**React - Formik Notes**

Introduction 2

A Simple Form 3

* Managing Form State 5
* Handling Form Submission 7
* Form Validation 8
* Displaying Error Messages 10
* Visited Fields 12
* Improving Validation UX 13
* Schema Validation with Yup 14

Reducing Boilerplate 15

* Fomik Components { Formik, Form, Field, ErrorMessage } 16

Field Component Revisited 17

ErrorMessage Revisited 19

Nested Objects 20

Formik Arrays 21

FieldArray Component 22

FastField Component 25

When does Validation Run 26

Field Level Validation 28

Manually Triggering Validation 29

Disabling the submit button 30

Load Saved Data 33

Reset Form Data 33

Reusable Formik Controls 34

**Introduction**

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

<https://formik.org/>

**A Simple Form**

Graphical user interface, text

Description automatically generated

First task is to build a standard html form and we will then see how Formik can help streamline the process and also add significant features and flexibility.

Text

Description automatically generated

So we’ve created a component called YoutubeForm and placed it in react-formik-demo/sec/components and now imported it into our App.js file.

A screenshot of a computer

Description automatically generated with medium confidence

Simple HTML form inside the JSX of our functional component.

Now, lets get Fomik into the equation. Firstly install it - yarn add formic

We need to use a special hook called useFormik which the library we’ve just installed provides.

This hook contains an object and it will help us to deal with the three points made above: form state, submission and validation/error messaging.

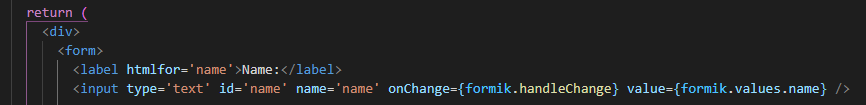
**Managing Form State**

When you type something into a form field, the value of that field changes. In React, if a value changes, then we need to incorporate a state variable. This is the form state which is an object that essentially keeps track of any changes in the variables within the form.

Diagram

Description automatically generated

So looking back at our code, this useFormik hook returns an object that we are storing within a constant called formik. This object contains the properties and methods that we need to manage our form state. The first step is to pass in a property called initialValues in the object that we’ve passed to the useFormik hook. This property is an object that contains ALL the initial values of our form input fields. It is worth noting here that these values correspond to the ‘name’ field in each of the input fields within our form. The next things we need to add are an onChange and value field on the input tag inside our form.

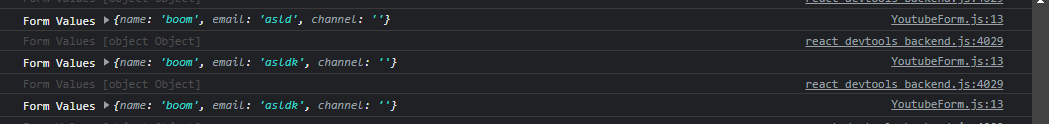


You can see here that the onChange and value fields are controlled by formik which as a result of being incorporated here will now track any changes to the specific value of ‘name’ if it changes:

A screenshot of a computer

Description automatically generated with medium confidence

Now that we’ve given formik access to our form we can see the values of our formik constant in a log statement.



The log shows the formik.values object. The key in the object corresponds to the name and the value is what you’ve specified by typing something in. It’s clear than that this object ALWAYS reflects the current state of the form.

You can load in an initial value of course:

Text

Description automatically generated

A picture containing table

Description automatically generated

**Handling Form Submission**

Using Formik this is a simple 2 step process:

1. We need to specific the onSubmit handler on the form tag. onSubmit={formik.handleSubmit}Text

   Description automatically generated
2. Now we need to revisit the object that we passed into the useFormik hook. As well as the initialValues property, which we already discussed above, there is another called onSubmit. This is a method that automatically receives the current state of the form as it’s argument.

Text

Description automatically generated

We’re just producing a log statement here but now that we have grabbed the form data, we can export it with an api to our database or do anything else we like with it!

**Form Validation**

Diagram, timeline

Description automatically generated

Here we will add a required validation rule to all the fields and another rule for the email to check it is a valid email format.

Formik allows you to create a validation function which is assigned to a property called ‘validate’ in the object we pass to the useFormik hook. Here we have an arrow function which taking in the values object. The first thing to note about this function is that for Formik to work as intended, this function must return an object which we will create inside the function and return at the bottom called errors. The keys within this errors object should correspond to the keys inside the values object which is why I have written both below in green.

