

# React Hook Form

Version - **7.44.3**

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

<https://github.com/react-hook-form/react-hook-form/releases>

## Installation

Installing React Hook Form only takes a single command and you're ready to roll.

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

## Example

The following code excerpt demonstrates a basic usage example:

```
import { useForm, SubmitHandler } from "react-hook-form";

type Inputs = {
  example: string,
  exampleRequired: string,
};

export default function App() {
  const { register, handleSubmit, watch, formState: { errors } } = useForm<Inputs>();
  const onSubmit: SubmitHandler<Inputs> = data => console.log(data);

  console.log(watch("example")) // watch input value by passing the name of it

  return (
    /* "handleSubmit" will validate your inputs before invoking "onSubmit" */
    <form onSubmit={handleSubmit(onSubmit)}>
      /* register your input into the hook by invoking the "register" function */
      <input defaultValue="test" {...register("example")} />

      /* include validation with required or other standard HTML validation rules */
      <input {...register("exampleRequired", { required: true })} />
      /* errors will return when field validation fails */
      {errors.exampleRequired && <span>This field is required</span>}

      <input type="submit" />
    </form>
  );
}
```

## Register fields

One of the key concepts in React Hook Form is to **register** your component into the hook. This will make its value available for both the form validation and submission.

**Note:** Each field is **required** to have a name as a key for the registration process.

```
import ReactDOM from "react-dom";
import { useForm, SubmitHandler } from "react-hook-form";

enum GenderEnum {
  female = "female",
  male = "male",
  other = "other"
}

interface IFormInput {
  firstName: String;
  gender: GenderEnum;
}

export default function App() {
  const { register, handleSubmit } = useForm<IFormInput>();
  const onSubmit: SubmitHandler<IFormInput> = data => console.log(data);

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <label>First Name</label>
      <input {...register("firstName")} />
      <label>Gender Selection</label>
      <select {...register("gender")} >
        <option value="female">female</option>
        <option value="male">male</option>
        <option value="other">other</option>
      </select>
      <input type="submit" />
    </form>
  );
}
```

## Apply validation

React Hook Form makes form validation easy by aligning with the existing [HTML standard for form validation](#).

List of validation rules supported:

- required
- min

- max
- minLength
- maxLength
- pattern
- validate

You can read more detail on each rule in the [register section](#).

```
import { useForm, SubmitHandler } from "react-hook-form";

interface IFormInput {
  firstName: string;
  lastName: string;
  age: number;
}

export default function App() {
  const { register, handleSubmit } = useForm<IFormInput>();
  const onSubmit: SubmitHandler<IFormInput> = data => console.log(data);

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register("firstName", { required: true, maxLength: 20 })} />
      <input {...register("lastName", { pattern: /^[A-Za-z]+$/i })} />
      <input type="number" {...register("age", { min: 18, max: 99 })} />
      <input type="submit" />
    </form>
  );
}
```

## Integrating an existing form

Integrating an existing form should be simple. The important step is to `register` the component's `ref` and assign relevant props to your input.

```
import { Path, useForm, UseFormRegister, SubmitHandler } from "react-hook-form";

interface IFormValues {
  "First Name": string;
  Age: number;
}

type InputProps = {
  label: Path<IFormValues>;
  register: UseFormRegister<IFormValues>;
  required: boolean;
};

// The following component is an example of your existing Input Component
```

```

const Input = ({ label, register, required }: InputProps) => (
  <>
    <label>{label}</label>
    <input {...register(label, { required })} />
  </>
);

// you can use React.forwardRef to pass the ref too
const Select = React.forwardRef<
  HTMLSelectElement,
  { label: string } & ReturnType<UseFormRegister<IFormValues>>
>(({ onChange, onBlur, name, label }, ref) => {
  <>
    <label>{label}</label>
    <select name={name} ref={ref} onChange={onChange} onBlur={onBlur}>
      <option value="20">20</option>
      <option value="30">30</option>
    </select>
  </>
});

const App = () => {
  const { register, handleSubmit } = useForm<IFormValues>();

  const onSubmit: SubmitHandler<IFormValues> = data => {
    alert(JSON.stringify(data));
  };

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <Input label="First Name" register={register} required />
      <Select label="Age" {...register("Age")} />
      <input type="submit" />
    </form>
  );
};

```

## Integrating with UI libraries

React Hook Form has made it easy to integrate with external UI component libraries. If the component doesn't expose input's ref, then you should use the Controller component, which will take care of the registration process.

```

import Select from "react-select";
import { useForm, Controller, SubmitHandler } from "react-hook-form";
import Input from "@material-ui/core/Input";

interface IFormInput {
  firstName: string;
  lastName: string;
  iceCreamType: {label: string; value: string };
}

```

```

}

const App = () => {
  const { control, handleSubmit } = useForm<IFormInput>();

  const onSubmit: SubmitHandler<IFormInput> = data => {
    console.log(data)
  };

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <Controller
        name="firstName"
        control={control}
        defaultValue=""
        render={({ field }) => <Input {...field} />}
      />
      <Controller
        name="iceCreamType"
        control={control}
        render={({ field }) => <Select
          {...field}
          options={[
            { value: "chocolate", label: "Chocolate" },
            { value: "strawberry", label: "Strawberry" },
            { value: "vanilla", label: "Vanilla" }
          ]}
        />
      />
      <input type="submit" />
    </form>
  );
};

```

## Integrating Controlled Inputs

This library embraces uncontrolled components and native HTML inputs, however, it's hard to avoid working with external controlled components such as [React-Select](#), [AntD](#) and [MUI](#). To make this simple, we provide a wrapper component: [Controller](#) to streamline the integration process while still giving you the freedom to use a custom register.

```

import { useForm, Controller, SubmitHandler } from "react-hook-form";
import { TextField, Checkbox } from "@material-ui/core";

interface IFormInputs {
  TextField: string
  MyCheckbox: boolean
}

```

```
function App() {
  const { handleSubmit, control, reset } = useForm<IFormInputs>();
  const onSubmit: SubmitHandler<IFormInputs> = data => console.log(data);

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <Controller
        name="MyCheckbox"
        control={control}
        defaultValue={false}
        rules={{ required: true }}
        render={({ field }) => <Checkbox {...field} />}
      />
      <input type="submit" />
    </form>
  );
}
```

## Integrating with global state

It doesn't require you to rely on a state management library, but you can easily integrate with them.

```
import { useForm } from "react-hook-form";
import { connect } from "react-redux";
import updateAction from "../actions";

export default function App(props) {
  const { register, handleSubmit, setValue } = useForm();
  // Submit your data into Redux store
  const onSubmit = data => props.updateAction(data);

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register("firstName")} defaultValue={props.firstName} />
      <input {...register("lastName")} defaultValue={props.lastName} />
      <input type="submit" />
    </form>
  );
}

// Connect your component with redux
connect(({ firstName, lastName }) => ({ firstName, lastName }), updateAction)(YourForm);
```

## Handle errors

React Hook Form provides an `errors` object to show you the errors in the form. `errors`' type will return given validation constraints. The following example showcases a required validation rule.

```

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

interface IFormInputs {
  firstName: string
  lastName: string
}

const onSubmit: SubmitHandler<IFormInputs> = data => console.log(data);

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

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register("firstName", { required: true })} />
      {errors.firstName && "First name is required"}
      <input {...register("lastName", { required: true })} />
      {errors.lastName && "Last name is required"}
      <input type="submit" />
    </form>
  );
}

```

## Schema Validation

We also support schema-based form validation with [Yup](#), [Zod](#), [Superstruct](#) & [Joi](#), where you can pass your schema to [useForm](#) as an optional config. It will validate your input data against the schema and return with either [errors](#) or a valid result.

