**React Formik Notes**

**Introduction**

* Forms are important for users to enter data
* Formik is a small library that helps you deal with forms in React
* Formik helps developers with the following:
  + Handle form data
  + Validation
  + Visual feedback with error messages
  + Form submission
* While you can handle forms with just React, Formik allows us to deal with forms in a scalable, performant and easy way.

**Creating an HTML Form**

* To start off, we will create a form that will be sent to YouTube that looks like:
* A picture containing text

  Description automatically generated
* When creating this form, we will be concerned with:
  + Managing the form state
  + Handling form submission
  + Validation and error messages
* To create this form, make a new react app as shown below:
* Text

  Description automatically generated Text

  Description automatically generated
* Text

  Description automatically generated Text

  Description automatically generated
* Focus on the Youtube Form component.
* Notice that since for is a reserved keyword in js, react uses ‘htmlFor’ instead of ‘for’ when dealing with forms. Recall that a label’s ‘htmlFor’ value should be the id of the input field it corresponds to. For example, the text field where the user can enter their channel name has an id of ‘channel’. Thus, the ‘Channel’ label’s htmlFor value is also ‘channel’. Not only is the htmlFor property is considered good practice to implement on a label tag, but the label’s corresponding input field is also focused when the user clicks on the lable.
* Recall that the name attribute specifies the name of an <input> element. The name attribute is used to reference elements in a JavaScript, or to reference form data after a form is submitted. Only form elements with a name attribute will have their values passed when submitting a form.

**useFormik Hook**

* To use the Formik library, we have to install it by running in cmd ‘npm i formik’
* This Formik library provides a hook called useFormik that we can import.
* The useFormik hook takes in an object as its parameter. This hook returns an object which contains a variety of user properties and methods that we can use with our form. This returned object will help us with managing the form state, handling form submission, validation and error messages.
* We can now use the useFormik hook as shown below.
* We can also log out the object that the useFormik hook returns.
* Text

  Description automatically generated Text

  Description automatically generated

**Managing Form State**

* Our youtube form has 3 input fields. However, we are not tracking the value of these three fields. When the user types in something, the value of the fields changes which means we need a state variable for that.
* Thus, we need state variables for the name, email, and channel input fields. Or collectively, we can call them as form state. The form state is an object that maintains the different form fields as shown below.
* Diagram

  Description automatically generated
* If we are able to manage the form state, we can submit this data when the user clicks on submit
* Formik helps us manage this form state.
* To do so, we will need to include an ‘initialValues’ property in the useFormik parameter object. The value of the ‘initialValues’ property is an object which contains the initial values for all our form fields. This ‘initialValues’ property’s object contains key-value pairs. The keys are names of the input field. The value is the initial value of that input field.
* Text

  Description automatically generated
* Notice how above we have an ‘initialValues’ property in the useFormik parameter function. This ‘initialValues’ property’s value is an object that contains the ‘name’, ‘email’, and ‘channel’ properties. Notice that there is a key-value pair of ‘name’ and ‘grant’. This mean that the input field with a name of ‘name’ has its initial value set to ‘grant’.
* Next, we must add the ‘onChange’ and ‘value’ props for each of the form input fields. This is required to ensure the form fields are tracked by Formik. The value of the input field’s ‘onChange’ prop will be formik.handleChange. If we don’t pass the formik.handleChange function to the ‘onChange’ property of a specific input field, that specific input field’s value cannot be changed since react does nothing when the input field changes (react does not update the formik state object).
* The value of the input field’s ‘value’ prop will be formik.values.INPUT\_FIELD’S\_NAME\_PROP\_VALUE. An example of this is shown below
* Text

  Description automatically generated
* Notice that the onChange prop has a value of formik.handleChange. Notice that the value we pass to the ‘value’ prop is formik.values.email since the input field has a name of ‘email’. When we change the email input field, the formik.handleChange function is executed which updates formik.values and this update value is then passed to the value ‘prop’ in the email input field.
* We can also access the values from the form through formik.values which we can log to the console.
* Now if we run the following code, we get the following webpage output:
* Text

  Description automatically generated
* Webpage Output: Table

  Description automatically generated with low confidence
* Console output:Text

  Description automatically generated
* Notice in the console, we log out the formik.values object. This is an object where the key corresponds to name attribute of the form input field and the value corresponds to the value of the form field. The value of this formil.values object will be the initialValues object we specified in the useFormik parameter.
* Now, if we enter ‘a’ in the channel input field, we get the following:
* Graphical user interface, application

  Description automatically generated
* Notice how the updated formik.values are logged to the screen which means the component rerenders. The reason why there are two rerenders is explain later.

**Handling Form Submission**

* We used formik to keep track of the form state, but now we want to get hold of this form state when the user submits the button.
* To handle form submission, we need to first specify the ‘onSubmit’ property on the form tag. The value of this ‘onSubmit’ property is formik.handleSubmit.
* 
* We also need to add a new ‘onSubmit’ property to the object we pass to useFormik as a parameter. This ‘onSubmit’ property’s value is a function. This function automatically receives the form state’s ‘value’ object as its argument. Since we set the ‘onSubmit’ property of the form to formik.handleSubmit, this function is executed whenever the form’s is submitted.
* Graphical user interface, text

  Description automatically generated
* Now, our code looks like the following:
* Text

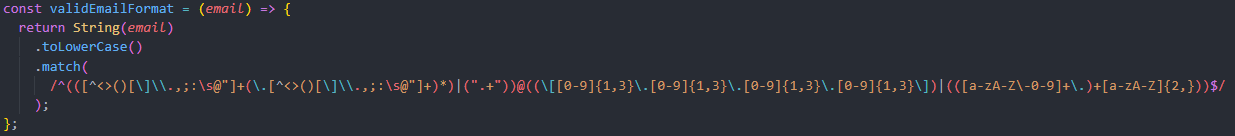
  Description automatically generated
* Enter the following input: Graphical user interface, text, application, email

  Description automatically generated
* Now, we see that the following is logged to the console:
* Graphical user interface, text

  Description automatically generated
* In our example, the ‘onSubmit’ property’s value is a function that logs out the values. In real world cases, this function likely makes an API call to the server to post the form data.
* Notice that the webpage did not reload when we click submit.
* To get rid of the above warning, add type = ‘submit’ to the button.