Text

Description automatically generated

Now we can see that there is some repeated code here so some refactoring will be great.

To make the code a little prettier, we can set the values within initialValues outside of the useFormik hook as a constant.

Text

Description automatically generated

We can do the same with the onSubmit method by initializing the function outside of the useFormik hook and again the same with validate.

Text

Description automatically generated

Once this has been done then we can make the hook itself much cleaner and clearer.

Text

Description automatically generated

Nice.

**Displaying Error Messages**

A screenshot of a computer

Description automatically generated with medium confidence

In this print statement, we have already seen that formik.values will return the initial values on a page refresh. If we change log to formik.errors we will get an object returned which is empty at this stage. I have initial values set for all three fields here so I will remove the value I have preset for the name field and lets see what happens:

Graphical user interface, application

Description automatically generated

The first log is from the page refresh and the second was as soon as I deleted the initial value inside the name field. Notice that we’re also getting the error for an invalid email format as ‘Email here’ is not a valid email address…

Graphical user interface, text

Description automatically generated

Here is the log statement if I remove everything from all the fields. We’re not getting the invalid email now because of the order of the if statement we have in our validate function.

So what we can see here is that just as the values of the values object are populated on any changes made to the form, so to is the errors object. It is run every time there is a change within the form. Now we simply access this error object and it’s properties.

At the bottom of each section of the form we need to add some conditional html to check if there is an error message and then render a div which will include the error message if there is an error. Text

Description automatically generated

If the error exists then render this div, otherwise do nothing.

Now to make this a little bit prettier…

At the moment, in our app.css file, the textarea has a margin at the bottom which is separating this error message from the box it corresponds to so lets comment that out. Now there’s no margin between each field so lets create a div tag container for each of the fields which we’ll call form-control and then add the margin back into the css for this class. While we’re at it, lets create an error class as well and change the font colour to red and reduce the font size.

A screenshot of a computer

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

Description automatically generatedNice.

**Visited Fields**

At the moment, the errors function runs on every single key stroke and actively runs on all the individual values within the fields. The errors object also contains all the validation errors at any given moment. This is because in our component we are checking if an error exists and immediately displaying this as a console message. If there happened to be 20 fields we wouldn’t want the user to receive error messages the moment they interacted with one of those fields. Most of the time we only want to check for errors in fields that the user has interacted with. Formik can manage this process. We can add the onBlur prop to the form input element if we need to track if an input field has been visited with by the user. Formik stores this information in an object called touched! Similar to the formik.values and formik.errors we now have formik.touched.

Now we modify our log statement to show the visited fields:

Graphical user interface

Description automatically generated

So we can see that on a refresh we have an empty object to begin with and then as I click into the fields one at a time, they have key and value pairs where the key relates to the field I clicked on and the value is a Boolean showing true if the user has been there.

**Improving Validation UX**

We’ve seen above how we can keep track of whether a user has interacted with a field within the form. Now we can take that information and add it into our conditional rendering statement for each form input field.

Text

Description automatically generated

Text

Description automatically generated

Graphical user interface, text

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Now we can see from the above that if I click on the first field, remove anything in there and click away, we will get an error message JUST for this first field despite the fact that ‘email here’ is not a valid email.

**Schema Validation with Yup**

It is worthwhile/better to go with a validation schema than write your own validation schema as we have done above. To this end, we’ll utilise a simple schema called Yup.

<https://github.com/jquense/yup>

So now with this installed we can add a validation schema which we do in the form of a constant called validationSchema which is a yup object. Inside the object we need to deal with the three fields and once we’ve done this we include it in our useFormik hook and comment out the old ‘home made’ validation schema.

Text

Description automatically generated

Don’t forget to import it as well!

A screenshot of a computer

Description automatically generated with medium confidence

**Reducing Boilerplate**

Now we’re going to look over the code we’ve created so far and find ways we can reduce it.

Let’s create a copy of our component so we can use it as a bit of a before and after comparison when we’ve finished refactoring.

This is a small change but it incorporates a special method available inside Formik called getFieldProps which allows us to remove some of the boilerplate code that we have been using for each of the fields.

Text

Description automatically generated

Into this…

Text

Description automatically generated

The object brought into the getFieldProps is the name value for the field.

There are a great many more input variables available in Formik which is why we add the spread operator at the beginning to assign this email prop to all the variables. Had we made a list with 10 or so variables and had to copy them for each of the input fields it would be even more obvious that this process would be necessary to reduce copied code.

