React hook Form

1. Introduction
2. What is react hook form

React Hook Form is a popular open-source library in the React ecosystem that provides a set of hooks and utilities for managing form state, validation, and submission in React applications. It allows developers to create complex forms with minimal boilerplate code and provides a flexible and efficient way to handle form data and user input. React Hook Form is known for its simplicity, performance optimizations, and ease of integration into React applications.

Une image contenant texte, capture d’écran, Police

Description générée automatiquementUne image contenant texte, capture d’écran, carte de visite, Police

Description générée automatiquement

1. Installation

npm install react-hook-form

1. HookFrorm
2. HookForm and reRendring

React Hook Form is designed to minimize unnecessary re-renders in your components, providing efficient form handling. It achieves this by leveraging React's built-in optimizations and using the **useForm** hook to manage form state.

1. DevTools

**@hookform/devtools** is a browser devtools extension for React Hook Form. It's designed to help developers debug and inspect the state of forms built with React Hook Form in their applications. This extension provides a visual interface that allows you to view and manipulate form data, validation, and other form-related information in real-time as your application runs in the browser.

With **@hookform/devtools**, you can:

1. **Inspect Form State**: View the current state of your form, including form data, errors, touched fields, and more.
2. **Modify Form State**: You can manually modify form values, errors, and other properties to test different scenarios and see how your form responds.
3. **Validate Fields**: Trigger form validation to see which fields have errors and what the error messages are.
4. **View Field Interactions**: See which fields have been interacted with (touched) and which ones haven't.
5. **Real-time Updates**: Changes in form state are updated in real-time as your application runs, making it easier to identify and resolve issues.
6. useForm

In React Hook Form, **useForm** is a hook that you can use to initialize and configure your form. It provides access to various methods and properties for managing form state, validation, and submission. Here's an example of how you can use **useForm**:

 function MyForm() {

    // Initialize the form using the useForm hook

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

    // Define a function to handle form submission

    const onSubmit = (data) => {

      // Do something with the form data

      console.log(data);

    };

    return (

      <form onSubmit={handleSubmit(onSubmit)}>

        {/\* Input fields \*/}

        <input type="text" name="firstName" ref={register({ required: true })} />

        {errors.firstName && <span>This field is required</span>}

        <input type="text" name="lastName" ref={register} />

        {/\* Submit button \*/}

        <button type="submit">Submit</button>

      </form>

    );

  }

In this example, **useForm** is used to initialize the form and obtain the **register**, **handleSubmit**, and **errors** properties. **register** is used to register input fields, **handleSubmit** is used to define the submission handler, and **errors** is used to handle form validation errors. This is a basic usage example, and React Hook Form offers many more features for advanced form handling.

* 1. Handle submission Error

Handling submission errors in React Hook Form is essential for providing a smooth user experience and giving feedback when something goes wrong during form submission.

  const onError = (errors: FieldErrors<FormValues>) => {

    console.log("Form errors", errors);

  };

<form onSubmit={handleSubmit(onSubmit, onError)}>

</form>

1. Register

In React Hook Form, the **register** function is used to register an input element in your form. By registering an input element, React Hook Form can track its value, handle validation, and collect form data when the form is submitted.

Here's how you typically use **register**:

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

function MyForm() {

  const { register, handleSubmit } = useForm();

  const onSubmit = (data) => {

    console.log(data);

  };

  return (

    <form onSubmit={handleSubmit(onSubmit)}>

      <input

        type="text"

        name="firstName"

        ref={register({ required: true })} // Register the input with validation rules

      />

      <input

        type="text"

        name="lastName"

        ref={register} // Register the input without validation rules

      />

      <button type="submit">Submit</button>

    </form>

  );

}

In the example above:

* We import **useForm** from 'react-hook-form' to create our form.
* Inside the form, we have two input elements, 'firstName' and 'lastName'.
* We use the **ref** attribute to register each input element using the **register** function.
* For the 'firstName' input, we specify validation rules by passing an object to **register**. In this case, we've marked it as required.
* For the 'lastName' input, we register it without any validation rules.

By registering these inputs, React Hook Form will handle their state and validation automatically. When you submit the form, you can access the form data in the **onSubmit** function.