**Form Validation**

* Now that we are able to get hold of the form state when the user submits the form, we also want to have field validation.
* Let’s make it so that every field is required (meaning all fields are mandatory and have to be filled). Moreover, we want the email field to be a valid e-mail format (not necessarily a valid e-mail, but rather valid e-mail format).
* To add validation, we need to add a ‘validate’ property to the object we passed into useFormik. The value of this ‘validate’ property will be a function.
* This ‘validate’ property’s function automatically receives the form state’s ‘value’-property object as its argument (which we will call values). Thus, if we have input fields with names of ‘name’, ‘email’, and ‘channel’, we can access the values in those input fields via ‘values.name’, ‘values.email’, and ‘values.channel’.
* This ‘validate’ property’s function returns an object. Additionally, the keys of this returned object must match the keys of the form state’s ‘value’ object. Thus, if the function parameter (which we will call ‘values’), has name, email, and channel properties (corresponding to the values with the input fields with names of ‘name’, ‘email’, and ‘channel’), the ‘validate’ property’s function’s return object should also have name, email, and channel properties. The value of these keys of this returned object should be a string indicating the error message for that particular field.
* This ‘validate’ property’s function gets executed every time the form state changes.
* A validate function could look like the following:
* Text

  Description automatically generated
* Notice this validate function takes in the form state values as its parameter and we access the values associated with the name, email, and channel input fields via values.name, values.email, and values.channel.
* Notice that we return an object (which in this case is the errors object). Moreover, notice this return error object may or may not have a name, email, or channel property depending on the if conditions. Its is fine if the name of an input field is not a property within the errors object. However, it is invalid to have a property within the errors object not be the name of an input field.
* Note that validEmailFormate is some function that we looks like the following (an absolute mess):
* 
* Now, our code could look something like the following
* Text

  Description automatically generated
* Notice that we refactored it to use es6 object literal syntax in the useFormik parameter
* Also notice that we collapse the returned JSX since that change from our previous code snippet of the YoutubeForm component.
* The webpage output looks like the following: A picture containing table

  Description automatically generated
* As of now, there is nothing logged to the console.
* Now, let’s remove the ‘t’ from ‘grant’ and we get the following:
* Webpage output: A picture containing table

  Description automatically generated
* Console output:A picture containing logo

  Description automatically generated
* Since we changed the value of the name input field from ‘grant’ to ‘gran’, the state of the form changed, executing the validate function (which gets executed on every form change). The validate function logs the error object which in this case, doesn’t have a name property which is fine. But the error object has a channel and email property, and their values are “required” which is a string representing the error.
* Now, if we type ‘g’ in the email input field and then type ‘g’ in the channel input field, we get the following:
* Webpage output: A picture containing table

  Description automatically generated
* Console output: Graphical user interface, text, application, email

  Description automatically generated
* Notice how we the third message logged to the console only has an email property which is ‘invalid email format’. This is because we did enter an email of ‘g’, but it is not a valid email format.

**Display Error Messages**

* In the previous section, we defined a validate function that checks form fields and returns an error object. Now, we will learn how to display the error messages.
* The formik object returned by useFormik includes a property called ‘errors’. Initially (on page load), the value of this error object is an empty object. Once we change the form state, the validate function gets executed, returning an error object which becomes the value of ‘errors’ property in the formik object.
* Ex:
  + Suppose we have the following code (notice that we log out formik.errors on component rerender) :
  + Text

    Description automatically generated
  + Before we type anything, the output is as shown below:
  + Webpage output: A picture containing table

    Description automatically generated
  + Console output: 
  + Notice how the formik.errors object we logged to the console is an empty object at first.
  + Now, let’s remove the ‘t’ in ‘grant’.
  + Webpage output:
  + A picture containing table

    Description automatically generated
  + Console output: Graphical user interface, text, application, email

    Description automatically generated
  + Notice how the last object logged to the console (the formik.errors object) is now populated with key-value pairs of input field names and their errors.
* Now that we learned we have access to form errors via the formik.errors object, we can conditionally render error messages as shown below (focus on line 53, 63, and 73):
* Text

  Description automatically generated
* Notice how in line 63, we say if the formik.errors object has a ‘email’ property (which checks if there is an error with the input field with a name of ‘email’), then render the error message with the email input field.
* Thus, when we load the page, we get the following:
* Webpage output: A picture containing table

  Description automatically generated
* Console output: 
* Recall that the formik.errors object is initially {}, so none errors on line 53, 63, and 73 get rendered and no error object is logged to the console (since the validate function didn’t execute yet).
* Now, if we remove the ‘t’ in ‘grant’, we get the following:
* Webpage output: A picture containing table

  Description automatically generated
* Console output: Graphical user interface, text, application, email

  Description automatically generated
* This change in the input field caused a change in the form state, causing the component to rerender twice (we will learn why it renders twice later). While the component rerenders twice, the validate function is only executed once and logs out the error object. Notice that the validate function is executed before the first rerender the validate function’s console log appears before the component’s console log. Because the errors object is no longer {}, we see the errors displayed on the webpage output from the conditional rendering.
* While the error messages are displayed, notice how the error message for the email and channel fields are displayed even though we didn’t visit those fields yet (we just interacted with the name field).

**Visited Fields**

* We learned in the previous section how to display errors, now we want to learn how to display a given input field’s error only if the user has visited that input field.
* To keep track of the field that the user has interacted with, we can again use the formik object.
* To start tracking a specific input field’s visited status, we add the ‘onBlur’ property to the input field. The value of this ‘onBlur’ property will be formik.handleBlur.
* To access which input fields have been visited, we use the formik.touched object which is initially set to {} since none of the fields have been visited yet.
* Ex:
  + Text

    Description automatically generated
  + In the above code, we only added the ‘onBlur’ property to the channel input field. The name and email input fields do not have a ‘onBlur’ property. We also added a console log of formik.touched on line 40.
  + When we start the app, we get the following output:
  + Webpage Output: A picture containing table

    Description automatically generated
  + Console Outut: A picture containing chart

    Description automatically generated
  + Since none of the fields have been touched, the console logs a {}.
  + Now, let’s click on the channel input field, type, nothing, then click somewhere else on the webpage:
  + Webpage output: A picture containing table

