<CourseSelect

department={**this**.state.fields.department}

course={**this**.state.fields.course}

onChange={**this**.onInputChange} />

The other change we make in render() is we add the new department and course fields to the “People” list. Once a user submits sign-up information, they appear on this list. To show the department and course information, we need to get that data from state and display it:

<h3>People</h3>

<ul>

{ **this**.state.people.map( ({name, email, department, course}, i) =>

<li key={i}>{[name, email, department, course].join(' - ')}</li>

) }

</ul>

This is as simple as pulling more properties from each item in the state.people array.

The only thing left to do is add these fields to our form-level validation. Our CourseSelect controls the UI to ensure that we won’t get invalid data, so we don’t need to worry about field-level errors. However, department and course are required fields, we should make sure that they are present before allowing the user to submit. We do this by updating our validate() method to include them:

**forms/src/09-async-fetch.js**

validate = () => {

**const** person = **this**.state.fields;

**const** fieldErrors = **this**.state.fieldErrors;

**const** errMessages = Object.keys(fieldErrors).filter((k) => fieldErrors[k]);

**if** (!person.name) **return true**;

**if** (!person.email) **return true**;

**if** (!person.course) **return true**;

**if** (!person.department) **return true**;

**if** (errMessages.length) **return true**;

**return false**;

};

Once validate() is updated, our app will keep the submit button disabled until we have both department and course selected (in addition to our other validation requirements).

Thanks to the power of React and composition our form was able to take on complicated functionality while keeping high maintainability.

### Separation of View and State

Once we’ve received information from the user and we’ve decided that it’s valid, we then need to convert the information to JavaScript objects. Depending on the form, this could involve casting input values from strings to numbers, dates, or booleans, or it could be more involved if you need to impose a hierarchy by corralling the values into arrays or nested objects.

After we have the information as JavaScript objects, we then have to decide how to use them. The objects might be sent to a server as JSON to be stored in a database, encoded in a url to be used as a search query, or maybe only used to configure how the UI looks.

The information in those objects will almost always affect the UI and in many cases will also affect your application’s behavior. It’s up to us to determine how to store that info in our app.

## Optional 10 Async Persistence

At this point our app is pretty useful. You could imagine having the app open on a kiosk where people can come up to it and sign up for things. However, there’s one big shortcoming: if the browser is closed or reloaded, all data is lost.

In most web apps, when a user inputs data, that data should be sent to a server for safe keeping in a database. When the user returns to the app, the data can be fetched, and the app can pick back up right where it left off.

In this example, we’ll cover three aspects of persistence: saving, loading, and handling errors. While we won’t be sending the data to a remote server or storing it in a database (we’ll be using localStorage instead), we’ll treat it as an asynchronous operation to illustrate how almost any persistence strategy could be used.

To persist the sign up list (state.people), we’ll only need to make a few changes to our parent form component. At a high level they are:

1. Modify state to keep track of persistence status. Basically, we’ll want to know if the app is currently loading, is currently saving, or encountered an error during either operation.
2. Make a request using our API client to get any previously persisted data and load it into our state.
3. Update our onFormSubmit() event handler to trigger a save.
4. Change our render() method so that the “submit” button both reflects the current save status and prevents the user from performing an unwanted action like a double-save.

First, we’ll want to modify our state keep track of our “loading” and “saving” status. This is useful to both accurately communicate the status of persistence and to prevent unwanted user actions. For example, if we know that the app is in the process of “saving”, we can disable the submit button. Here’s the updated state method with the two new properties:

**forms/src/10-remote-persist.js**

state = {

fields: {

name: '',

email: '',

course: **null**,

department: **null**

},

fieldErrors: {},

people: [],

\_loading: **false**,

\_saveStatus: 'READY',

};

The two new properties are \_loading and \_saveStatus. As before, we use the underscore prefix convention to signal that the y are private to this component. There’s no reason for a parent or child component to ever know their values.

\_saveStatus is initialized with the value "READY", but we will have four possible values: "READY", "SAVING", "SUCCESS", and "ERROR". If the \_saveStatus is either "SAVING" or "SUCCESS", we’ll want to prevent the user from making an additional save.

