UsfForm Hook in React JS

`useForm` is a custom hook for managing forms with ease. It takes one object as **optional** argument. The following example demonstrates all of its properties along with their default values.

How to Create Forms in React using react-hook-form

Creating forms in React is a complex task. It involves handling all the input states and their changes and validating that input when the form gets submitted.

For simple forms, things are generally manageable. But as your form gets more complex and you need to add various validations, it becomes a complicated task.

So instead of manually writing all of the code and handling complex forms with validation logic, we can use the most popular React library for this, react-hook-form.

It's the most popular React library for creating forms compared to formik, react final form, and others, and I use it for all my client projects.

Why the react-hook-form Library is the Most Popular Form Library in React

Following are some of the reasons why `react-hook-form` is a popular choice for creating React forms.

- The number of re-renders in the application is smaller compared to the alternatives because it uses refs instead of state.
- The amount of code you have to write is less as compared to `formik`, `react-final-form` and other alternatives.
- `react-hook-form` integrates well with the `yup` library for schema validation so you can combine your own validation schemas.
- Mounting time is shorter compared to other alternatives.

How to Create a Form Without Using a Library

Before creating a form using the `react-hook-form` library, let's create a simple form without using any library.

Take a look at the below code:

```
import React, { useState } from "react";
import "./styles.css";

export default function App() {
  const [state, setState] = useState({
    email: "",
    password: ""
  });

  const handleInputChange = (event) => {
    const { name, value } = event.target;
    setState((prevProps) => ({
      ...prevProps,
      [name]: value
    }));
  };

  const handleSubmit = (event) => {
    event.preventDefault();
    console.log(state);
  };

  return (
    <div className="App">
      <form onSubmit={handleSubmit}>
        <div className="form-control">
          <label>Email</label>
          <input type="text"
            name="email"
            value={state.email}
            onChange={handleInputChange}
          />
        </div>
        <div className="form-control">
          <label>Password</label>
          <input
            type="password"
            name="password"
            value={state.password}
            onChange={handleInputChange}
          />
        </div>
        <div className="form-control">
          <label></label>
          <button type="submit">Login</button>
        </div>
      </form>
    </div>
  );
}
```

In the above code, we have only two input fields, namely `email` and `password` and a submit button.

Each input field has a `value` and `onChange` handler added so we can update the state based on the user's input.

Also, we have added a `handleSubmit` method which displays the data entered in the form to the console.

This looks fine. But what if we need to add validations like required field validation, minimum length validation, password validation, email field validation and also display the corresponding error messages?

The code will become more complex and lengthy as the number of input fields and their validations increases.

How to Install react-hook-form

Displaying forms is a very common requirement in any application.

So let’s learn why and how to use react-hook-form. For that, we'll create a new React application.

Create a new React project by running the following command from the terminal:

```
create-react-app demo-react-hook-form
```

Once the project is created, delete all files from the `src` folder and create new `index.js` and `styles.css` files inside the `src` folder.

To install the `react-hook-form` library, execute the following command from the terminal:

```
npm install react-hook-form
```

```
import React from "react";
import "./styles.css";

export default function App() {
  return (
    <div className="App">
      <form>
        <div className="form-control">
          <label>Email</label>
          <input type="text" name="email" />
        </div>
        <div className="form-control">
          <label>Password</label>
          <input type="password" name="password" />
        </div>
        <div className="form-control">
          <label></label>
          <button type="submit">Login</button>
        </div>
      </form>
    </div>
  );
}
```

Here, we have just added the email and password fields to the form.

How to Create a Basic Form with react-hook-form

The `react-hook-form` library provides a `useForm` hook which we can use to work with forms.