    Description automatically generated
  + Console output: A screenshot of a computer

    Description automatically generated with medium confidence
  + When we click the channel input field, nothing happens since we are still interacting with the input field. But once we click away (we are done interacting with the input field), the form state has changed since one of the fields visited status has changed (namely the channel field). This causes the component to rerender twice and execute the validate function once. Recall the validate function is executed before the first rerender which is why we can see the error object logged to the console first.
  + Now, when we rerender the component, the formik.touched object is logged and we see that it is a ‘channel’ property and its value is true. Also notice that the errors for both the email and channel input fields are shown on the webpage.
* The previous example only applied onBlur to the channel input field and simply logged the formik.touched object.
* To display a given input field’s error only if the user has visited that input field, we can have the following code.
* Text

  Description automatically generated
* Notice we added the ‘onBlur’ property to all the other input fields. As well, we changed the condition for rendering the input field’s error from {formik.errors.email && <div>{formik.errors.email}</div>} to {formik.errors.email && formik.touched.email && <div>{formik.errors.email}</div>}.

**Schema Validation with Yup**

* While we already learned how to add form validation via Formik, there is an alternative way using the Yup library. More information on Yup at <https://github.com/jquense/yup>
* To use the Yup library, install it by running in cmd: ‘npm i yup’
* After install Yup, we can use it by importing it as shown below:
* 
* We will not learn how Yup works. Instead, we will learn how to rewrite our validate function with Yup.
* To use Yup, we first create the a validation schema object. To do so, we can use the Yup.object method which returns a validation schema object that we can store in a constant. We then pass in an object which contains key-value pairs. The keys are the name of a given input field. The value are the rules/assertions that the given input field must satisfy.
* Text

  Description automatically generated
* In the ‘email’ property, we assert that the input is a string via string(). We assert that the input is required via required(‘Required’) which also sets the error message for failing the required assertion to be ‘Required’. We assert that the input is in a valid email format via .email(‘Invalid email format’) which also sets the error message for failing the required assertion to be ‘Invalid email format’. Note that the order of these assertions does not matter. A similar concept applies for the ‘name’ and ‘channel’ properties. Some other insertions include number(), positive(), integer(), url(), date(), default().
* Secondly, we pass this validationSchema object into our useFormik hook as shown below (we no longer need to use the validate function so we can comment it out):
* Text

  Description automatically generated
* Notice how the above validationSchema object is syntactic sugar to replace the validate function(shown below). Text

  Description automatically generated
* Since we longer need the validate function and no longer need the validate property in the object passed to the useFormik hook, we can remove the validate function and the validate property.

**Reducing Boilerplate**

* While we already used Formik to make dealing with forms easier, there are more Formik features that allow us to reduce boilerplate code.
* formik.getFieldProps
  + When we look at the three form input fields, we see that the onBlur, onChange, and value props are similar across all three form input fields. Thus, these props are boilerplate code.
  + To get rid of the boilerplate code, we can use a method called formik.getFieldProps() inside curly brackets while using the spread operator. This method takes in one parameter which is the name of the form input field. This method will add the onBlur, onChange, and value props to the specified input field. These props are added behind the scenes.
  + Before using formik.getFieldProps(): Text

    Description automatically generated
  + After using formik.getFieldProps(): Text

    Description automatically generated
  + We can now apply the formik.getFieldProps to the email and channel input fields as well.
* While the formik.getFieldProps does reduce boilerplate code, we still have to manually pass in the name of the input field into the function as a parameter.
* To further save us time, Formik provides components that implicitly use react context to reduce boilerplate code.
* We will look at 4 of the several components (listed below) that Formik provides to refactor our code:
  + Formik
  + Form
  + Field
  + ErrorMessage
* Formik Component
  + The Formik component is a replacement to the useFormik hook.
  + While The Formik component doesn’t reduce boilerplate code itself, it behaves like a context provider component that provides the different properties and helper methods for the Form, Field, and ErrorMessage components that we will later use. We will then use the Form, Field, and ErrorMessage components to reduce boilerplate code.
  + Using the useFormik hook, we passed in an object with properties such as ‘initialValues’, ‘onSubmit’, and ‘validationSchema’. Using the Formik component, we will pass in those properties as props.
  + Since we are no longer using the useFormik hook, we can remove the following:
  + Text

    Description automatically generated
  + To use the Formik component, import Formik from ‘formik’
  + 
  + Then, wrap the entire form with the Formik component.
  + Then, pass in the props to the Formik component.
  + Now, our code should look like the following.
  + Text

    Description automatically generated
  + Notice the use of the Formik Component in line 21 along with the props we passed to it
* Form Component
  + To use the Form component, import Form from ‘formik’
  + 
  + Then replace the HTML form element with the Form component.
  + Then remove the ‘onSubmit’ prop. We can remove this prop since the Form component is a wrapper around the HTML form element that automatically hooks into Formik’s handleSubmit method.
  + Text

    Description automatically generated
  + Notice the use of the Form component in line 26 to replace the HTML form element ang the removal of the ‘onSubmit’ prop
* Field Component
  + The Field component simplifies the code for a form field
  + We are currently manually pass in the name of the input field into the getFieldProps function as a parameter. The passing of this input field name into the function is a common pattern shared among all the different form fields, hence boilerplate code exists.
  + Formik provides the Field component to resolve this issue.
  + To use the Field component, we have to import Field from ‘formik’
  + 
  + Then, replace the HTML input tags with the Field component.
  + Then, remove the formik.getFieldProps method from each of the fields. We can remove this getFieldProps method since the Field component does three things behind the scenes. First, the Field component hooks up to the Form component, allowing the Field to access handleBlur, handleChange, and values. Second, it uses the name attribute to match up with the Formik state. Third, the Field component will render an input element by default.
  + Text

    Description automatically generated
  + Our code should now look like the above.
  + Notice the use of the Field component in lines 27, 33, and 39
* ErrorMessage Component
  + Currently, when we want to display an error message for any field, we have to check if the error exists and if the input field has been touched. In code, we do this: Text

    Description automatically generated.
  + Since this pattern exists whenever we try to render an error message, boilerplate code exists.
  + Formik provides the ErrorMessage component that resolves this issue.
  + To use the ErrorMessage component, import ErrorMessage from ‘formik’
  + Then, replace the block of code (Text

    Description automatically generated), with the following ErrorMessage component.
  + We then have to include a ‘name’ property in the ErrorMessage component. This ‘name’ property’s value is the name attribute of the input field that the message corresponds to.
  + 
  + This replacement is possible because behind the scenes, the ErrorMessage component takes care of rendering the error message for the particular input field (indicated by the ‘name’ prop passed to it) only if the field has been visited and an error exists.
  + Text

    Description automatically generated
  + Our code should now look like the above. Notice the use of the ErrorMessage component on lines 28, 32, and 36.
* Now, we are done refactoring our code and no more errors exist.