Next, when the component has been successfully loaded and is about to be added to the DOM, we’ll want to request any previously saved data. To do this we’ll add the lifecycle method componentWillMount() which is automatically called by React at the appropriate time. Here’s what that looks like:

**forms/src/10-remote-persist.js**

componentWillMount() {

**this**.setState({ \_loading: **true** });

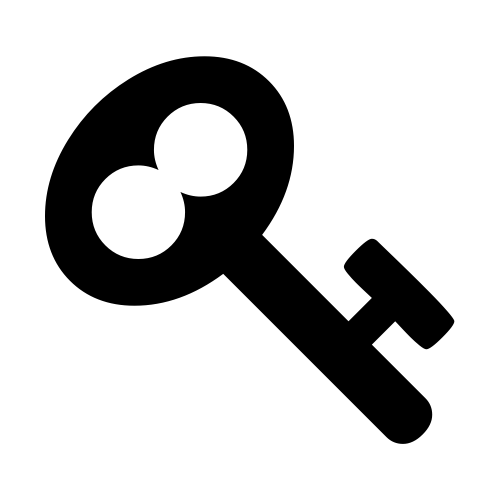
apiClient.loadPeople().then((people) => {

**this**.setState({ \_loading: **false**, people: people });

});

}

Before we start the fetch with apiClient, we set state.\_loading to true. We’ll use this in render() to show a loading indicator. Once the fetch returns, we update our state.people list with the previously persisted list and set \_loading back to false.

apiClient is a simple object we created to simulate asynchronous loading and saving. If you look at the code for this chapter, you’ll see that the “save” and “load” methods are thin async wrappers around localStorage. In your own apps you could create your own apiClient with similar methods to perform network requests.

Unfortunately, our app doesn’t yet have a way to persist data. At this point there won’t be any data to load. However, we can fix that by updating onFormSubmit().

As in the previous sections, we’ll want our user to be able to fill out each field and hit “submit” to add a person to the list. When they do that, onFormSubmit() is called.

We’ll make a change so that we not only perform the previous behavior (validation, updating state.people), but we also persist that list using apiClient.savePeople():

**forms/src/10-remote-persist.js**

onFormSubmit = (evt) => {

**const** person = **this**.state.fields;

evt.preventDefault();

**if** (**this**.validate()) **return**;

**const** people = [ ...**this**.state.people, person ];

**this**.setState({ \_saveStatus: 'SAVING' });

apiClient.savePeople(people)

.then(() => {

**this**.setState({

people: people,

fields: {

name: '',

email: '',

course: **null**,

department: **null**

},

\_saveStatus: 'SUCCESS',

});

})

.**catch**((err) => {

console.error(err);

**this**.setState({ \_saveStatus: 'ERROR' });

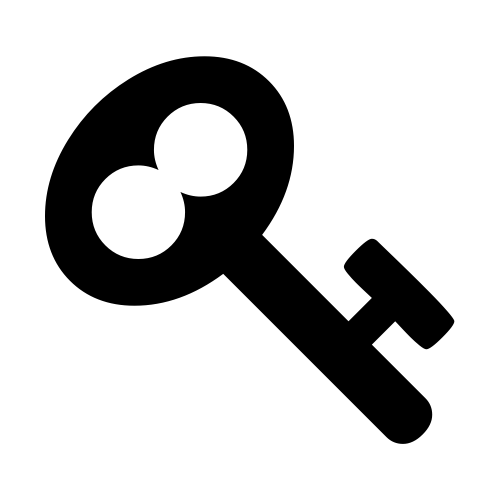
});

};

In the previous sections, if the data passed validation, we would just update our state.people list to include it. This time we’ll also add the person to the people list, but we only want to update our state if apiClient can successfully persist. The order of operations looks like this:

* 1. Create a new array, people with both the contents of state.people and the new person object.
  2. Update state.\_saveStatus to "SAVING"
  3. Use apiClient to begin persisting the new people array from #1.
  4. If apiClient is successful, update state with our new people array, an empty fields object, and \_saveStatus: "SUCCESS". If apiClient is not successful, leave everything as is, but set state.\_saveStatus to "ERROR".

Put simply, we set the \_saveStatus to "SAVING" while the apiClient request is “in-flight”. If the request is successful, we set the \_saveStatus to "SUCCESS" and perform the same actions as before. If not, the only update is to set \_saveStatus to "ERROR". This way, our local state does not get out of sync with our persisted copy. Also, since we don’t clear the fields, we give the user an opportunity to try again without having to re-input their information.

For this example we are being conservative with our U I updates. We only add the new person to the list if*apiClient*issuccessful. This is in contrast to an optimistic update, where we would add the person to the list locally first, and later make adjustments if there was a failure. To do an optimistic update we could keep track of which person objects were added before which apiClient calls. Then if an apiClient call fails, we could selectively remove the particular person object associated with that call. We would also want to display a message to the user explaining the issue.