**Step 1:** Install `yup` into your project.

```
npm install @hookform/resolvers yup
```

**Step 2:** Prepare your schema for validation and register inputs with React Hook Form.

```

import { useForm } from "react-hook-form";
import { yupResolver } from '@hookform/resolvers/yup';
import * as yup from "yup";

interface IFormInputs {
  firstName: string
  age: number
}

const schema = yup.object({
  firstName: yup.string().required(),
  age: yup.number().positive().integer().required(),
}).required();

export default function App() {

```

```

const { register, handleSubmit, formState: { errors } } = useForm<IFormInputs>({
  resolver: yupResolver(schema)
});
const onSubmit = (data: IFormInputs) => console.log(data);

return (
  <form onSubmit={handleSubmit(onSubmit)}>
    <input {...register("firstName")} />
    <p>{errors.firstName?.message}</p>

    <input {...register("age")} />
    <p>{errors.age?.message}</p>

    <input type="submit" />
  </form>
);
}

```

## React Native

You will get the same performance boost and enhancement in React Native. To integrate with input component, you can wrap it with `Controller`.

```

import { Text, View, TextInput, Button, Alert } from "react-native";
import { useForm, Controller } from "react-hook-form";

export default function App() {
  const { control, handleSubmit, formState: { errors } } = useForm({
    defaultValues: {
      firstName: "",
      lastName: ""
    }
  });
  const onSubmit = data => console.log(data);

  return (
    <View>
      <Controller
        control={control}
        rules={{
          required: true,
        }}
        render={({ field: { onChange, onBlur, value } }) => (
          <TextInput
            style={styles.input}
            onBlur={onBlur}
            onChangeText={onChange}
            value={value}
          />
        )}
      />
      <Text>name="firstName"

```



```

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

<Controller
  control={control}
  rules={{
    maxLength: 100,
  }}
  render={({ field: { onChange, onBlur, value } }) => (
    <TextInput
      style={styles.input}
      onBlur={onBlur}
      onChangeText={onChange}
      value={value}
    />
  )}
  name="lastName"
/>

<Button title="Submit" onPress={handleSubmit(onSubmit)} />
</View>
);
}

```

## TypeScript

React Hook Form is built with TypeScript, and you can define a `FormData` type to support form values.

```

import * as React from "react";
import { useForm } from "react-hook-form";

type FormData = {
  firstName: string;
  lastName: string;
};

export default function App() {
  const { register, setValue, handleSubmit, formState: { errors } } = useForm<FormData>();
  const onSubmit = handleSubmit(data => console.log(data));
  // firstName and lastName will have correct type

  return (
    <form onSubmit={onSubmit}>
      <label>First Name</label>
      <input {...register("firstName")} />
      <label>Last Name</label>
      <input {...register("lastName")} />
      <button
        type="button"
        onClick={() => {

```

```

    setValue("lastName", "luo"); // ✓
    setValue("firstName", true); // ✗ : true is not string
    errors.bill; // ✗ : property bill does not exist
  }}
  >
  SetValue
</button>
</form>
);
}

```

## Design and philosophy

React Hook Form's design and philosophy focus on user and developer experience. The library aims to provide users with a smoother interaction experience by fine-tuning the performance and improving accessibility. Some of the performance enhancements include:

- Introducing form state subscription model through the proxy
- Avoiding unnecessary computation
- Isolating component re-rendering when required

Overall, it improves the user experience while users interact with the application. As for the developers, we introduce built-in validation and are closely aligned with HTML standards allowing further extension with powerful validation methods and integration with schema validation natively. In addition, having a strongly type-checked form with the help of typescript provides early build-time feedback to help and guide the developer to build a robust form solution.

## useForm

React hooks for form validation

**useForm:** [UseFormProps](#)

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.

Generic props:

<a href="#"><u>mode</u></a>	Validation strategy before submitting behaviour.
<a href="#"><u>reValidateMode</u></a>	Validation strategy after submitting behaviour.
<a href="#"><u>defaultValues</u></a>	Default values for the form.
<a href="#"><u>values</u></a>	Reactive values to update the form values.

<u>resetOptions</u>	Option to reset form state update while updating new form values.
<u>criteriaMode</u>	Display all validation errors or one at a time.
<u>shouldFocusError</u>	Enable or disable built-in focus management.
<u>delayError</u>	Delay error from appearing instantly.
<u>shouldUseNativeValidation</u>	Use browser built-in form constraint API.
<u>shouldUnregister</u>	Enable and disable input unregister after unmount.
Schema validation props:	
<u>resolver</u>	Integrates with your preferred schema validation library.
<u>context</u>	A context object to supply for your schema validation.

## Props

mode: onChange | onBlur | onSubmit | onTouched | all = 'onSubmit'

### !React Native: compatible with Controller

This option allows you to configure the validation strategy before a user submits the form. The validation occurs during the `onSubmit` event, which is triggered by invoking the `handleSubmit` function.

Name	Type	Description
onSubmit	string	Validation is triggered on the <code>submit</code> event, and inputs attach <code>onChange</code> event listeners to re-validate themselves.
onBlur	string	Validation is triggered on the <code>blur</code> event.
onChange	string	Validation is triggered on the <code>change</code> event for each input, leading to multiple re-renders. Warning: this often comes with a significant impact on performance.
onTouched	string	Validation is initially triggered on the first <code>blur</code> event. After that, it is triggered on every <code>change</code> event. Note: when using with <code>Controller</code> , make sure to wire up <code>onBlur</code> with the <code>render</code> prop.
all	string	Validation is triggered on both <code>blur</code> and <code>change</code> events.

### reValidateMode: onChange | onBlur | onSubmit = 'onChange' !React Native: Custom register or using Controller

This option allows you to configure validation strategy when inputs with errors get re-validated **after** a user submits the form (`onSubmit` event and `handleSubmit` function executed). By default, re-validation occurs during the input change event.