1. Submission

In React Hook Form, the **handleSubmit** function is used to handle form submissions. It works in conjunction with the **onSubmit** function to manage the submission of form data. Here's how you use it:

type FormValues = {

  userName: string;

  email: string;

  channel: string;

};

export const YouTubeForm = () => {

  const form = useForm<FormValues>();

  const { register, control, handleSubmit } = form;

  const onSubmit = (data: FormValues) => {

    console.log("Form submitted", data);

  };

  renderCount++;

  return (

    <div>

      <h1>YouTube Form {renderCount / 2}</h1>

      <form onSubmit={handleSubmit(onSubmit)}>

        <label htmlFor="username">Username</label>

        <input type="text" id="userName" {...register("userName")} />

        <label htmlFor="email">E-mail</label>

        <input type="email" id="email" {...register("email")} />

        <label htmlFor="channel">Channel</label>

        <input type="text" id="channel" {...register("channel")} />

        <button>Submit</button>

      </form>

      {/\* <DevTool control={control} /> \*/}

    </div>

  );

};

Using **handleSubmit** simplifies the process of handling form submissions in React Hook Form and ensures that your form data is captured and passed to the **onSubmit** function when the form is submitted.

1. Validation

React Hook Form provides a straightforward way to handle form validation. You can define validation rules for your form fields using the **register** function and specifying validation criteria using the **validate** property. Additionally, you can display validation error messages to users based on the validation results.

Apply validation

React Hook Form makes form validation easy by aligning with the existing [HTML standard for form validation](https://developer.mozilla.org/en-US/docs/Learn/HTML/Forms/Form_validation).

List of validation rules supported:

* required
* min
* max
* minLength
* maxLength
* pattern
* validate
  1. Validation

To create custom validation in React Hook Form, you can use the **register** function along with your custom validation function.

type FormValues = {

  userName: string;

  email: string;

  channel: string;

};

export const YouTubeForm = () => {

  const form = useForm<FormValues>();

  const {

    register,

    control,

    handleSubmit,

    formState: { errors },

  } = form;

  const onSubmit = (data: FormValues) => {

    console.log("Form submitted", data);

  };

  renderCount++;

  return (

    <div>

      <h1>YouTube Form {renderCount / 2}</h1>

      <form onSubmit={handleSubmit(onSubmit)}>

        <div className="form-control">

          <label htmlFor="username">Username</label>

          <input

            type="text"

            id="userName"

            {...register("userName", {

              required: "Username is required",

            })}

          />

          <p className="error">{errors.userName?.message}</p>

        </div>

        <div className="form-control">

          <label htmlFor="email">E-mail</label>

          <input

            type="email"

            id="email"

            {...register("email", {

              required: "Email is required",

              pattern: {

                value: /^(?!\s\*$)(^\S+@\S+\.\S+$)$/,

                message: "Invalid email format",

              },

              validate: {

                notAdmin: (fieldValue) => {

                  return (

                    fieldValue !== "admin@example.com" ||

                    "Enter a different email adreess"

                  );

                },

                notBlackListed: (fieldValue) => {

                  return (

                    !fieldValue.endsWith("baddomain.com") ||

                    "This domain is not supported"

                  );

                },

              },

            })}

          />

          <p className="error">{errors.email?.message}</p>

        </div>

        <div className="form-control">

          <label htmlFor="channel">Channel</label>

          <input

            type="text"

            id="channel"

            {...register("channel", {

              required: {

                value: true,

                message: "Channel is required",

              },

            })}

          />

          <p className="error">{errors.channel?.message}</p>

        </div>

        <button>Submit</button>

      </form>

      <DevTool control={control} />

    </div>

  );

};

* 1. Async validation

 validate: {

                notAdmin: (fieldValue) => {

                  return (

                    fieldValue !== "admin@example.com" ||

                    "Enter a different email adreess"

                  );

                },

                notBlackListed: (fieldValue) => {

                  return (

                    !fieldValue.endsWith("baddomain.com") ||

                    "This domain is not supported"

                  );

                },

                emailAvialable: async (fieldValue) => {

                  const response = await fetch(

                    `https://jsonplaceholder.typicode.com/users?email=${fieldValue}`

                  );

                  const data = await response.json();

                  return data.length == 0 || "Email already exists";

                },

              },

* 1. Disable

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

1. Default value

Normal default:

 const form = useForm<FormValues>({

    defaultValues: {

      userName: "Batman",

      email: "",

      channel: "",

    },

  });

Async Default:

 const form = useForm<FormValues>({

    defaultValues: async () => {

      const response = await fetch(

        "https://jsonplaceholder.typicode.com/users/1"

      );

      const data = await response.json();

      return {

        userName: "Batman",

        email: data.email,

        channel: "",

      };

    },

  });

1. Reset

In React Hook Form, you can reset the form to its initial state by using the **reset** method provided by the hook. The **reset** method clears the form data, errors, and any custom field values you may have set.