**Formik Components**

The above was an example of just one of the many components within Formik that exist to make our lives easier. Here we’ll take a look at for of them.

**Formik**

This is a replacement for the useFormik Hook significantly reducing the amount of code required. Firstly lets change the import from { useFormik } to just { Formik}. This also means removing the call to the useFormik hook! Now, we wrap our jsx in a Formik tag. Last step is to pass in some props into this new tag. These are the same as were specified when calling the useFormik hook. Text

Description automatically generated

As you can see the code is broken now but never fear - it will be fixed as we work through the next three parts. This part doesn’t really reduce the number of lines of code we have to write but it is a requirement of the next two components we will cover which DO reduce the number of lines of code we need to write.

**Form**

To introduce this component, first we import it. Then we need to replace the html form element with this form component. Third and final thing to do is to remove the onSubmit prop from the newly changed Form component. We are able to do this because the Form component is a small wrapper around the html form element which automatically hooks into Formik’s handleSubmit method. In other words, the Form component automatically links the onSubmit method to the actual form’s submit event.

**Field**

As before, import Field from Formik. Secondly, replace each input html field with the new <Field> tags. Then remove the getFieldProps hook from each of these fields. This is essentially what we are replacing by adding this Field Component.

**ErrorMessage**

As with the above, there is repetition in the code here and Formik wants to help us out in these scenarios with useful components to reduce the lines of code we need to write. So we first import the ErrorMessage component. Then, delete the sections of code within each form element that deal with error messaging and replace them with an <ErrorMessage> tag. Include as a prop the name which corresponds to the name value of each input field. There we have it.

**Field Revisited**

By default, the Field component renders an html input element. Behind the scenes, this Field hooks up the input element to Formik - in our example above that would mean that it hooks up ‘handleChange’, ‘handleBlur’ and the value of the form field as well. There are some additional points about the Field component that are worth visiting.

The first point to cover here is that the Field component will pass through any additional props that you specify.

Text

Description automatically generated

If we inspect this element in the browser we can see that all the props have been brought through into the html for the page.

Graphical user interface, text, application

Description automatically generated

The second point to cover is that the Field component can render different types of element, not just an input field. Lets say we want to add a text area field such as a comments box…

Text

Description automatically generated

Here we see the ‘as’ prop making all the difference to this input area.

The third point has to do with the renderProps pattern. Basically this is an alternative process for rending a field. It takes the form of an arrow function within the Form component. The function returns a props object which we destructure into the three parts, field, form and meta. As it’s a function, we return something. What we return is a div tag with the input field inside it. The input field will have an id prop, a type (to help with css) and ALSO the field props which is spread so it includes all the methods and components within the field prop.

A screenshot of a computer

Description automatically generated with medium confidence

Also included here is an error message which includes a condition based on the meta props. We then render a div tag with the error message or return null. This is basically the same as the ErrorMessage field.

**ErrorMessage Revisited**

For a quick recap, we’ve covered that the errorMessage component receives a prop called name which is equal to the value of the field for which it is applied. It will display an error message if the field has been visited and if there is an error message set to display. At the moment though, the error message is just ‘free floating’ and not wrapped in any html tag…

Graphical user interface, text, application

Description automatically generated

Here is the error message within the browser console after being inspected.

Adding a prop to the errorMessage component called ‘component’ allows you to wrap this error message into any html tag. As below:

In VS code:

Text

Description automatically generated

Resulting render in browser:

Graphical user interface, text

Description automatically generated

It is also possible to set it to a custom React component. Let’s create a component that renders text in the colour red and pass that as a prop to our errorMessage component…

Simple enough:

Text

Description automatically generated with medium confidence

Here we’re using the className which is in our css specifying that text should be red.

Back in our form component we add this TextError component to the props for our errorMessage component.



Now the error will be rendered in red.

This process can also be done in the same way that we created the Field component in our address field - using the render props pattern. Back in our code, lets apply this to the email error. Firstly we create a closing tag for the ErrorMessage tag for email.

**Nested Objects in Formik**

Text

Description automatically generatedSometimes it’s important to group fields together within a form. It may be that an api only accepts information in a certain way or the database only stores information in a certain way. Either way, we need to be able to group things together - nested objects is the solution. In the example we will do here with out youtube form is that we will assume the for wants to know what the twitter and facebook profiles of the user filling in the form are. These two bits of data are related and need to be grouped together. Firstly in our initialValues field we will be adding a new key called social whose value is an object. The object will have two key/value pairs which are facebook and twitter. We’ll give them an initial value of an empty string. The next job is to create the html form element to display this. This is pretty standard as we have been doing for the other fields UNTIL we get to the props for the Field tag. In here, the name needs to correspond to the nested object so it can’t be name or email or channel etc it has to be social.facebook or social.twitter.