**defaultValues: FieldValues | Promise<FieldValues>**

The `defaultValues` prop populates the entire form with default values. It supports both synchronous and asynchronous assignment of default values. While you can set an input's default value using `defaultValue` or `defaultChecked` ([as detailed in the official React documentation](#)), it is **recommended** to use `defaultValues` for the entire form.

```
// set default value sync
useForm({
  defaultValues: {
    firstName: "",
    lastName: ""
  }
})

// set default value async
useForm({
  defaultValues: async () => fetch('/api-endpoint');
})
```

#### Rules

- You **should avoid** providing undefined as a default value, as it conflicts with the default state of a controlled component.
- `defaultValues` are cached. To reset them, use the [reset](#) API.
- `defaultValues` will be included in the submission result by default.
- It's recommended to avoid using custom objects containing prototype methods, such as Moment or Luxon, as `defaultValues`.
- There are other options for including form data

```
// include hidden input

```

**values: FieldValues**

The `values` props will react to changes and update the form values, which is useful when your form needs to be updated by external state or server data.

```
// set default value sync
function App({ values }) {
  useForm({
    values // will get updated when values props updates
  })
}
```

```
function App() {
  const values = useFetch('/api');

  useForm({
    defaultValues: {
      firstName: "",
      lastName: "",
    },
    values, // will get updated once values returns
  })
}
```

#### **resetOptions: KeepStateOptions**

This property is related to value update behaviors. When `values` or `defaultValues` are updated, the `reset` API is invoked internally. It's important to specify the desired behavior after `values` or `defaultValues` are asynchronously updated. The configuration option itself is a reference to the [reset](#) method's options.

```
// by default asynchronously value or defaultValues update will reset the form values
useForm({ values })
useForm({ defaultValues: async () => await fetch() })

// options to config the behaviour
// eg: I want to keep user interacted/dirty value and not remove any user errors
useForm({
  values,
  resetOptions: {
    keepDirtyValues: true, // user-interacted input will be retained
    keepErrors: true, // input errors will be retained with value update}})
```

#### **context: object**

This context object is mutable and will be injected into the resolver's second argument or [Yup](#) validation's context object. [CODESANDBOX](#)

#### **criteriaMode: firstError | all**

- When set to `firstError` (default), only the first error from each field will be gathered. [CODESANDBOX](#)
- When set to `all`, all errors from each field will be gathered.

#### **shouldFocusError: boolean = true**

When set to `true` (default), and the user submits a form that fails validation, focus is set on the first field with an error.

**Note:** only registered fields with a `ref` will work. Custom registered inputs do not apply. For example: `register('test')` // doesn't work

**Note:** the focus order is based on the register order.

`delayError: number`

This configuration delays the display of error states to the end-user by a specified number of milliseconds. If the user corrects the error input, the error is removed instantly, and the delay is not applied. [CODESANDBOX](#)

`shouldUnregister: boolean = false`

By default, an input value will be retained when input is removed. However, you can set `shouldUnregister` to `true` to unregister input during unmount.

- This is a global configuration that overrides child-level configurations. To have individual behavior, set the configuration at the component or hook level, not at `useForm`.
- By default, `shouldUnregister: false` means unmounted fields are **not validated** by built-in validation.
- By setting `shouldUnregister` to `true` at `useForm` level, `defaultValues` will **not** be merged against submission result.
- Setting `shouldUnregister: true` makes your form behave more closely to native forms.
  - Form values are stored within the inputs themselves.
  - Unmounting an input removes its value.
  - Hidden inputs should use the `hidden` attribute for storing hidden data.
  - Only registered inputs are included as submission data.
  - Unmounted inputs must be notified at either `useForm` or `useWatch`'s `useEffect` for the hook form to verify that the input is unmounted from the DOM.

```
const NotWork = () => {
  const [show, setShow] = React.useState(false);
  // ✗ won't get notified, need to invoke unregister
  return {show && <input {...register('test')} />}
}

const Work = ({ control }) => {
  const { show } = useWatch({ control })
  // ✓ get notified at useEffect
  return {show && <input {...register('test1')} />}
}

const App = () => {
  const [show, setShow] = React.useState(false);
  const { control } = useForm({ shouldUnregister: true });
  return (
    <div>
      // ✓ get notified at useForm's useEffect
      {show && <input {...register('test2')} />}
      <NotWork />
      <Work control={control} />
    </div>
  )
}
```

`shouldUseNativeValidation: boolean = false`

This config will enable [browser native validation](#). It will also enable CSS selectors `:valid` and `:invalid` making styling inputs easier. You can still use these selectors even when client-side validation is disabled.

- Only works with `onSubmit` and `onChange` modes, as the `reportValidity` execution will focus the error input.
- Each registered field's validation message is required to be string to display them natively.
- This feature only works with the `register` API and `useController/Controller` that are connected with actual DOM references.

Examples

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

export default function App() {
  const { register, handleSubmit } = useForm({ shouldUseNativeValidation: true });
  const onSubmit = async data => { console.log(data); };

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input
        {...register("firstName", { required: "Please enter your first name." })} //
        custom message
      />
      <input type="submit" />
    </form>
  );
}
```

resolver: [Resolver](#)

This function allows you to use any external validation library such as [Yup](#), [Zod](#), [Joi](#), [Vest](#), [Ajv](#) and many others. The goal is to make sure you can seamlessly integrate whichever validation library you prefer. If you're not using a library, you can always write your own logic to validate your forms.

```
npm install @hookform/resolvers
```

Props

Name	Type	Description
values	object	This object contains the entire form values.
context	object	This is the context object which you can provide to the <code>useForm</code> config. It is a mutable object that can be changed on each re-render.
options	<code>{ criteriaMode: string, fields: object, names: string[] }</code>	This is the option object containing information about the validated fields, names and <code>criteriaMode</code> from <code>useForm</code> .

## Rules

- Schema validation focuses on field-level error reporting. Parent-level error checking is limited to the direct parent level, which is applicable for components such as group checkboxes.
- This function will be cached.
- Re-validation of an input will only occur one field at time during a user's interaction. The lib itself will evaluate the error object to trigger a re-render accordingly.
- A resolver can not be used with the built-in validators (e.g.: required, min, etc.)
- When building a custom resolver:
  - Make sure that you return an object with both values and errors properties. Their default values should be an empty object. For example: {}.
  - The keys of the error object should match the name values of your fields.
  -

```
import React from 'react';
import { useForm } from 'react-hook-form';
import { yupResolver } from '@hookform/resolvers/yup';
import * as yup from "yup";

const schema = yup.object().shape({
  name: yup.string().required(),
  age: yup.number().required(),
}).required();

const App = () => {
  const { register, handleSubmit } = useForm({
    resolver: yupResolver(schema),
  });

  return (
    <form onSubmit={handleSubmit(d => console.log(d))}>
      <input {...register("name")} />
      <input type="number" {...register("age")} />
      <input type="submit" />
    </form>
  );
};
```

Tip / You can debug your schema via the following code snippet:

```
resolver: async (data, context, options) => {
  // you can debug your validation schema here
  console.log('formData', data)
  console.log('validation result', await anyResolver(schema)(data, context, options))
  return anyResolver(schema)(data, context, options)
},
```



**register:** (name: string, RegisterOptions?) => ({ onChange, onBlur, name, ref })

This method allows you to register an input or select element and apply validation rules to React Hook Form. Validation rules are all based on the HTML standard and also allow for custom validation methods.

By invoking the register function and supplying an input's name, you will receive the following methods:

## Props

Name	Type	Description
onChange	ChangeHandler	onChange prop to subscribe the input change event.
onBlur	ChangeHandler	onBlur prop to subscribe the input blur event.
ref	React.Ref<any>	Input reference for hook form to register.
name	string	Input's name being registered.
Input <b>Name</b>		Submit Result
register("firstName")		{firstName: 'value'}
register("name.firstName")		{name: { firstName: 'value' }}
register("name.firstName.0")		{name: { firstName: [ 'value' ] }}

## Return

**Tip::** What's happened to the input after invoke register API:

COPY

```
const { onChange, onBlur, name, ref } = register('firstName');
// include type check against field path with the name you have supplied.

<input
  onChange={onChange} // assign onChange event
  onBlur={onBlur} // assign onBlur event
  name={name} // assign name prop
  ref={ref} // assign ref prop
/>
// same as above
<input {...register('firstName')} />
```

## Options

By selecting the register option, the API table below will get updated.

Name	Description	Code Examples
<b>ref</b> React.Ref	React element ref	<pre>&lt;input {...register("test")} /&gt;</pre>
<b>required</b> string   { value: boolean, message: string }	A Boolean which, if true, indicates that the input must have a value before the form can be submitted. You can assign a string to return an error message in the errors object.  <b>Note:</b> This config aligns with web constrained API for required input validation, for object or array type of input use validate function instead.	<pre>&lt;input   {...register("test",   {     required: 'error message' // JS only: &lt;p&gt;error message&lt;/p&gt; TS only support string   })} /&gt;</pre>
<b>maxLength</b> { value: number, message: string }	The maximum length of the value to accept for this input.	<pre>&lt;input   {...register("test",   {     maxLength : {       value: 2,       message: 'error message' // JS only: &lt;p&gt;error message&lt;/p&gt; TS only support string     }   })} /&gt;</pre>
<b>minLength</b> { value: number, message: string }	The minimum length of the value to accept for this input.	<pre>&lt;input   {...register("test",   {     minLength: {       value: 1,       message: 'error message' // JS only: &lt;p&gt;error message&lt;/p&gt; TS only support string     }   })} /&gt;</pre>
<b>max</b> { value: number, message: string }	The maximum value to accept for this input.	<pre>&lt;input   type="number"   {...register('test',   {     max: {       value: 3,       message: 'error message' // JS only: &lt;p&gt;error message&lt;/p&gt; TS only support string     }   })} /&gt;</pre>
<b>min</b> { value: number, message: string }	The minimum value to accept for this input.	<pre>&lt;input   type="number"   {...register("test",   {     min: {       value: 3,       message: 'error message' // JS only: &lt;p&gt;error message&lt;/p&gt; TS only support string     })} /&gt;</pre>

<b>pattern</b> { value: RegExp, message: string }	The regex pattern for the input.  <b>Note:</b> A RegExp object with the /g flag keeps track of the lastIndex where a match occurred.	<input {...register("test", { pattern: { value: /[A-Za-z]{3}/, message: 'error message' // JS only: <p>error message</p> TS only support string } })} />
<b>validate</b> Function   Object	You can pass a callback function as the argument to validate, or you can pass an object of callback functions to validate all of them. This function will be executed on its own without depending on other validation rules included in the required attribute.  <b>Note:</b> for object or array input data, it's recommended to use the validate function for validation as the other rules mostly apply to string, string[], number and boolean data types.	<input {...register("test", { validate: value => value === '1'    'error message' // JS only: <p>error message</p> TS only support string })} /> // object of callback functions <input {...register("test1", { validate: { positive: v => parseInt(v) > 0    'should be greater than 0', lessThanTen: v => parseInt(v) < 10    'should be lower than 10', // you can do asynchronous validation as well checkUrl: async () => await fetch()    'error message', // JS only: <p>error message</p> TS only support string messages: v => !v && ['test', 'test2'] } })} />
<b>valueAsNumber:</b> boolean	Returns a Number normally. If something goes wrong NaN will be returned.  <ul style="list-style-type: none"> <li>• valueAs process is happening <b>before</b> validation.</li> <li>• Only applicable and support to text input, but we still cast to number type to the field value.</li> <li>• Does not transform defaultValue or defaultValues.</li> </ul>	<input type="number" {...register("test", { valueAsNumber: true, })} />

valueAsDate: boolean	Returns a Date object normally. If something goes wrong Invalid Date will be returned. <ul style="list-style-type: none"> <li>valueAs process is happening <b>before</b> validation.</li> <li>Only applies to text input.</li> <li>Does not transform defaultValue or defaultValues.</li> </ul>	<input &gt;="" <="" td="" true,="" type="date" valueasdate:="" {="" {...register("test",="" })}=""/>
setValueAs: <T> (value: any) => T	Return input value by running through the function. <ul style="list-style-type: none"> <li>valueAs process is happening <b>before</b> validation. Also, setValueAs is ignored if either valueAsNumber or valueAsDate are true.</li> <li>Only applies to text input.</li> <li>Does not transform defaultValue or defaultValues.</li> </ul>	<input &gt;="" <="" parseint(v),="" setvalueas:="" td="" type="number" v="&gt;" {="" {...register("test",="" })}=""/>
disabled boolean = false	Set disabled to true will lead input value to be undefined and input control to be disabled. <p>Disabled prop will also omit <b>build-in</b> validation rules.</p> <p>For schema validation, you can leverage the undefined value returned from input or context object.</p>	<input &gt;="" <="" disabled:="" td="" true="" {="" {...register("test",="" })}=""/>
onChange (e: SyntheticEvent) => void	onChange function event to be invoked in the change event.	register('firstName', { onChange: (e) => console.log(e) })
onBlur (e: SyntheticEvent) => void	onBlur function event to be invoked in the blur event.	register('firstName', { onBlur: (e) => console.log(e) })
value unknown	Set up value for the registered input. This prop should be utilised inside useEffect or invoke once, each re-run will update or overwrite the input value which you have supplied.	register('firstName', { value: 'bill' })
shouldUnregister: boolean	Input will be unregistered after unmount and defaultValues will be removed as well. <p><b>Note:</b> this prop should be avoided when using with useFieldArray as unregister function gets called after input unmount/remount and reorder.</p>	<input &gt;="" <="" shouldunregister:="" td="" true,="" {="" {...register("test",="" })}=""/>
deps: string   string[]	Validation will be triggered for the dependent inputs, it only limited to register api not trigger.	<input &gt;="" 'inputb'],="" <="" ['inputa',="" deps:="" td="" {="" {...register("test",="" })}=""/>

## Rules

- name is **required** and **unique** (except native radio and checkbox). Input name supports both dot and bracket syntax, which allows you to easily create nested form fields.
- name can neither start with a number nor use number as key name.
- we are using dot syntax only for typescript usage consistency, so bracket [ ] will not work for array form value.