**Field Revisited**

* We already know that the Field component renders an HTML input element by default. As well, behind the scenes, the Field component hooks up the input element to to Formik’s handleChange, handleBlur and values.
* There are other things about the Field component.
* The field component can take in HTML attributes as props.
  + For example, we can add a ‘readOnly’
  + 
  + When we look for that Field component when we inspect the page (not Ctrl+U), we see the following:
  + 
  + Notice the input field has a readonly attribute.
* The field component can also take in a placeholder value
  + For example, we add the placeholder value of ‘Yo’ below:
  + 
  + Webpage output: A picture containing table

    Description automatically generated
  + Inspect element output: 
  + Suppose we add a placeholder value to an input field, but that input field already has an initialValue (as specified by the Formik components’ ‘initialValues’ prop). The placeholder value we added will not get displayed, but rather the ‘initialValues’ prop’s value will get displayed.
* The field component can also render a different element other than the input element.
  + To determine what kind of element the Field prop should render, we pass the ‘as’ prop to the Field component. The value associated with this ‘as’ prop can either be ‘input’, ‘textarea’, ‘select’, or a custom react component.
  + Suppose we want to make the channel input a text area field instead of a text input field
  + To do so, we pass the ‘as’ prop to the Field component. The value associated with his ‘as’ prop will be ‘textarea’.
  + 
  + Webpage output:Graphical user interface, text, application

    Description automatically generated
  + Inspect element output: 
  + Alternative to the ‘as’ prop, we could use the ‘component’ prop. The internal implementation of the two slightly differ. As well, the ‘component’ prop was depricted, then undepricated (so we should just stick to the ‘as’ prop).
* The field component can also use the renderProps pattern
  + The render props pattern give us more fine-grained control over the rendering and behavior of our Field component.
  + Suppose we need to render another input element to collect the user’s address.
  + We could do the following:
  + Text

    Description automatically generatedText

    Description automatically generated
  + Note that we added ‘address’ to the initialValues object.
  + While this works, we could also use the renderProps pattern.
  + We still need to add ‘address’ to the initialValues object.
  + Text

    Description automatically generated
  + Recall in the renderProps pattern, we can use a function as the children to the component. This function will return JSX which in our case will be an input element for the user to enter their address.
  + Text

    Description automatically generated
  + However, as of now, the input element is not hooked up to Formik’s handleBlur, handleChange, and values methods.
  + To resolve this issue, we add a parameter to the function called ‘props’. We can log out the ‘props’ as shown below.
  + Text

    Description automatically generated
  + Console output:
  + A picture containing text

    Description automatically generated
  + Notice there are three properties in this props parameter. Namely, they are ‘field’, ‘form’, and ‘meta’.
  + As well, notice the ‘field’ property has within it ‘name’, ‘onBlur’, ‘onChange’, and ‘value’ properties. These are all the properties necessary for formik to handle the state.
  + As well, notice the ‘form’ property has within it ‘errors’, ‘values’, ‘touched’, ‘handleChange’, ‘handleBlur’ properties.
  + As well, notice the ‘meta’ property which gives us information on whether a particular field has visited, if there is an error, etc. This information is what we can use to render error messages.
  + Now, we can destructure the field and meta properties, we don’t really need the form property. Then, we can destructure field object’s ‘name’, ‘onBlur’, ‘onChange’, and ‘value’ properties as props to the input element. We can also use the meta object’s ‘touched’ and ‘error’ properties to render the error message.
  + Text

    Description automatically generated
  + Using renderProps is overkill for what we want to do here, but for more complex forms, renderProps will give us more control and better scalability.

**ErrorMessage Revisited**

* We already know that the ErrorMesage component accepts a ‘name’ prop whose value is the name attribute of the input field the ErrorMessage component represents. This ErrorMessage component renders the error message for that particular field if the field has been visited and an error message exists for that field.
* There are other things about the Field component.
* When we inspect the ErrorMessage, we get the following:
* Text

  Description automatically generated with medium confidence
* Notice how the ErrorMessage is just a text node, it’s not wrapped inside any HTML element.
* The ErrorMessage component provides a way to wrap the error message inside an HTML element
  + To wrap it inside an HTML element, we can add a ‘component’ prop to ErrorMessage. This ‘component’ prop’s value can be set to HTML elements such as ‘div’, ‘p’, ‘h1’, etc.
  + 
  + Text

    Description automatically generated with medium confidence
* Not only can we set the value of the ErrorMessage’s ‘component’ prop to be an HTML element, it can also be a custom react component.
  + Suppose we make the following new TextError component:
  + Text

    Description automatically generated
  + This component turns the text color of its children component to be red.
  + Now, we can set the value of our ErrorMessage’s ‘component’ prop to be the TextError component
  + 
  + Now, when we get an error message, the text is red:
  + Table

    Description automatically generated with low confidence
* The ErrorMessage component can also use the renderProps pattern
  + When we follow the renderProps pattern, we specify the children to be a function and this function receives the error message as a string as its parameter.
  + To demonstrate this, when we run the below code and get an error in the email input field, we get the following:
  + Text

    Description automatically generated
  + 
  + Now, we can use this renderProps function to render the TextError component which will make the errorMessage a red color.
  + Text

    Description automatically generated

**Nested Objects**

* When we look at the initialValues object, there are 4 fields. We track our forms values in the same structure as the following key value pairs.
* Text

  Description automatically generated
* And when we log the form state values to the console via the onSubmit function, we get the following:
* A picture containing graphical user interface