Our last change is to modify our render() method so that the UI accurately reflects our status with respect to loading and saving. As mentioned, we’ll want the user to know if we’re in the middle of a load or a save, or if there was a problem saving. We can also control the UI to prevent them from performing unwanted actions such as a double save.

Here’s the updated render() method:

**forms/src/10-remote-persist.js**

render() {

**if** (**this**.state.\_loading) {

**return** <img alt='loading' src='/img/loading.gif' />;

}

**return** (

<div>

<h1>Sign Up Sheet</h1>

<form onSubmit={**this**.onFormSubmit}>

<Field

placeholder='Name'

name='name'

value={**this**.state.fields.name}

onChange={**this**.onInputChange}

validate={(val) => (val ? **false** : 'Name Required')}

/>

<br />

<Field

placeholder='Email'

name='email'

value={**this**.state.fields.email}

onChange={**this**.onInputChange}

validate={(val) => (isEmail(val) ? **false** : 'Invalid Email')}

/>

<br />

<CourseSelect

department={**this**.state.fields.department}

course={**this**.state.fields.course}

onChange={**this**.onInputChange}

/>

<br />

{{

SAVING: <input value='Saving...' type='submit' disabled />,

SUCCESS: <input value='Saved!' type='submit' disabled />,

ERROR: <input

value='Save Failed - Retry?'

type='submit'

disabled={**this**.validate()}

/>,

READY: <input

value='Submit'

type='submit'

disabled={**this**.validate()}

/>,

}[**this**.state.\_saveStatus]}

</form>

<div>

<h3>People</h3>

<ul>

{ **this**.state.people.map(({ name, email, department, course }, i) =>

<li key={i}>{[ name, email, department, course ].join(' - ')}</li>

) }

</ul>

</div>

</div>

);

}

First, we want to show the user a loading indicator while we are loading previously persisted data. Like the previous section, this is done on the first line of render() with a conditional and an early return. While we are loading (state.\_loading is truthy), we won’t render the form, only the loading indicator:

**if** (**this**.state.\_loading) **return** <img src='/img/loading.gif' />

Next, we want the submit button to communicate the current save status. If no save request is inflight, we want the button to be enabled if the field data is valid. If we are in the process of saving, we want the button to read “Saving…” and to be disabled. The user will know that the app is busy, and since the button is disabled, they won’t be able to submit duplicate save requests. If the save request resulted in an error, we use the button text to communicate that and indicate that they can try again. The button will be enabled if the input data is still valid.

Finally, if the save request completed successfully, we use the button text to communicate that. Here’s how we render the button:

{{

SAVING: <input value='Saving...' type='submit' disabled />,

SUCCESS: <input value='Saved!' type='submit' disabled/>,

ERROR: <input value='Save Failed - Retry?' type='submit'

disabled={**this**.validate()}/>,

READY: <input value='Submit' type='submit' disabled={**this**.validate()}/>

}[**this**.state.\_saveStatus]}

What we have here are four different buttons – one for each possible state.\_saveStatus. Each button is the value of an object keyed by its corresponding status. By accessing the key of the current save status, this expression will evaluate to the appropriate button.

The last thing that we have to do is related to the "SUCCESS" case. We want to show the user that the addition was a success, and we do that by changing the text of the button. However, “Saved!” is not a call to action. If the user enters another person’s information and wants to add it to the list, our button would still say “Saved!”. It should say “Submit” to more accurately reflect its purpose.

The easy fix for this is to change our state.\_saveStatus back to "READY" as soon as they start entering information again. To do this, we update our onInputChange() handler:

**forms/src/10-remote-persist.js**

onInputChange = ({ name, value, error }) => { **const** fields = **this**.state.fields;

**const** fieldErrors = **this**.state.fieldErrors;

fields[name] = value;

fieldErrors[name] = error;

**this**.setState({ fields, fieldErrors, \_saveStatus: 'READY' });

};

Now instead of just updating state.fields and state.fieldErrors, we also set state.\_saveStatus to 'READY'. This way after the user acknowledges their previous submit was a success and starts to interact with the app again, the button reverts to its “ready” state and invites the user to submit again.

At this point our sign-up app is a nice illustration of the features and issues that you’ll want to cover in your own forms using React.

1. <http://npm.im/validator> [↑](#footnote-ref-1)
2. [http://nodeschool.io](http://nodeschool.io/) [↑](#footnote-ref-2)