```
register('test.0.firstName'); // ✓  
register('test[0]firstName'); // ✗
```

- disabled input will result in an undefined form value. If you want to prevent users from updating the input, you can use `readOnly` or disable the entire `<fieldset />`. Here is an [example](#).
- To produce an array of fields, input names should be followed by a dot and number. For example: `test.0.data`
- Changing the name on each render will result in new inputs being registered. It's recommend to keep static names for each registered input.
- Input value and reference will no longer gets removed based on unmount. You can invoke `unregister` to remove that value and reference.
- Individual register option can't be removed by `undefined` or `{}`. You can update individual attribute instead.

```
register('test', { required: true });  
register('test', {}); // ✗  
register('test', undefined); // ✗  
register('test', { required: false }); // ✓
```

## Examples

```
import * as React from "react";  
import { useForm } from "react-hook-form";  
  
export default function App() {  
  const { register, handleSubmit } = useForm();  
  const onSubmit = (data) => alert(JSON.stringify(data));  
  
  return (  
    <form onSubmit={handleSubmit(onSubmit)}>  
      <input {...register("firstName")} placeholder="First name" />  
  
      <input {...register("lastName")} placeholder="Last name" />  
  
      <select {...register("category")}>  
        <option value="">Select...</option>  
        <option value="A">Category A</option>  
        <option value="B">Category B</option>  
      </select>  
    </form>  
  );  
}
```

```
    </select>

    <input type="submit" />
  </form>
);
}
```

## Tips

### *Custom Register*

You can also register inputs with `useEffect` and treat them as virtual inputs. For controlled components, we provide a custom hook `useController` and `Controller` component to take care of this process for you.

If you choose to manually register fields, you will need to update the input value with `setValue`.

```
register('firstName', { required: true, min: 8 });

<TextInput onChange={(value) => setValue('lastName', value)} />
```

### *How to work with innerRef, inputRef?*

When the custom input component didn't expose ref correctly, you can get it working via the following.

```
// not working, because ref is not assigned
<TextInput {...register('test')} />

const firstName = register('firstName', { required: true })
<TextInput
  name={firstName.name}
  onChange={firstName.onChange}
  onBlur={firstName.onBlur}
  inputRef={firstName.ref} // you can achieve the same for different ref name such as innerRef
/>

// correct way to forward input's ref
const Select = React.forwardRef(({ onChange, onBlur, name, label }, ref) => (
  <select name={name} ref={ref} onChange={onChange} onBlur={onBlur}>
    <option value="20">20</option>
    <option value="30">30</option>
  </select>
));
```

**unregister:** (name: string | string[], options) => void

This method allows you to unregister a single input or an array of inputs. It also provides a second optional argument to keep state after unregistering an input.

## Props

The example below shows what to expect when you invoke the `unregister` method.

```
<input {...register('yourDetails.firstName')} />
<input {...register('yourDetails.lastName')} />
```

Type	Input Name	Value
string	<code>unregister("yourDetails")</code>	<code>{}</code>
string	<code>unregister("yourDetails.firstName")</code>	<code>{ lastName: '' }</code>
string[]	<code>unregister(["yourDetails.lastName"])</code>	<code>''</code>

### Options

Name	Type	Description	Code Examples
<code>keepDirty</code>	boolean	<code>isDirty</code> and <code>dirtyFields</code> will be remained during this action. However, this is not going to guarantee the next user input will not update <code>isDirty</code> formState, because <code>isDirty</code> is measured against the <code>defaultValues</code> .	<pre>unregister('test',   { keepDirty:     true } )</pre>
<code>keepTouched</code>	boolean	<code>touchedFields</code> will no longer remove that input after <code>unregister</code> .	<pre>unregister('test',   { keepTouched:     true } )</pre>
<code>keepIsValid</code>	boolean	<code>isValid</code> will be remained during this action. However, this is not going to guarantee the next user input will not update <code>isValid</code> for schema validation, you will have to adjust the schema according with the <code>unregister</code> .	<pre>unregister('test',   { keepIsValid:     true } )</pre>
<code>keepError</code>	boolean	errors will not be updated.	<pre>unregister('test',   { keepError:     true } )</pre>
<code>keepValue</code>	boolean	input's current value will not be updated.	<pre>unregister('test',   { keepValue:     true } )</pre>
<code>keepDefaultValue</code>	boolean	input's <code>defaultValue</code> which defined in <code>useForm</code> will be remained.	<pre>unregister('test',   {     keepDefaultValue:     true } )</pre>

## Rules

- This method will remove input reference and its value which means **build-in validation** rules will be removed as well.
- By unregister an input, it will not affect the schema validation.

```
const schema = yup.object().shape({
  firstName: yup.string().required()
}).required();

unregister("firstName"); // this will not remove the validation against firstName input
```

- Make sure you unmount that input which has register callback or else the input will get registered again.

```
const [show, setShow] = React.useState(true)

const onClick = () => {
  unregister('test');
  setShow(false); // make sure to unmount that input so register not invoked again.
}

{show && <input {...register('test')} />}
```

## Examples

```
import React, { useEffect } from "react";
import { useForm } from "react-hook-form";

interface IFormInputs {
  firstName: string;
  lastName?: string;
}

export default function App() {
  const { register, handleSubmit, unregister } = useForm<IFormInputs>();
  const onSubmit = (data: IFormInputs) => console.log(data);

  React.useEffect(() => {
    register("lastName");
  }, [register])