Text

Description automatically generated

A quick glance at the console after writing some data into our form and submitting it yields the following:

Text

Description automatically generated

We can see here then that the social element is an object with two nested values.

2 simple steps for this one.

**Arrays**

OK so lets say that we want a mobile phone number and a landline number for a user but we don’t need any clear distinction between the two in terms of storing the data, we just need to store them both. Here we can use an array. So first of all we add to the initial values once again and this time we’ve got phoneNumbers which will be an array of strings.

Graphical user interface

Description automatically generated with medium confidence

As with the nested objects above, where the main change comes into play here is in the name attribute. Here we have phoneNumbers[0] and phoneNumbers[1] corresponding to the two places within the array that we have set up in the initialValues.

Text

Description automatically generated

There’s a pattern emerging here 😊

**FieldArray Component**

This component has many uses but it is most useful in a commonly occurring scenario which is dynamic form controls. It helps with array and list manipulations. Let’s use an example to see some of this in action.

Just above we added an array to our form, phoneNumbers, with two empty strings in its array. In our code we produced fields that would focus on the name prop that we gave them which were the two positions within the array, [0] and [1]. It is far more common, however, for the collection of multiple numbers or addresses for example to be handled through dynamic form controls. What this essentially means is that the user will be given one field to provide one number/address/whatever and if they wish to add more then they can click to add another field. The input data will then be managed as a form array. Lets change our current set up to make a dynamic form for the numbers.

First task is to import FieldArray from Formik.

Now, as with all the other examples above, we need to add a new entry into the initial values component. Lets call it phNumbers to differentiate between the phoneNumbers already in the list. It’s going to be an array with only one string inside. This is because we’re only asking for one thing to begin with.

Next we add the jsx which is where things get a bit more funky.

1. Create the div tag with its class and then inside create the label tag with text ‘list of phone numbers’
2. Now we add the FieldArray tag and give it the prop we’ve created above in our initalValues component.
3. This FieldArray is going to have the render props pattern which means there will be an object with an arrow function taking in props which we will call fieldArrayProps. Before we go any further, let’s see what these props actually entail…

Text

Description automatically generated

Text

Description automatically generated

OK so above, first we have the code. Then what follows is a shot of the console statement. We can see that the fieldArrayProps is an object which contains a large number of properties and methods which we can use to manipulate arrays - all the classic stuff is there. Also there is the form object which essentially houses all the methods we would need for Formik to control our form. In our example though we will focus just on push and remove.

Also inside this object, inside the form nested object, is something that we need to make all this work:

Text

Description automatically generated

So we have the phNumbers array object inside our values object. We need to use this to keep track of what we’ve got in our field. Time for step 4.

1. As stated above, we need the push and remove methods and the phNumbers property from deep inside our FieldArray component. So we need to destructure these into our render props pattern. We also need the values object from inside form so lets assign that to a constant as well. We ALSO need the phNumbers object which we can now extract from our values object we’ve just created. Now we have all the information we require to make our field and get the functions working.
2. Next job is to work on the jsx. We need to return a div tag but inside it we need to have a function that will iterate over our array and for each value in the array we need to do some jsx. So we’re doing a foreach kind of thing here so will need to map over each of the items in the array and for each we need to display the number and also some buttons to add more fields or remove the current field. A bit confusing maybe so lets go over it…Text

   Description automatically generated

So we’ve got our return statement which is a div in itself. Then we have our function because we need to iterate over the values in our array. Inside our function we access the phNumbers array which we’ve extracted from values.phNumbers above in a constant. We need to iterate over any numbers within this array using the map method. We’re going to call each value in the array phNumber and we’ll need the index of that number too so these are our two props for our map method. For each of the numbers in the array that we’re mapping over we need to return some jsx. This jsx needs to be an input field so we can add a number which is why we then use the <Field> tag to render the jsx appropriately. The name prop for this field needs to be a template literal bringing in the bringing in the index of the currently iterated over value of the array - essentially this is a dynamically indexed Field tag name prop. This will be accessed later. After that, we just need two buttons, one to remove this index from the array and one to push a new empty array object. These buttons will need to have functions as their onClick handlers.