  Description automatically generated
* While the above works, sometimes we may want to group these properties together into its own object. This may be because the API requires the data to be structured in a certain way.
* To do so, we will need nested objects.
* Suppose we want to prompt the user for their facebook and twitter information and group it together in one heading of ‘social’s’. Thus, the initialValues object would look like the following:
* Text

  Description automatically generated
* Make sure to also add the Field components to prompt the user for their ‘facebook’ and ‘twitter’. However, since facebook and twitter are nested, their name attributes are ‘socials.facebook’ and ‘socials.twitter’.
* Text

  Description automatically generated
* As well, to validate the nested object, you can do the following:
* Text

  Description automatically generated

**Arrays**

* In the previous section, we learned how to group data into its own object. In this lesson, we will learn how to group data into its own array.
* Suppose we want to prompt the user for their primary and secondary phone numbers and group it together in one heading of ‘phoneNumbers’. Thus, the initialValues object would look like the following:
* Text

  Description automatically generated
* Make sure to also add the Field components to prompt the user for their first and second phone numbers. However, since these phone numbers are nested, their name attributes are ‘phoneNumbers[0]’ and ‘phoneNumbers[1]’. The 0 and the 1 represent the index positions in the phoneNumbers array in the initialValues.
* Text

  Description automatically generated
* As well, to validate the array, you can do the following:
* Text

  Description automatically generated

**FieldArray Component**

* The formik library also provides the FieldArray component which helps with dynamic form controls (usually with common array manipulations).
* We previously learned how to manage the state of the form field as an array in our initialValues object (as shown below)
* Text

  Description automatically generated
* We allowed the user to input two phone numbers. However, we had to hard code these input fields by referencing the index (as shown below);
* Text

  Description automatically generated
* However, if we need to collect more phone numbers, this method is not scalable.
* Instead of rendering two input fields for the two phone numbers, we should start off with one input field for the user to enter a phone number. In addition, we also provide the user the option to add more input fields to add more phone numbers. As well, we can also provide the user the option to remove input fields. This list of phone numbers would be managed as an array in our form state.
* To do so, we first need to import the FieldArray component from ‘formik’
* 
* Then, we have to add a property to our initialValues object. This property will be the name we use to reference the array of phone numbers.
* Text

  Description automatically generated
* In our example, the we added the ‘phNumbers’ property. We set its value to an array with just one item since we start off with just one input field since the user initially only adds one phone number.
* Instead of using Field components to prompt the user for inputs, we use the FieldArray component. This FieldArray component takes in a ‘name’ prop/attribute. To use the dynamic form control, we will need to use the renderProps pattern for the FieldArray component. The renderProps function’s parameter is an object with many properties and methods that will allow us to modify/access the form field state. For now, we will just log that parameter object to see it’s value. Let’s call the parameter object ‘fieldArrayProps’.
* Text

  Description automatically generated
* Console Output:
* Text

  Description automatically generated
* The main methods we will use in this ‘fieldArrayProps’ are the push and remove methods. Moreover, ‘fieldArrayProps’ has a ‘form’ property which a ‘values’ property as shown below.
* Text

  Description automatically generated
* Notice the values for our phNumbers field are accessible via this specific property. We can use this property along with the push and remove methods to render out the list of inputs as shown below.
* Text

  Description automatically generated
* Webpage Output: Graphical user interface, text, application

  Description automatically generated
* In line 84 to 87, we destructure to fieldArrayProps to get hold of the push/remove methods and also the phNumbers property whose value is the list of phone numbers in our form state.
* We In line 90, we map over the list of phone numbers and for phone number, we will render a Field component and two button components.
* The Field component allows the user to input the phone number. Notice how we set the ‘name’ prop of the Field component to by phNumbers[${index}]. This is similar to when we previously had a static number of input fields for a static number of ‘phoneNumbers’ and we hard coded their index. In our case, we use the index parameter within the map function so we don’t have to hard code the index of the ‘name’ property for each Field component.
* The first button is a button that appends a new phone number to our list of phone numbers in our form state. We set the value of this new number to be “” since the user hasn’t inputted anything. Since we added a new item, we changed the form state, causing the entire form to rerender.
* The second button is a button that removes the phone number at the current index from our list of phone numbers in our form state. Since we remove a new item, we changed the form state, causing the entire form to rerender. Additionally, this second button is conditionally rendered. It is only rendered if the index is greater than 0. This is why in the webpage output, the first row does not have a Delete Phone Number button. This feature is important because of the following. If the there was only 1 phone number in our list of phone numbers and we displayed an button to delete it, the user could click it and would not be able to add new phone numbers. The list of phone numbers would forever just be an empty array.

**FastField**

* Fomik provides the FastField component which is meant for performance optimization
* The FastField component should be considered if our form has more than 30 fields or fields with very complex validation requirements.
* To determine the differences between the Field and FastField components, we will add a console log to Field component in the address input field. We choose the address input field since it follows the render props method which makes it very easy to add a console.log statement
* Text

  Description automatically generated
* On page load, the console output is: which makes sense since this is the address input field’s first render.
* Now, type the letter ‘a’ into the channel name. We now get the following console output:
* Graphical user interface, text, application, email

  Description automatically generated
* Notice that ‘render’ is getting logged to the console. Thus, we see that changing the value of the channel input field caused the address input field to rerender.
* This means that a change in one input field is a change in the form state is causing all of the fields in our form to rerender as well.
* Now, let’s replace Field with FastField.
* To use FastField, we have to import FastField from formik
* 
* Then, we will replace the address input’s Field component to a FastField component.
* Text

  Description automatically generated
* On page load, the console output is: Graphical user interface, text, application

  Description automatically generatedwhich makes sense since this is the address input field’s first render.
* Now, type ‘a’ into the channel input field, notice nothing is logged to the console.
* Now, type ‘a’ into the address input field. We get the following logged to the console:
* Graphical user interface, text, application, email

  Description automatically generated
* FastField is an optimized version of the Field component which internally implements the shouldComponentUpdate lifecycle method to block all additional rerenders unless there are direct updates to the FastField form control itself.
* Thus, if a particular field is independent from all the other fields in your form, we can use the FastField component.
* However, we should use this component with caution as there are only a few conditions under which a FastField component will rerender (more information at <https://formik.org/docs/api/fastfield>)