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <button type="button" onClick={() => unregister("lastName")}>unregister</button>
      <input type="submit" />
    </form>
  );
};
```



## formState: Object

This object contains information about the entire form state. It helps you to keep on track with the user's interaction with your form application.

### Return



Name	Type	Description
isDirty	boolean	<p>Set to true after the user modifies any of the inputs.</p> <ul style="list-style-type: none"><li>Make sure to provide all inputs' defaultValues at the useForm, so hook form can have a single source of truth to compare whether the form is dirty.</li></ul> <pre>const {   formState: { isDirty, dirtyFields },   setValue, } = useForm({ defaultValues: { test: "" } });  // isDirty: true setValue('test', 'change')  // isDirty: false because there getValues() === defaultValues setValue('test', '')</pre> <ul style="list-style-type: none"><li>File typed input will need to be managed at the app level due to the ability to cancel file selection and <a href="#">FileList</a> object.</li></ul>
dirtyFields	object	<p>An object with the user-modified fields. Make sure to provide all inputs' defaultValues via useForm, so the library can compare against the defaultValues.</p> <p>Dirty fields will <b>not</b> represent as isDirty formState, because dirty fields are marked field dirty at field level rather the entire form. If you want to determine the entire form state use isDirty instead.</p>
touchedFields	object	<p>An object containing all the inputs the user has interacted with.</p>
defaultValues	object	<p>The value which has been set at <a href="#">useForm</a>'s defaultValues or updated defaultValues via <a href="#">reset</a> API.</p>
isSubmitted	boolean	<p>Set to true after the form is submitted. Will remain true until the reset method is invoked.</p>

isSubmitSuccessful	boolean	Indicate the form was successfully submitted without any Promise rejection or Error been thrown within the handleSubmit callback.
isSubmitting	boolean	true if the form is currently being submitted. false otherwise.
submitCount	number	Number of times the form was submitted.
isValid	boolean	Set to true if the form doesn't have any errors. <ul style="list-style-type: none"> <li>• isValid is affected by mode at <a href="#">useForm</a>. This state is only applicable with onChange, onTouched, and onBlur mode.</li> <li>• setError has no effect on isValid formState, isValid will always derived via the entire form validation result.</li> </ul>
isValidating	boolean	Set to true during validation.
errors	object	An object with field errors. There is also an <a href="#">ErrorMessage</a> component to retrieve error message easily.

## Rules

- formState is wrapped with a [Proxy](#) to improve render performance and skip extra logic if specific state is not subscribed to. Therefore make sure you invoke or read it before a render in order to enable the state update.
- formState is updated in batch. If you want to subscribe to formState via useEffect, make sure that you place the entire formState in the optional array

## Snippet:

```
useEffect(() => {
  if (formState.errors.firstName) {
    // do the your logic here
  }
}, [formState]); // 
//  formState.errors will not trigger the useEffect
```

- Pay attention to the logical operator when subscription to `formState`.

```
// ✗ formState.isValid is accessed conditionally,
// so the Proxy does not subscribe to changes of that state
return <button disabled={!formState.isDirty || !formState.isValid} />;

// ✔ read all formState values to subscribe to changes
const { isDirty, isValid } = formState;
return <button disabled={!isDirty || !isValid} />;
```

### Example:

```
import React from "react";
import { useForm } from "react-hook-form";
type FormInputs = {
  test: string
}
export default function App() {
  const {
    register,
    handleSubmit,
    formState
  } = useForm<FormInputs>();
  const onSubmit = (data: FormInputs) => console.log(data);

  React.useEffect(() => {
    console.log("touchedFields", formState.touchedFields);
  }, [formState]);

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register("test")} />
      <input type="submit" />
    </form>
  );
}
```

- Pay attention to the logical operator when subscription to `formState`.

```
// ✗ formState.isValid is accessed conditionally,
// so the Proxy does not subscribe to changes of that state
return <button disabled={!formState.isDirty || !formState.isValid} />;

// ✔ read all formState values to subscribe to changes
const { isDirty, isValid } = formState;
return <button disabled={!isDirty || !isValid} />;
```

## Examples

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

type FormInputs = {
  test: string
}

export default function App() {
  const {
    register,
    handleSubmit,
    // Read the formState before render to subscribe the form state through Proxy
    formState: { errors, isDirty, isSubmitting, touchedFields, submitCount },
  } = useForm<FormInputs>();
  const onSubmit = (data: FormInputs) => console.log(data);

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register("test")} />
      <input type="submit" />
    </form>
  );
}
```

**watch:** (names?: string | string[] | (data, options) => void) => unknown

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.

### Props

Type	Description
string	Watch input value by name (similar to <a href="#">lodash get</a> function)
string[]	Watch multiple inputs
undefined	Watch all inputs
(data: unknown, { name: string, type: string }) => void	Watch all inputs and invoke a callback

### Return

Example	Return
watch('inputName')	unknown
watch(['inputName1'])	unknown[]
watch()	{ [key:string]: unknown }
watch((data, { name, type }) => console.log(data, name, type))	{ unsubscribe: () => void }

## Rules

- When `defaultValue` is not defined, the first render of `watch` will return `undefined` because it is called before `register`. It's **recommend** to provide `defaultValues` at `useForm` to avoid this behaviour, but you can set the inline `defaultValue` as the second argument.
- When both `defaultValue` and `defaultValues` are supplied, `defaultValue` will be returned.
- This API will trigger re-render at the root of your app or form, consider using a callback or the `useWatch` api if you are experiencing performance issues.
- `watch` result is optimised for render phase instead of `useEffect`'s deps, to detect value update you may want to use an external custom hook for value comparison.

## Examples

```
FORM

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

interface IFormInputs {
  name: string
  showAge: boolean
  age: number
}

function App() {
  const { register, watch, formState: { errors }, handleSubmit } = useForm<IFormInputs>();
  const watchShowAge = watch("showAge", false); // you can supply default value as second argument
  const watchAllFields = watch(); // when pass nothing as argument, you are watching everything
  const watchFields = watch(["showAge", "age"]); // you can also target specific fields by their names

  // Callback version of watch. It's your responsibility to unsubscribe when done.
  React.useEffect(() => {
    const subscription = watch((value, { name, type }) => console.log(value, name, type));
    return () => subscription.unsubscribe();
  }, [watch]);

  const onSubmit = (data: IFormInputs) => console.log(data);

  return (
    <>
      <form onSubmit={handleSubmit(onSubmit)}>
        <input {...register("name", { required: true, maxLength: 50 })} />
        <input type="checkbox" {...register("showAge")} />
        { /* based on yes selection to display Age Input */ }
        { watchShowAge && (
          <input type="number" {...register("age", { min: 50 })} />
        ) }
        <input type="submit" />
      </form>
    </>
  );
}
```

```

import * as React from "react";
import { useForm, useFieldArray, ArrayField } from "react-hook-form";

function App() {
  const { register, control, handleSubmit, watch } = useForm();
  const { fields, remove, append } = useFieldArray({
    name: "test",
    control
  });
  const onSubmit = (data: FormValues) => console.log(data);

  console.log(watch("test"));

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      {fields.map((field, index) => {
        return (
          <input
            key={field.id}
            defaultValue={field.firstName}
            {...register(`test[${index}].firstName`)}
          />
        );
      })}
      <button
        type="button"
        onClick={() =>
          append({
            firstName: "bill" + renderCount,
            lastName: "luo" + renderCount
          })
        }
      >
        Append
      </button>
    </form>
  );
} import * as React from "react";
import { useForm, useFieldArray, ArrayField } from "react-hook-form";

function App() {
  const { register, control, handleSubmit, watch } = useForm();
  const { fields, remove, append } = useFieldArray({
    name: "test",
    control
  });
  const onSubmit = (data: FormValues) => console.log(data);

  console.log(watch("test"));

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      {fields.map((field, index) => {
        return (

```

```

      <input
        key={field.id}
        defaultValue={field.firstName}
        {...register(`test[${index}].firstName`)}
      />
    );
  }}}
  <button
    type="button"
    onClick={() =>
      append({
        firstName: "bill" + renderCount,
        lastName: "luo" + renderCount
      })
    }
  >
    Append
  </button>
</form>
);
}

```

## handleSubmit:

```
((data: Object, e?: Event) => void, (errors: Object, e?: Event) => void)
=> Function
```

This function will receive the form data if form validation is successful.