  useEffect(() => {

    if (isSubmitSuccessful) {

      reset();

    }

  }, [isSubmitSuccessful]);

<button

          type="button"

          onClick={() => {

            reset();

          }}

        >

          Reset

        </button>

1. Nested Object

const form = useForm<FormValues>({

    defaultValues: {

      userName: "Batman",

      email: "",

      channel: "",

      social: {

        twitter: "",

        facebook: "",

      },

    },

  });

<div className="form-control">

          <label htmlFor="Twitter">twitter</label>

          <input type="text" id="channel" {...register("social.twitter")} />

        </div>

        <div className="form-control">

          <label htmlFor="facebook">facebook</label>

          <input type="text" id="channel" {...register("social.facebook")} />

        </div>

1. Array

const form = useForm<FormValues>({

    defaultValues: {

      userName: "Batman",

      email: "",

      channel: "",

      social: {

        twitter: "",

        facebook: "",

      },

      phoneNumbers: ["", ""],

    },

  });

 <div className="form-control">

          <label htmlFor="primary-phone">Primary phone number</label>

          <input

            type="text"

            id="primary-phone"

            {...register("phoneNumbers.0")}

          />

        </div>

        <div className="form-control">

          <label htmlFor="secondary-phone">Secondary phone number</label>

          <input

            type="text"

            id="secondary-phone"

            {...register("phoneNumbers.1")}

          />

        </div>

1. Numeric and date values

type FormValues = {

  userName: string;

  email: string;

  channel: string;

  social: {

    twitter: string;

    facebook: string;

  };

  phoneNumbers: string[];

  phNumbers: {

    number: string;

  }[];

  age: number;

  dob: Date;

};

 const form = useForm<FormValues>({

    defaultValues: {

      userName: "Batman",

      email: "",

      channel: "",

      social: {

        twitter: "",

        facebook: "",

      },

      phoneNumbers: ["", ""],

      phNumbers: [{ number: "" }],

      age: 0,

      dob: new Date(),

    },

  });

 <div className="form-control">

          <label htmlFor="age">Age</label>

          <input

            type="number"

            id="age"

            {...register("age", {

              valueAsNumber: true,

              required: {

                value: true,

                message: "Age is required",

              },

            })}

          />

        </div>

        <div className="form-control">

          <label htmlFor="date">Data</label>

          <input

            type="date"

            id="dob"

            {...register("dob", {

              valueAsDate: true,

              required: {

                value: true,

                message: "Date is required",

              },

            })}

          />

        </div>

1. Api
2. useForm
   1. Watch

This method will watch specified inputs and return their values. It is useful to render input value and for determining what to render by condition.

const {

    register,

    control,

    handleSubmit,

    formState: { errors },

    watch,

  } = form;

useEffect((): any => {

    const subscription = watch((value) => {

      console.log(value);

    });

    return () => subscription.unsubscribe;

  }, [watch]);

* 1. getValue

An optimized helper for reading form values. The difference between watch and getValues is that getValues **will not** trigger re-renders or subscribe to input changes.

 const {

    register,

    control,

    handleSubmit,

    formState: { errors },

    watch,

    getValues,

  } = form;

 const handleGetValues = () => {

    console.log("Get values", getValues(["userName", "email"]));

  };

 <button type="button" onClick={handleGetValues}>

          Get values

        </button>

* 1. setValue

This function allows you to dynamically set the value of a **registered** field and have the options to validate and update the form state. At the same time, it tries to avoid unnecessary rerender.

 const {

    register,

    control,

    handleSubmit,

    formState: { errors },

    watch,

    getValues,

    setValue,

  } = form;

  const handleSetValues = () => {

    setValue("userName", "", {

      shouldValidate: true,

      shouldDirty: true,

      shouldTouch: true,

    });

  };

<button type="button" onClick={handleSetValues}>

          Set values

        </button>

* 1. Touched and dirty state

In React Hook Form, you can use the concepts of "touched" and "dirty" to track the state of form fields. These concepts help you determine whether a user has interacted with a field (touched) and whether the field value has changed from its initial state (dirty). React Hook Form provides functions and properties to work with these states:

 const {

    register,

    control,

    handleSubmit,

    formState: { errors, touchedFields, dirtyFields, isDirty },

    watch,

    getValues,

    setValue,

  } = form;

  console.log("Toued and dirty", touchedFields, dirtyFields, isDirty);

**touchedFields:**

**formState.touchedFields** is an object that contains information about which form fields have been touched (i.e., focused on and then blurred or interacted with).