**When does validation run**

* As of now, we know that once the validation rules run, formik auto-populates the formic dot errors object with the error messages. We can use this fact to monitor exactly when validation runs
* In the form numbers field, we can add a log statement of the form errors in the renderProps function.
* Text

  Description automatically generated
* When we load the page, we get the following in the console:
* 
* We have an empty form error object which makes sense since the form validation not been executed yet. As well, it is only logged once since this is the initial render.
* This errors object gets populated (meaning the validation function get executed) when:
  + 1. A change event occurs
    - If we type something into one of the form input fields, we see that the error object gets populated:
    - Shape, rectangle

      Description automatically generated
    - Graphical user interface, text, application, email

      Description automatically generated
  + 2. A blur event occurs
    - If we click a field, then click somewhere else on the form, we see that the error object gets populated:
    - A picture containing text

      Description automatically generated
    - Graphical user interface, text, application, email

      Description automatically generated
  + 3. Form submission is attempted
    - If we don’t interact with any o the form input fields and just click submit, we see that the error object gets populated
    - Graphical user interface, application

      Description automatically generated with medium confidence
    - Graphical user interface, text, application, email

      Description automatically generated
    - Note that if one or more fields don’t pass validation, the onSubmit function never gets executed.
* Sometimes, we might not want formik to automatically run the validation function for us
* Thus, formik provides two props to prevent validation from occurring after a change or blur event.
* On the Fromik component, we can specify a prop called ‘validateOnChange’ and set the value to false. This will instruct formik to not run the validation function on a change event.
* Similrly, we can also specify a prop called ‘validateOnBlur’ and set the value to false. This will instruct formik to not run the validation function on a blue event.
* Text

  Description automatically generated
* When we run our code and then type into an input field and then click somewhere else on the webpage that was not in the field, we get the following:
* Graphical user interface, application, email

  Description automatically generated
* Notice that the fields still rerender (since the error object is logged every time the form rerenders). However, the error object is not populated, it remains with a value of {}.

**Field Level Validation**

* We previously learned two ways to specify validation rules: a validation function or the yup validation schema object. These validation rules are then passed into the ‘validate’ or ‘validationSchema’ props on the Formik component. Both of these props are available on the top-level formic component.
* Formik also allows us to provide a validation function at the field level. This field level validation function will be similar to the top-level formic component validation function.
* Suppose we want to a validation function that only checks the twitter input field.
* To do so, we create a new function (usually outside the component function). This function will automatically receive the value of the twitter input field as its parameter. Thus, let’s call the parameter ‘value’. This function will return a string representing the error message associated with ‘values’.
* Text

  Description automatically generated
* Once we have the validate function, we add a ‘validate’ prop to the Field component for the twitter input field. We will set the value of the ‘validate’ prop to be the validateTwitter function we just defined. Note that this ‘validate’ prop can also be added to the FastField component.
* Text

  Description automatically generated
* Make sure to set the Fromik component’s ‘validateOnChange’ and ‘validateOnBlur’ props to be true. As well, remove any validation regarding the twitter field from the top-level validate function and the Yup validation schema object.
* Now, if we run the webpage, click on the twitter field, then click away, we get the following webpage:
* Application

  Description automatically generated with low confidence

**Manually triggering validation**

* To be able to trigger validation manually, Formik provides us with two helper methods. In order to access these methods, we have toe use the renderProps pattern on the entire Form component.
* The top level Formik component will have children which is a function to follow the renderProps pattern. This function will receive some props which we call as formic. This function will return the entire form component.
* Before using renderProps:
* Text

  Description automatically generated
* After using renderProps:
* Text

  Description automatically generated
* By using the renderProps pattern, we have access to formik object (which allows us to control everything that has to do with our form) which is the parameter to the renderProps method.
* For now, we will just log that formik parameter to the console.
* Text

  Description automatically generated
* Console output:
* Text

  Description automatically generated with medium confidence
* This formik parameter object is actually the same as the return value of the useFormik hook. As well, in the Field and FieldArray components that used the renderProps pattern, we destructured a ‘form’ property which is the same as this formik parameter object.
* Now a question that arises is why we have the same object appearing at both the form and field level. We should use the field level version when we need to deal with an individual field. Likewise, we should us the form level version when we need to deal with the entire form. Even though we the form level version should manage just the form level, we can use it manage both the form and field level functionality.
* Our intention is the manually trigger validation at both the form level and the field level.
* We will add two buttons right before the submit button. The first will validate the twitter field. The second validate all fields within the form.
* The formik parameter object provides the ‘validateField’ and ‘validateForm’ methods which we will make use of in our two buttons.
* In the first button, we will add an onClick property whose callback function executes the formik.validateField() method. This method takes in the name of the field we want to validate as a parameter. This formik.validateField() will validate the field that we specified in the parameter.
* In the second button, we will add an onClick property whose callback function executes the formik.validateForm method. This method doesn’t take in any parameters. This formik.validateForm will validate the entire form.
* Text

  Description automatically generated
* Now, when we run our code and then click the ‘validate twitter field’ button, we see that nothing changed on the webpage. If we check the console, we see an object. Examining the ‘errors’ and ‘touched’ property of that object, we see the following:
* Error property:Text

  Description automatically generated
* Touched property: 
* Notice the error object does get populated with the twitter field’s error. Recall that in order for an error to be rendered, the input field’s value must cause an error and the field must be visited. Since we didn’t visit the field as the ‘touched’ property’s value is an empty object, the error is not rendered on the webpage.
* If we rerun the code and then click the ‘validate all’ button, we see that nothing changed on the webpage. If we check the console, we see an object. Examining the ‘errors’ and ‘touched’ property of that object, we see the following:
* Error property: Text

  Description automatically generated
* Touched property: 
* Notice the error object does get populated with the every form field’s error (if an error exists). Recall that in order for an error to be rendered, the input field’s value must cause an error and the field must be visited. Since we didn’t visit the field as the ‘touched’ property’s value is an empty object, the error is not rendered on the webpage.
* If we want to display these errors (by changing the value of the ‘touched’ property), we can use two more helper methods that the formik parameter object provides, setFieldTouched and setTouched.
* setFieldTouched takes in one parameter which is the name of a form field. This setFieldTouched function will add that parameter field to the ‘touched’ property object. Calling this function will trigger validation to run if validateOnChange is set to true (which it is by default)
* setTouched takes in one parameter which is an object that contains all the fields we want to have touched set to true. Calling this will trigger validation to run if validateOnBlur is set to true (which it is by default). You can also explicitly prevent/skip validation by passing a second argument as false. In our example, we will want to set the ‘touched’ property to include the name, email, twitter, and facebook fields.
* Text

