**Form Validation**

React Hook Form makes simple form validation very straightforward by aligning with ["browser native validation"](https://developer.mozilla.org/en-US/docs/Learn/Forms/Form_validation) attributes. As such, the following basic validation rules are provided:

* **required:** Indicates that the input must have a value before the form can be submitted.
* **min:** The minimum value to accept for this input (number)
* **max:** The maximum value to accept for the input (number)
* **minLength:** The minimum length of the value to accept for the input
* **maxLength:** The maximum length of the value to accept for the input
* **pattern:** The regex pattern for the input.

**Adding Validation Rules to "register"**[**​**](https://webprogrammingforappsandservices.sdds.ca/Forms-Introduction/validation#adding-validation-rules-to-register)

In addition to "registering" our form controls, the ["register"](https://react-hook-form.com/api/useform/register) function also accepts a second "options" parameter to configure how the control behaves, including setting validation rules.

For example, we will start with the following simple form:

import { useForm } from 'react-hook-form';  
import { useEffect } from 'react';  
  
export default function FormWithValidation() {  
  
 const { register, handleSubmit, setValue } = useForm({  
 defaultValues: {  
 firstName: "",  
 lastName: "",  
 age: 0  
 }  
 });  
  
 useEffect(() => {  
 let data = {  
 firstName: "Homer",  
 lastName: "Simpson",  
 age: 42  
 }  
  
 *// set the values of each form field to match "data"*  
 for (const prop in data) {  
 setValue(prop, data[prop]);  
 }  
 }, []);  
  
 function submitForm(data) {  
 console.log(data);  
 }  
  
 return (  
 <form onSubmit={handleSubmit(submitForm)}>  
 First Name: <br />  
 <input {...register("firstName")} /><br /><br />  
  
 Last Name: <br />  
 <input {...register("lastName")} /><br /><br />  
  
 Age: <br />  
 <input type="number" {...register("age")} /><br /><br />  
  
 <button type="submit">Update User</button>  
 </form>  
 );  
}

Here, we ask the user to modify their first name, last name and age, starting from initial values obtained in the "useEffect" hook. At the moment, there are no restrictions on what the user may enter, apart from the restriction built in to the input type ["number"](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input/number).

If we wish to add specific validation rules for each form field individually, we can specify each validation rule as a *property* of the *RegisterOptions* parameter of the ["register"](https://react-hook-form.com/api/useform/register) method. For example, we can modify the 3 form fields above to use specific validation rules such as "required", "maxLength", "pattern", "min", "max", etc.

<form onSubmit={handleSubmit(submitForm)}>  
 First Name: <br />  
 <input {...register("firstName", { required: true, maxLength: 20 })} /><br /><br />  
  
 Last Name: <br />  
 <input {...register("lastName", { pattern: /^[A-Za-z]+$/i })} /><br /><br />  
  
 Age: <br />  
 <input type="number" {...register("age", { min: 18, max: 99 })} /><br /><br />  
  
 <button type="submit">Update User</button>  
 </form>

AFter this change has been made, the form should still appear the same, however if we try to break one of the validation rules (ie: removing the first name value) before submitting the form, we will notice that:

1. The (first) form field that is in violation of a validation rule is focused
2. The "submitForm" function does not run

**Custom Validation Rules**[**​**](https://webprogrammingforappsandservices.sdds.ca/Forms-Introduction/validation#custom-validation-rules)

In addition to the above "native validation", React Hook Form also allows us to create our own validation rules by specifying one or more callback functions as properties within a "validate" parameter to *RegisterOptions*. As a simple example, say we wish to ensure that we are only accepting "Age" values that are even, in the above form. We can add the following **custom** "onlyEven" validation rule to our "nunber" field:

<input type="number" {...register("age", { min: 18, max: 99, validate: { onlyEven: v => v % 2 == 0 } })} /><br /><br />

**Showing Errors**[**​**](https://webprogrammingforappsandservices.sdds.ca/Forms-Introduction/validation#showing-errors)

While restricting the form submission and automatically focusing the problematic field is an effective way to perform client side validation, we are missing an important piece: showing a *description* of the error to the user, so that they can correct the mistake.

In order to actually show the errors, we must first add the "errors" object from ["formState"](https://react-hook-form.com/api/useform/formstate), to give us access to any errors in the form:

const { register, handleSubmit, setValue, formState: { errors } } = useForm({...});

This will give us access to the "errors" object, which will contain each form field that contains an error, as a property. Additionally, each specific validation error will also be identified for each form field using a "type" property. If there are no invalid fields, this object will be empty, ie "{}".

Using this, we can easily inspect a specific form control to see if it contains any errors and if so, which rules have been violated. For example, if we wish to show errors for the "firstName" field, we could use the following code:

<input {...register("firstName", { required: true, maxLength: 20 })} />  
{errors.firstName?.type === "required" && <span><br />First Name is required</span>}  
{errors.firstName?.type === "maxLength" && <span><br />First Name Cannot contain more than 20 characters</span>}  
<br /><br />

Notice how we use the ["optional chaining operator"](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Optional_chaining) before checking the "type" of the error message. This ensures that we do not receive the following error if the form field is currently valid (ie: no errors):

TypeError: Cannot read properties of undefined (reading 'type')

Also, if you try running the example again, you should see that the error messages show up only after the form is first submitted. After this first submit, the errors are shown / hidden as the user modifies the form. This lets the user know immediately if they have corrected the error.

**NOTE**: In addition to the "errors" object, you can include additional objects such as *"dirtyFields"* and *"touchedFields"* to monitor which fields have been modified and visited.

**Highlighting Fields**[**​**](https://webprogrammingforappsandservices.sdds.ca/Forms-Introduction/validation#highlighting-fields)

Given that the "errors" object contains property names of all form fields that are currently in violation of a validation rule, it is a simple task to conditionally add a CSS class to a form field that is in error. For example, if we had the class "inputError", we could conditionally add it to the field using the code:

className={errors.firstName && "inputError"}

Here, we check if the "errors" object contains a *firstName* property, and if it does, add the "inputError" class.

**Disabling the "submit" Button**[**​**](https://webprogrammingforappsandservices.sdds.ca/Forms-Introduction/validation#disabling-the-submit-button)

Another interesting UI improvement that can be made using "errors" is conditionally disabling the "submit" button if the form is "in error" (has one or more fields with validation errors). This can be done by counting how many properties exist on the "errors" object using "[Object.keys()"](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys" \t "_blank), since only form fields that are in error should have a corresponding property within the errors object, ie:

<button type="submit" disabled={Object.keys(errors).length > 0}>Update User</button>