**dirtyFields:**

**formState.dirtyFields** is an object that contains information about which form fields have changed from their initial values (i.e., they are "dirty").

**isDirty:**

**formState.isDirty** is a boolean value that indicates whether the entire form is dirty, meaning at least one field has changed from its initial value.

* 1. IsValid

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

<button disabled={!isValid}>Submit</button>

* 1. IsSubmitting &isSubmitted & isSubmitSuccessful & submitCount

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Une image contenant capture d’écran, Police, Graphique, texte

Description générée automatiquement

console.log(

    "Submit",

    isSubmitting,

    isSubmitted,

    isSubmitSuccessful,

    submitCount

  );

1. Dynamic fields (useFieldArray)

In React Hook Form (RHF), **useFieldArray** is a custom hook that allows you to work with arrays of input fields more easily. It's particularly useful when you need to manage dynamic sets of input fields, such as a list of items, without the need to manually control each field.

Here's how you can use **useFieldArray** in React Hook Form:

<https://react-hook-form.com/docs/usefieldarray>

const { fields, append, remove } = useFieldArray({

    name: "phNumbers",

    control,

  });

 <div className="form-control">

          <label htmlFor="phoneNumbers">List of phone numbers</label>

          {fields.map((field, index) => {

            return (

              <div className="form-control">

                <input

                  type="text"

                  {...register(`phNumbers.${index}.number` as const)}

                />

                {index > 0 && (

                  <button type="button" onClick={() => remove(index)}>

                    Remove

                  </button>

                )}

              </div>

            );

          })}

          <button type="button" onClick={() => append({ number: "" })}>

            Add phone number

          </button>

        </div>

1. Mode

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

 const form = useForm<FormValues>({

    defaultValues: {

      userName: "Batman",

      email: "",

      channel: "",

      social: {

        twitter: "",

        facebook: "",

      },

      phoneNumbers: ["", ""],

      phNumbers: [{ number: "" }],

      age: 0,

      dob: new Date(),

    },

    mode: "onChange",

  });

1. Trigger

In React Hook Form (RHF), a "trigger" refers to a function or method that manually triggers the validation and re-rendering of one or more form fields. This can be useful in situations where you want to programmatically validate and update the state of specific fields or the entire form.

The primary function used for triggering validation in RHF is **trigger**, which is available as a method on the **form** object returned by the **useForm** hook.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

 const form = useForm<FormValues>({

    defaultValues: {

      userName: "Batman",

      email: "",

      channel: "",

      social: {

        twitter: "",

        facebook: "",

      },

      phoneNumbers: ["", ""],

      phNumbers: [{ number: "" }],

      age: 0,

      dob: new Date(),

    },

    // mode: "onSubmit",

  });

  const {

    register,

    control,

    handleSubmit,

    formState: {

      errors,

      touchedFields,

      dirtyFields,

      isDirty,

      isValid,

      isSubmitting,

      isSubmitted,

      isSubmitSuccessful,

      submitCount,

    },

    watch,

    getValues,

    setValue,

    reset,

    trigger,

  } = form;

<button type="button" onClick={() => trigger()}>

          Trigger All

        </button>

        <button type="button" onClick={() => trigger("channel")}>

          Trigger channel

        </button>

1. MUI Integration

import { useForm } from "react-hook-form";

import { TextField, Button, Stack } from "@mui/material";

import { DevTool } from "@hookform/devtools";

type FormValues = {

  email: string;

  password: string;

};

let renderCount = 0;

export const LoginForm = () => {

  const {

    handleSubmit,

    register,

    formState: { errors },

    control,

  } = useForm<FormValues>({

    defaultValues: {

      email: "",

      password: "",

    },

  });

  const onSubmit = (data: FormValues) => {

    console.log(data);

  };

  renderCount++;

  return (

    <>

      <h1>Login ({renderCount / 2})</h1>

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

        <Stack spacing={2} width={400}>

          <TextField

            label="Email"

            type="email"

            {...register("email", { required: "Email is required" })}

            error={!!errors.email}

            helperText={errors.email?.message}

          />

          <TextField

            label="Password"

            type="password"

            {...register("password", { required: "Password is required" })}

            error={!!errors.password}

            helperText={errors.password?.message}

          />

          <Button type="submit" variant="contained" color="primary">

            Login

          </Button>

        </Stack>

      </form>

      <DevTool control={control} />

    </>

  );

};

1. Summary

Une image contenant texte, capture d’écran, Police

Description générée automatiquement