Table

Description automatically generated

It's working as expected. Interestingly though we’ve not dealt with what would happen if the user deleted all of their numbers. There would be no array and as a result nothing would be displayed. We wouldn’t want that so lets Deal With It. A simple condition would suffice.

Graphical user interface, text

Description automatically generated

A picture containing table

Description automatically generated

Done.

**FastField Component**

This component is principally designed for performance optimisation. The documentation recommends this for forms with over 30 fields or for fields that have very complex validation requirements. There are some words of warning in the documentation as well. <https://formik.org/>

OK so lets look at the field form and see how it renders in our browser by adding a simple log statement to the render props pattern in our address field. When we look at the browser we can see that on page refresh, our statement is logged twice. It is also logged twice again every time a change is made to any of the form fields, not just the address one.

NOW, let’s import FastField and change the Field component tag in our address form to FastField. Now if we load the browser and refresh we see that the render statement again has loaded twice. HOWEVER, if we type into any other field within the form we do not see any new statements being logged. If we then type directly into the address field we see that these statements reappear. This is the difference between Field and FastField components. FastField is an optimised version of the Field component which internally implements the shouldComponentUpdate lifecycle method to block all additional re-renders unless there are direct updates to the FastField form control itself. I’m not sure of a usecase right now and the Formik example (<https://formik.org/docs/api/fastfield>) is a little complicated at the moment. Probably something to come back to at a later date.

Straight from the docs…

Graphical user interface, text, application, email

Description automatically generated

**When does Validation run?**

What we know so far is that once the validation rules are run, Formik auto-populates the formic.errors object with the error messages. Lets see how this formik.errors object behaves. Firstly lets look at one of our fields which incorporates the render-props pattern. We need to access the form object to retrieve our errors object so lets create a log statement with form.errors.

A screenshot of a computer

Description automatically generated with low confidence

First Scenario.

Lets try to input some data into the channel field and see what happens to the errors object.

Before we do this we can see the log statement printing our an empty forms object.

A screenshot of a computer

Description automatically generated with medium confidence

Now we type anything into the channel field.

Graphical user interface, text

Description automatically generated

So this shows that Formik is running validation after any change within the form.

Second Scenario.

Click inside the channel field and then simply click out of that field…

A screenshot of a computer

Description automatically generated with medium confidence

Formik also runs validation after any blur event in the form. (If you blur out from a form field)

Third Scenario.

Without interacting with any of the form fields, directly click on the submission button.

Graphical user interface, text, application

Description automatically generated

Same again. So whenever a submission attempt is made, formik will run validation. The onSubmit handler does not get executed if there are any errors produced by the validation process. This means that I don’t have to worry about manually handling form submission and submitting only if the validation is successful. Formik will do this for me - less code.

So we have these three different scenarios where validation occurs: on change, on blur and on submission. It is possible to stop the validation process though and this can be useful in some cases where you don’t, say, need the validation to run on changes in a field and perhaps only when the form is submitted. This is easily taken care of in formik with the incorporation of a function called validateOnChange.

Text

Description automatically generated

Default is set to true obvs.

Graphical user interface, text, application

Description automatically generated

Now with this function in place we see that changing a field does not cause the errors object to populated but look, down at the bottom we have it populating - this is because the onBlur is still initiating the validation process. Funnily enough there is a function to stop that from happening too.

Graphical user interface, application

Description automatically generated

Now we have clicked into the channel field, added some data and then clicked out of that field and still the errors object has not been populated.

**Field Level Validation**

Text

Description automatically generated

As we can see here, we have incorporated a validation schema into our form right at the top of the Formik tree and set validation rules for three of the input fields. It is also possible to run validation in a field at the field level rather than at the top of the formik tree. Firstly lets simply add a validation rule to the comments field using the pre-existing schema.

A screenshot of a computer

Description automatically generated with medium confidence



Yep, that works fine. Now lets try to do the same but on the field itself.

So firstly we need to create a new arrow function which takes in an object called value. This value comes from the field object and is populated with whatever is put into the field.

Text

Description automatically generated

In order for this function to run we need to add an attribute object to our Field tag for the comments field in our form.

Text

Description automatically generated

We’ve also added the component ‘TextError’ to make the error red when it appears on the screen.

A picture containing graphical user interface

Description automatically generated

Nice.