## Props

Name	Type	Description
SubmitHandler	<code>(data: Object, e?: Event) =&gt; void</code>	A successful callback.
SubmitErrorHandler	<code>(errors: Object, e?: Event) =&gt; void</code>	An error callback.

## Rules

- You can easily submit form asynchronously with handleSubmit.

```

// It can be invoked remotely as well
handleSubmit(onSubmit());

// You can pass an async function for asynchronous validation.
handleSubmit(async (data) => await fetchAPI(data))

```

- disabled inputs will appear as undefined values in form values. If you want to prevent users from updating an input and wish to retain the form value, you can use `readOnly` or disable the entire `<fieldset />`. Here is an [example](#).
- `handleSubmit` function will not swallow errors that occurred inside your `onSubmit` callback, so we recommend you to try and catch inside async request and handle those errors gracefully for your customers.

```
const onSubmit = () => {
  // async request which may result error
  throw new Error("Something is wrong");
};

<>
<form
  onSubmit={(e) => {
    handleSubmit(onSubmit)(e)
    // you will have to catch those error and handle them
    .catch(() => {});
  }}
/>
// The following is a better approach
<form
  onSubmit={handleSubmit(() => {
    try {
      request();
    } catch (e) {
      // handle your error state here
    }
  })}
/>
</>;
```

## Examples

```
SYNC

import React from "react";
import { useForm, SubmitHandler } from "react-hook-form";

type FormValues = {
  firstName: string;
  lastName: string;
  email: string;
};

export default function App() {
  const { register, handleSubmit } = useForm<FormValues>();
  const onSubmit: SubmitHandler<FormValues> = data => console.log(data);

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register("firstName")} />
      <input {...register("lastName")} />
```



```

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

    <input type="submit" />
  </form>
);
}

```

ASYNC

```

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

const sleep = ms => new Promise(resolve => setTimeout(resolve, ms));

function App() {
  const { register, handleSubmit, formState: { errors }, formState } = useForm();
  const onSubmit = async data => {
    await sleep(2000);
    if (data.username === "bill") {
      alert(JSON.stringify(data));
    } else {
      alert("There is an error");
    }
  };

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <label htmlFor="username">User Name</label>
      <input placeholder="Bill" {...register("username")} />

      <input type="submit" />
    </form>
  );
}

```

**reset:** `<T>(values?: T | ResetAction<T>, options?: Record<string, boolean>) => void`

Reset the entire form state, fields reference, and subscriptions. There are optional arguments and will allow partial form state reset.

## Props

Reset has the ability to retain formState. Here are the options you may use:

Name	Type	Description
values	object	An optional object to reset form values, and it's recommended to provide the <b>entire</b> defaultValues when supplied.
keepErrors	boolean	All errors will remain. This will not guarantee with further user actions.
keepDirty	boolean	<b>DirtyFields</b> form state will remain, and <b>isDirty</b> will temporarily remain as the current state until further user's action.  <b>Important:</b> this keep option doesn't reflect form input values but only dirty fields form state.
keepDirtyValues	boolean	<b>DirtyFields</b> and <b>isDirty</b> will remained, and only none dirty fields will be updated to the latest rest value. <u>Check out the example.</u>  <b>Important:</b> formState dirtyFields will need to be subscribed.
keepValues	boolean	Form input values will be unchanged.
keepDefaultValues	boolean	Keep the same defaultValues which are initialised via useForm.  <ul style="list-style-type: none"> <li><b>isDirty</b> will be checked again: it is set to be the result of the comparison of any new values provided against the original <b>defaultValues</b>.</li> <li><b>dirtyFields</b> will be updated again if values are provided: it is set to be result of the comparison between the new values provided against the original defaultValues.</li> </ul>
keepIsSubmitted	boolean	<b>isSubmitted</b> state will be unchanged.
keepTouched	boolean	<b>isTouched</b> state will be unchanged.
keepIsValid	boolean	<b>isValid</b> will temporarily persist as the current state until additional user actions.
keepSubmitCount	boolean	<b>submitCount</b> state will be unchanged.

## Rules

- For controlled components you will need to pass defaultValues to useForm in order to reset the Controller components' value.

- When `defaultValues` is not supplied to `reset` API, then HTML native `reset` API will be invoked to restore the form.
- Avoid calling `reset` before `useForm`'s `useEffect` is invoked, this is because `useForm`'s subscription needs to be ready before `reset` can send a signal to flush form state update.
- It's recommended to `reset` inside `useEffect` after submission.

```
useEffect(() => {
  reset({
    data: 'test'
  })
}, [isSubmitSuccessful])
```

## Examples

### Uncontrolled

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

interface UseFormInputs {
  firstName: string
  lastName: string
}

export default function Form() {
  const { register, handleSubmit, reset, formState: { errors } } = useForm<UseFormInputs>();
  const onSubmit = (data: UseFormInputs) => {
    console.log(data)
  };

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <label>First name</label>
      <input {...register("firstName", { required: true })} />

      <label>Last name</label>
      <input {...register("lastName")} />

      <input type="submit" />
      <input
        type="reset"
        value="Standard Reset Field Values"
      />
      <input
        type="button"
        onClick={() => reset()}
        value="Custom Reset Field Values & Errors"
      />
    </form>
  );
}
```

## Controller

```
import React from "react";
import { useForm, Controller } from "react-hook-form";
import { TextField } from "@material-ui/core";

interface IFormInputs {
  firstName: string
  lastName: string
}

export default function App() {
  const { register, handleSubmit, reset, setValue, control } = useForm<IFormInputs>();
  const onSubmit = (data: IFormInputs) => console.log(data);

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <Controller
        render={({ field }) => <TextField {...field} />}
        name="firstName"
        control={control}
        rules={{ required: true }}
        defaultValue=""
      />
      <Controller
        render={({ field }) => <TextField {...field} />}
        name="lastName"
        control={control}
        defaultValue=""
      />

