**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

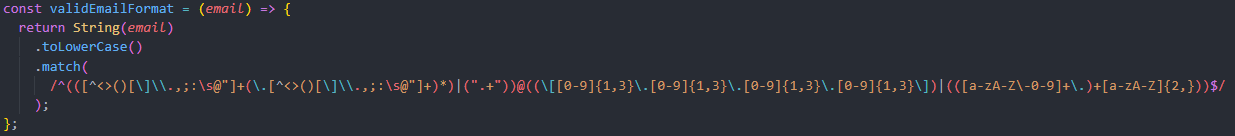
  Description automatically generated
* Enter the following input: Graphical user interface, text, application, email

  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.