  Description automatically generated
* Now, if we run our code and click the ‘visit twitter field’ button, we get the following:
* Webpage: A picture containing application

  Description automatically generated
* Console: Text

  Description automatically generated Text

  Description automatically generated
* Notice that the ‘errors’ property does not include just the twitter field, but also the other fields. The other fields are added to the ‘errors’ property since setFieldValue triggers validation to run if validateOnChange is set to true (which it is by default). However, notice only the twitter field is added to the ‘touched’ property.
* Now, if we run our code and click the ‘visit all’ button, we get the following:
* Webpage:Graphical user interface, text

  Description automatically generated with medium confidence
* Console: Text

  Description automatically generated Text

  Description automatically generated
* Notice that the ‘errors’ does not just include the fields we specified in the setTouched method, but also the other fields such as phoneNumbers. Other fields such as phoneNumbers are added to the ‘errors’ property since the setTouched method triggers validation to run if validateOnBlur is set to true (which it is by default). However, notice only the fields that we specified in the setTouched method are added to the ‘touched’ property.
* In our example, we used the form level formik paramter to access the helper methods. We could also access these methods from the field level version of the formik parmater via renderProps in the field components.
* One use of this manually validation triggering is having a button that allows the user to check if a username already exists.

**Disabling Submit Button**

* We can disable the submit button based on two scenarios:
  + the validity of the form state
  + when the form submission is in progress.
* Disabling the submit button based on validity of the form state
  + Recall that we used the renderProps pattern with the Formik component to get access of the form level formik parameter object. This object has a property called ‘isValid’. This is a read-only property which is set to true if the errors object is empty and false if the errors object is not empty.
  + We can use this ‘isValid’ property to disable to submit button.
  + 
  + In the above code snippet, we disable the submit button if the form is invalid. If the form is valid, we don’t disable the button.
  + Let’s run the code and on page load, we get the following:
  + Webpage output: Graphical user interface, text, application

    Description automatically generated
  + Console output: Graphical user interface, text, application

    Description automatically generated
  + Notice that the submit button is not disabled. However, this is strange since we have some required fields such as email and channel. The reason the submit button is not disabled is because on page load, the initial values of the ‘error’s property in the form level formik parameter object is an empty object. As a result, the form level formik parameter object’s ‘isValid’ property is true.
  + However, if we change the form state (such as by typing at in a field), the ‘errors’ property gets updated, causing the ‘isValid’ property to be updated. Thus, if we type ‘g’ in the email field for example, we get an error since the input is not an email format. Thus, the ‘error’ property has a ‘email’ property. Since the ‘error’ property is not an empty object, the ‘isValid’ property is set to false so the submit button is disable.
  + What if we want the submit button to be disable right when the page loads?
  + There are two ways to solve this.
  + Method 1:
    - The first method is to set the Fromik component’s ‘validateOnMount’ prop value to be true. Since the value of this prop is a boolean, we can just say validateOnMount. Setting this ‘validateOnMount’ prop to be true will cause formik to run the validation against each field as soon as the form mounts, causing the ‘errors’ property to possibly be populated.
    - Text

      Description automatically generated
    - The drawback of this method is that if our validation is very complex, it can use too much performance. Thus, this method is more suitable for forms with fewer fields and simpler validations.
  + Method 2:
    - The second method is to use the form level formik parameter object’s ‘dirty’ property. This ‘dirty’ property is a boolean value that indicates if at least one of the form field’s values has changed since it was initialized.
    - When we run our application, we see the following (not we set the Formik component’s ‘validateOnMount’ property to still be true) :
    - Webpage: Graphical user interface, text, application

      Description automatically generated
    - Console:
    - Notice that the ‘dirty’ property is false since we haven’t changed anything yet, so the all the form field’s values are the same as when they were initialized.
    - Now, if we remove the ‘t’ in ‘grant’ in the name field, we get the following:
    - Webpage: Application

      Description automatically generated with medium confidence
    - Console: 
    - Notice the ‘dirty’ property is now false since the name field’s value of ‘gran’ is not the same as the initial value of ‘grant’.
    - Now, if we add the ‘t’ back in ‘grant’, we get the following:
    - Webpage: A picture containing application

      Description automatically generated
    - Console: 
    - Notice that the ‘dirty’ property is false again since all the form field’s values are the same as when they were initialized.
    - To disable the submit button right when the page loads, we could make use of the ‘dirty’ property as shown below.
    - 
    - Here, we disable the submit button if the form has been changed (!formik.dirty) or if the form is invalid (!formik.isValid).
    - However, the drawback to this example is if the initialValue of a form field is what the user wants to input, the submit button is still disabled. For example, the initial value of the name field was ‘grant’. If ‘grant’ is indeed the name I want to submit to the form, the form won’t let me submit since (!formik.dirty) will be true so the submit button will be disabled.
    - Thus, setting disabled = {!formik.dirty || !formik.isValid} is based on the assumption that on page load, the form state is initially invalid.
* Disabling the submit button when the form is being submitted
  + Suppose we have a form on the front end. When we click the form, an API call is made to the backend. When we are waiting for a response from the API call, it is important we disable the submit button. Otherwise, the user could submit multiple times.
  + To solve this issue, we have to look at the ‘isSubmitting’ property from the form level formik parameter object. This property will have a Boolean property which formik will set to true if a form submission has been attempted.
  + Thus, we can disable the submit button if ‘isSubmitting’ is true.
  + 
  + Now, if we run our application and then clear the console, then click the submit button, we get the following:
  + Text

    Description automatically generated with low confidence
  + Console output: Graphical user interface, text, application, email