Import the `useForm` hook like this:

```
import { useForm } from 'react-hook-form';
```

You can use the `useForm` hook like this:

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

Here,

- `register` is a function provided by the `useForm` hook. We can assign it to each input field so that the `react-hook-form` can track the changes for the input field value
- `handleSubmit` is the function we can call when the form is submitted
- `errors` is a nested property in the `formState` object which will contain the validation errors, if any

Now, replace the contents of the `App.js` file with the following code:

```
import React from "react";
import { useForm } from "react-hook-form";
import "./styles.css";

export default function App() {
  const {
    register,
    handleSubmit,
    formState: { errors }
  } = useForm();

  const onSubmit = (data) => {
    console.log(data);
  };

  return (
    <div className="App">
      <form onSubmit={handleSubmit(onSubmit)}>
        <div className="form-control">
          <label>Email</label>
          <input type="text" name="email" {...register("email")} />
        </div>
        <div className="form-control">
          <label>Password</label>
          <input type="password" name="password" {...register("password")} />
        </div>
        <div className="form-control">
          <label></label>
          <button type="submit">Login</button>
        </div>
      </form>
    </div>
  );
}
```

In the above code, we have added a register function to each input field that we got from the `useForm` hook by passing a unique name to each `register` function like this:

```
{...register("email")}
```

We're using the spread operator so `react-hook-form` will spread out all the required event handlers like `onChange`, `onBlur`, and other props for that input field.

If you add a `console.log({ ...register("email") });` inside the component, you will see what it returns as can be seen below:

We also added the `onSubmit` function which is passed to the `handleSubmit` method like this:

```
<form onSubmit={handleSubmit(onSubmit)}>
```

Note that, you need to pass a unique name to the `register` function added for each input field so `react-hook-form` can track the changing data.

When we submit the form, the `handleSubmit` function will handle the form submission. It will send the user entered data to the `onSubmit` function where we're logging the user data to the console.

```
const onSubmit = (data) => {
 console.log(data);
};
```

Now, start the application by running the `npm start`

As you can see, when we submit the form, the details entered by the user are displayed in the console.

This is because we don't have to add the `value` and `onChange` handler for each input field and there is no need to manage the application state ourselves.

How to Add Validations to the Form

Now, let’s add the required field and minimum length validation to the input fields.

To add validation we can pass an object to the `register` function as a second parameter like this:

```
<input
  type="text"
  name="email"
  {...register("email", {
    required: true
  })}
/>

<input
  type="password"
  name="password"
  {...register("password", {
    required: true,
    minLength: 6
  })}
/>
```

Here, for the email field, we're specifying required field validation. For the password field we're specifying the required field and minimum 6 character length validation.

When the validation fails, the `errors` object coming from the `useForm` hook will be populated with the fields for which the validation failed.

So we will use that `errors` object to display custom error messages.

```
import React from "react";
import { useForm } from "react-hook-form";
import "./styles.css";

export default function App() {
  const {
    register,
    handleSubmit,
    formState: { errors }
  } = useForm();

  const onSubmit = (data) => {
    console.log(data);
  };

  return (
    <div className="App">
      <form onSubmit={handleSubmit(onSubmit)}>
        <div className="form-control">
          <label>Email</label>
          <input
            type="text"
            name="email"
            {...register("email", {
              required: true,
              pattern: /^[^@ ]+@[^@ ]+\.[^@ .]{2,}$/
            })}
          />
          {errors.email && errors.email.type === "required" && (
            <p className="errorMsg">Email is required.</p>
          )}
          {errors.email && errors.email.type === "pattern" && (
            <p className="errorMsg">Email is not valid.</p>
          )}
        </div>
        <div className="form-control">
          <label>Password</label>
          <input
            type="password"
            name="password"
            {...register("password", {
              required: true,
              minLength: 6
            })}
          />
          {errors.password && errors.password.type === "required" && (
            <p className="errorMsg">Password is required.</p>
          )}
          {errors.password && errors.password.type === "minLength" && (
            <p className="errorMsg">
              Password should be at-least 6 characters.
            </p>
          )}
        </div>
        <div className="form-control">
          <label></label>
          <button type="submit">Login</button>
```

```
        </div>
      </form>
    </div>
  );
}
```

As you can see, we're getting instant validation errors for each input field once we submit the form and then try to enter the values in the input fields.