Though to get this to work we did have to mute our functions for validatOnChange and validateOnBlur.

**Manually Triggering Validation**

This has to do with wrapping the entire form within the renderprops pattern which gives us top level access to all the methods available in the form object which we’ve used in our address and phone numbers fields. This can be handy for a number of reasons, not just manually triggering validation but I think this is out of the scope of this project.

**Disabling the Submit Button**

Yes this is possible too.

Graphical user interface, text, application

Description automatically generated

**Scenario 1**

Right, we missed the ‘manually triggering validation’ step earlier but we will be requiring some of the props from the formik object that was created in that step by wrapping our entire form in a render props pattern.

Text

Description automatically generated

We’ve incorporated a little log statement to look at this object in detail on the browser console. The first method we will be focusing on is the ‘isValid’ one.

Graphical user interface, text, application

Description automatically generated

This is a read only property that is set to true if the errors object is empty.

Clicking inside the channel field and clicking out again will trigger a validation call and result in the error object being populated which will cause the isValid prop to be false.

Text

Description automatically generated

Now we can capture this and apply it to our submit button to deactivate it.



Diagram

Description automatically generated with medium confidence

Nice.

Interestingly, when you initially load the page, the submit button is accessible because at this stage the errors object has not yet been populated with anything. There is another prop within the formik object that we can use should we wish to make the button unavailable on the page’s initial load. This is called ‘dirty’ and is set to false when the page loads. It is a prop that will change to true as soon as a field has been altered (so an onChange event has happened - NOT on a blur). There is a slight drawback to this in that the dirty prop works on the assumption that when the page loads, without the user changing any of the form field values, the form’s state is always invalid. If you know for certain that the user will input values into your form and that those values are not going to be the same as the initial values of the form then you can stick to this option. I think for my usecase, this would work well or I could just not include this and have a submit button not greyed out when the page loads… probably looks better that way anyway. The form will not submit if the user hasn’t entered any information because of the validation rules I have already incorporated so it’s not like they would get a serverside error popping up from submitting the form too soon.

**Scenario 2**

This is quite an important addition to a form. Imagine you’re buying something from an ecommerce site and you click the purchase button. In the background an api call is being made to the server and while that’s happening you might be thinking ‘maybe that didn’t work’ and click the button again, then again-again. Or you’re on a registration form and you’ve entered your credentials and submitted the registration form and are waiting for something to happen so click on the submit button again. All of a sudden you’ve got two or more sets of the exact same data being sent to the DB. PROBLEMS!

Graphical user interface, application

Description automatically generated with medium confidence

So, we disable the button until this background process has completed.

Graphical user interface, application

Description automatically generated

Enter the isSubmitting prop. This is another read-only prop that is set to true if a form submission has been attempted. So all we need to do is check if this is true and if so disable the submit button.



Simple enough. It works too!

But it worked because there were errors in the form. What happens if we fill out the form and submit it? This time the submit button stayed disabled even though we have a submitted form. This is an intended response and is because formik is waiting for a response from something like an api to say that the form submitted and a command to set the isSubmitting prop back to false needs to run. All this happens inside the onSubmit function further up this tree. At this point we can see that the onSubmit function only has one prop but it can, in fact, have more! One of which is called onSubmitProps. Lets call it to the console and see what it is.

Text

Description automatically generated

So it’s an object with a number of methods/functions. One of these is setSubmitting. This is the function we need to set our isSubmitting back to false which will then turn our submit button back on for us. Obviously in our own website we will have a great deal more code within the onSubmit function which will include an CRUD calls to our DB and an API response which, when successful we then would set our submitting prop back to false.

Now we re-introduce the validation prop from part one of this disabling the submit button section so we’re also checking if the form isValid at any point and if not then the submit button is greyed out.

Nice user experience.

**Load Saved Data**

This process is not really relevant to my use-case and applies more to forms when a user would like to perhaps go away and do something else then come back to a form half way through completing it. Perhaps for a long questionnaire or an online exam/tutorial. Not really useful for a registration page.

**Reset Form Data**

As above, this functionality is not really required for a simple registration form. Simply adding a reset button and setting its type attribute to reset will do this, resetting to the initial values. Also check out the onSubmitProps methods for a way of resetting the form after submission. Very simple.

**Reusable Formik Controls**

Now lets get into the nitty gritty and actually create something that we can use within our application but also learn some techniques that will help going forward with any other forms I might need to create in the future.

**Graphical user interface

Description automatically generated with medium confidence**