    Description automatically generated
  + If we expand the first object that was logged and look at its ‘isSubmitting’ property, we see that it is true. If we expand the last object that was logged and look at its ‘isSubmitting’ property, we see that it is false.
  + Thus, the second we click on the submit button, the ‘isSubmitting’ property will be set to true and the submit button will get disabled. Then, formik will validate the inputs and if there are errors, formik will set the ‘isSubmitting’ property to be false, enabling the submit button.
  + What happens if the input is valid (there are no errors)? If the input is valid, the data gets submitted (as the onSubmit function is executed), but the ‘isSubmitting’ property will still be true. The ‘isSubmitting’ property is still true since formik does not know when your API will respond, so we have manually set ‘isSubmitting’ back to false.
  + To changing ‘isSubmitting’ to false, we to use the onSubmit method. The onSubmit method can take in a second parameter which we will call ‘onSubmitProps’
  + Text

    Description automatically generated
  + Now, if we run our code and enter valid inputs, then click submit, we get the following console output for the onSubmitProps:
  + Graphical user interface, text

    Description automatically generated
  + We see that the onSubmitProps is an object provided by Formik that has many helper methods. To modify the ‘isSubmitting’ property from the form level formik parameter object, we will use the onSubmitProps’ ‘setSubmitting’ method.
  + The setSubmitting method takes in a Boolean value as its parameter and will set the value of the ‘isSubmitting’ property to be whatever Boolean value that was passed in.
  + Text

    Description automatically generated
  + Now, if we run our code, enter valid input, clear the console, and then click submit. We get the following console output:
  + Graphical user interface, text, application, email

    Description automatically generated
  + In the first object that was logged, the value of the ‘isSubmitting’ property is true, disabling the submit button.
  + In the last object that was logged, the value of the ‘isSubmitting’ property is false, enabling the submit button.
  + In a real-world scenario, we would wait for an API response and then use the setSubmitting method after we get the response.

Load Saved Data

* We will learn how to load saved data back into our form
* If we’re working on small forms like user registration or login, we probably don’t need to load saved data.
* However, when we work with large forms that are broken into sections with several fields, we want to allow the user to save their progress so they can come back to complete the form some other time.
* To load saved data, we need first need to save/fetch the form data via an API. Then we need to fill that from data into our form.
* We will not use an API, but rather mock the loading of data via a saved data object as shown below:
* Text

  Description automatically generated
* This object above will represent the data the user has inputted so far
* Now, we can make a button that will fill the form with that data inside the savedValues object.
* Before we make the button, we will create a state variable that will represent the input in the form so far.
* 
* Next, we will create a button that will set the change the formValues state variable’s value from null to the savedValues object.
* Graphical user interface

  Description automatically generated with medium confidence
* Then we set the Formik component’s ‘initialValues’ prop value to be {formValues || initialValues}. We then add the ‘enableReinitialize’ prop to the Formik component as well. This ‘enableReinitialize’ prop controls whether Formik should reset the form if initialValues changes (using deep equality).
* Text

  Description automatically generated
* Now, when we load the application we get the following:
* Graphical user interface

  Description automatically generated
* The Formik component’s ‘initialValues’ prop has a value of {formValues || initialValues}. As of now, the formValues state variable’s value is null the expression { formValues || initialValues} returns the initialValues object. This is the input to the name field is ‘grant’.
* We can add type in whatever we want into the fields as shown below (we change the channel and phoneNumbers0 field).
* Graphical user interface, text, application

  Description automatically generated
* Now, make sure we are done interacting with the form (by clicking on anywhere that’s not an input field).
* Then click the ‘load saved data’ button. We now get the following.
* Graphical user interface, text, application

  Description automatically generated
* When we click submit, the ‘load saved data’ button’s onClick function get executed. This function sets the value of the formValues state variable from null to the savedValues object. Now if we consider the expression {formValues || initialValues}, it would return formValues as the formValues state variable’s value is no longer null. Just because this expression changed does not mean that the form fields get replaced by the data within the foomValues state variable. It is because we also passed in the enableReinitialize to the Formik component that causes the form fields to get replaced by the data within the formValues state variable. Recall that the enableReinitialize controls whether Formik should reset the form if the ‘initialValues’ prop’s value changes or not. And since the ‘initialValues’ prop’s value changed from initialValues to formValues (whose value is savedValues), the form is reset.
* In our example, the user had to manually click a button to load in the saved data that we stored in an object. In a real world application, the user would fill in fields which get stored to the database as they type in fields. Then when the user leaves and comes back and restarts the form, the saved data should be automatically loaded back when the form loads.

**Reset Form Data**

* There are two scenarios when resetting the form data. We could have a button that the user can click to reset the form data. We can also reset the form data upon form submission.
* First, we will learn how to create a button that the user can click to reset the form data.
* To do so, we will create a button and set it’s type to be ‘reset’.
* 
* When the user clicks this ‘reset’ button, the form will reset to its initial values.
* Second, we will learn how to reset the form data upon form submission.
* To reset the form data upon form submission, we will make use of the onSubmit method’s onSubmitProps parameter. This paramter has a ‘resetForm’ method which we can execute that will reset the form.
* Text

  Description automatically generated
* When the user submits the form and the data is valid, the form gets reset to its initial values.

**Reusable Formik Controls**

* We will build a set of reusable formik controls that can then be applied across a variety of forms such as registration, login, etc.
* We will create a FormikContainer component which is basically a Formik wrapper
* We will create a FormikControl component which will be capable of rendering the different types of form fields.
* The core form fields are:
  + Input
  + Text area
  + Select dropdown
  + Radio buttons
  + Check boxes
  + Date picker
* Once we have created all of the above, we will create a registration, login, and course enrollment form.
* Lastly we will learn how to use a UI component library with our reusable formik controls

**FormikContainer**

* We will start by making a new create-react-app and install formik and yup.
* Inside the src folder, we will create a new folder called components. Inside the components folder, we will create the following FormikContainer.js file
* Graphical user interface

  Description automatically generated
* Text

  Description automatically generated
* This component simply just a Formik component while using the renderProps method. The renderProps function returns a Form component with a submit button inside it.
* Make sure to include the FormikContainer inside the App component.

**FormikControl**

* We will create the FormikControl component that will decide which of the different core form fields to render based on its props.
* Create this component in FormikControl.js inside the components folder
* Text

  Description automatically generated
* Notice this FormikControl simply destructures the props’ ‘control’ property. Then, depending what the value of this ‘control’ property is, we will render different form field components that we have yet to create and yet to include in this FormikControl component.