If there is any error for any of the input field, the `errors` object will be populated with the type of error which we're using to display our own custom error message like this:

```
{errors.email && errors.email.type === "required" && (
    <p className="errorMsg">Email is required.</p>
)}
{errors.email && errors.email.type === "pattern" && (
    <p className="errorMsg">Email is not valid.</p>
)}
```

Here, based on the type of error, we're displaying different error messages.

Using the ES11 optional chaining operator, you can further simplify the above code like this:

```
{errors.email?.type === "required" && (
    <p className="errorMsg">Email is required.</p>
)}
{errors.email?.type === "pattern" && (
    <p className="errorMsg">Email is not valid.</p>
)}
```

In the similar way, we have added the password field validation.

Also, as you can see, each input field is automatically focused when we submit the form if there is any validation error for that input field.

Also, the form is not submitted as long as there is a validation error. If you check the browser console, you will see that the `console.log` statement is only printed if the form is valid and there are no errors.

But as the number of validations for each field increases, the conditional checks and error message code will still increase. So we can further refactor the code to make it even simpler.

Take a look at the below code:

```
import React from "react";
import { useForm } from "react-hook-form";
import "./styles.css";

export default function App() {
  const {
    register,
    handleSubmit,
    formState: { errors }
  } = useForm();

  const onSubmit = (data) => {
    console.log(data);
  };

  return (
    <div className="App">
      <form onSubmit={handleSubmit(onSubmit)}>
        <div className="form-control">
          <label>Email</label>
          <input
            type="text"
            name="email"
            {...register("email", {
              required: "Email is required.",
              pattern: {
                value: /^[^@ ]+@[^@ ]+\.[^@ .]{2,}$/,
                message: "Email is not valid."
              }
            })}
          />
          {errors.email && <p
className="errorMsg">{errors.email.message}</p>}
        </div>
        <div className="form-control">
          <label>Password</label>
          <input
            type="password"
            name="password"
            {...register("password", {
              required: "Password is required.",
              minLength: {
                value: 6,
                message: "Password should be at-least 6 characters."
              }
            })}
          />
          {errors.password && (
            <p className="errorMsg">{errors.password.message}</p>
          )}
        </div>
```

```
        <div className="form-control">
          <label></label>
          <button type="submit">Login</button>
        </div>
      </form>
    </div>
  );
}
```

In the code above, we have changed the email and password validation code.

For the email input field, we changed this previous code:

```
 {...register("email", {
     required: true,
     pattern: /^[^@ ]+@[^@ ]+\.[^@ .]{2,}$/
 })}
```

to the below code:

```
{...register("email", {
    required: "Email is required.",
    pattern: {
        value: /^[^@ ]+@[^@ ]+\.[^@ .]{2,}$/,
        message: "Email is not valid."
    }
})}
```

Here, we've directly provided the error message we want to display while adding the validation itself.

So we no longer need to add extra checks for each validation. We are displaying the error message using the `message` property available inside the `errors` object for each input field like this:

```
{errors.email && <p className="errorMsg">{errors.email.message}</p>}
```

So by doing it this way, the code is further simplified which makes it easy to add extra validations in the future.

Note that, if there are validation errors, the onSubmit handler will not be executed and the corresponding input field will automatically be focused (which is a good thing).

How to Add a Multiple Validations

You can even provide multiple validations for the input field by adding a `validate` object. This is useful if you need to perform complex validations like this:

```
<input
    type="password"
    name="password"
    {...register("password", {
        required: true,
        validate: {
            checkLength: (value) => value.length >= 6,
            matchPattern: (value) =>
            /(?=.*\d)(?=.*[a-z])(?=.*[A-Z])(?!.*\s)(?=.*[!@#$*])/.test(
                value
            )
        }
    })}
/>
```

and to display the error messages, we use it like this:

```
{errors.password?.type === "required" && (
    <p className="errorMsg">Password is required.</p>
)}
{errors.password?.type === "checkLength" && (
    <p className="errorMsg">
        Password should be at-least 6 characters.
    </p>
)}
{errors.password?.type === "matchPattern" && (
    <p className="errorMsg">
        Password should contain at least one uppercase letter, lowercase
letter, digit, and special symbol.
    </p>
)}
```