      <input type="submit" />
      <input type="button" onClick={reset} />
      <input
        type="button"
        onClick={() => {
          reset({
            firstName: "bill",
            lastName: "luo"
          });
        }}
      />
    </form>
  );
}
```

## Submit with Reset

```
import { useForm, useFieldArray, Controller } from "../src";
```

```

function App() {
  const {
    register,
    handleSubmit,
    reset,
    formState,
    formState: { isSubmitSuccessful }
  } = useForm({ defaultValues: { something: "anything" } });

  const onSubmit = (data) => {
    // It's recommended to reset in useEffect as execution order matters
    // reset({ ...data })
  };

  React.useEffect(() => {
    if (formState.isSubmitSuccessful) {
      reset({ something: " " });
    }
  }, [formState, submittedData, reset]);

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register("something")} />
      <input type="submit" />
    </form>
  );
}

```

## Field Array

```

import React, { useEffect } from "react";
import { useForm, useFieldArray, Controller } from "react-hook-form";

function App() {
  const { register, control, handleSubmit, reset } = useForm({
    defaultValues: {
      loadState: "unloaded",
      names: [{ firstName: "Bill", lastName: "Luo" }]
    }
  });

  const { fields, remove } = useFieldArray({
    control,
    name: "names"
  });
}

```

```

useEffect(() => {
  reset({
    names: [
      {
        firstName: "Bob",
        lastName: "Actually"
      },
      {
        firstName: "Jane",
        lastName: "Actually"
      }
    ]
  });
}, [reset]);

const onSubmit = (data) => console.log("data", data);

return (
  <form onSubmit={handleSubmit(onSubmit)}>
    <ul>
      {fields.map((item, index) => (
        <li key={item.id}>
          <input {...register(`names.${index}.firstName`)} />

          <Controller
            render={({ field }) => <input {...field} />}
            name={`names.${index}.lastName`}
            control={control}
          />
          <button type="button" onClick={() => remove(index)}>Delete</button>
        </li>
      ))}
    </ul>

    <input type="submit" />
  </form>
);
}

```

**resetField:** (name: string, options?: Record<string, boolean | any>) => void

Reset an individual field state.

## Props

After invoke this function.

- `isValid` form state will be reevaluated.
- `isDirty` form state will be reevaluated.

ResetField has the ability to retain field state. Here are the options you may want to use:

Name		Type	Description
name		string	registered field name.
options	keepError	boolean	When set to true, field error will be retained.
	keepDirty	boolean	When set to true, dirtyFields will be retained.
	keepTouched	boolean	When set to true, touchedFields state will be unchanged.
	defaultValue	unknown	When this value is not provided, field will be revert back to it's defaultValue. When this value is provided: field will be updated with the supplied value. field's defaultValue will be updated to this value.

## Rules

name need to match registered field name.

```
register('test');  
  
resetField('test'); // ☒ register input and resetField works  
resetField('non-existent-name'); // ☐ failed by input not found
```

## Examples

Reset field state

```
import * as React from "react";  
import { useForm } from "react-hook-form";  
  
export default function App() {  
  const {  
    register,  
    resetField,  
    formState: { isDirty, isValid }  
  } = useForm({  
    mode: "onChange",  
    defaultValues: {  
      firstName: ""  
    }  
  });  
  const handleClick = () => resetField("firstName");  
  
  return (  
    <form>  
      <input {...register("firstName", { required: true })} />  
  
      <p>{isDirty && "dirty"}</p>  
      <p>{isValid && "valid"}</p>  
    </form>  
  );  
}
```

```

    <button type="button" onClick={handleClick}>
      Reset
    </button>
  </form>
);
}

```

### Reset with options

```

import * as React from "react";
import { useForm } from "react-hook-form";

export default function App() {
  const {
    register,
    resetField,
    formState: { isDirty, isValid }
  } = useForm({
    mode: "onChange",
    defaultValues: {
      firstName: ""
    }
  });
  const handleClick = () => resetField("firstName");

  return (
    <form>
      <input {...register("firstName", { required: true })} />

      <p>{isDirty && "dirty"}</p>
      <p>{isValid && "valid"}</p>

      <button type="button" onClick={handleClick}>
        Reset
      </button>
    </form>
  );
}

```

**setError:** (name: string, error: FieldError, { shouldFocus?: boolean }) => void

The function allows you to manually set one or more errors.

## Props



Name	Type	Description
name	string	input's name.
error	{ type: string, message?: string, types: MultipleFieldErrors }	Set an error with its type and message.
config	{ shouldFocus?: boolean }	Should focus the input during setting an error. This only works when the input's reference is registered, it will not work for custom register as well.

## Rules

- This method will not persist the associated input error if the input passes register's associated rules.

```
register('registerInput', { minLength: 4 });
setError('registerInput', { type: 'custom', message: 'custom message' });
// validation will pass as long as minLength requirement pass
```

- An error that is not associated with an input field will be persisted until cleared with `clearErrors`.

```
setError('notRegisteredInput', { type: 'custom', message: 'custom message' });
// clearErrors() need to invoked manually to remove that custom error
```

- Can be useful in the `handleSubmit` method when you want to give error feedback to a user after async validation. (ex: API returns validation errors)
- `shouldFocus` doesn't work when an input has been disabled.
- This method will force set `isValid` formState to `false`, however, it's important to aware `isValid` will always be derived by the validation result from your input registration rules or schema result.

## Examples

### Single error

```
import * as React from "react";
import { useForm } from "react-hook-form";

type FormInputs = {
  username: string;
};

const App = () => {
  const { register, handleSubmit, setError, formState: { errors } } = useForm<FormInputs>();
  const onSubmit = (data: FormInputs) => {
    console.log(data)
  };
};
```

```

React.useEffect(() => {
  setError("username", {
    type: "manual",
    message: "Dont Forget Your Username Should Be Cool!"
  });
}, [setError])

return (
  <form onSubmit={handleSubmit(onSubmit)}>
    <input {...register("username")} />
    {errors.username && <p>{errors.username.message}</p>}

    <input type="submit" />
  </form>
);
};

```

## Multiple errors

```

import * as React from "react";
import { useForm } from "react-hook-form";

type FormInputs = {
  username: string;
  firstName: string;
};

const App = () => {
  const { register, handleSubmit, setError, formState: { errors } } = useForm<FormInputs>();

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

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <label>Username</label>
      <input {...register("username")} />
      {errors.username && <p>{errors.username.message}</p>}
      <label>First Name</label>
      <input {...register("firstName")} />
      {errors.firstName && <p>{errors.firstName.message}</p>}
      <button
        type="button"
        onClick={() => {
          [
            {
              type: "manual",
              name: "username",
              message: "Double Check This"
            },
            {

```

```

        type: "manual",
        name: "firstName",
        message: "Triple Check This"
      }
    ].forEach(({ name, type, message }) =>
      setError(name, { type, message })
    );
  }}
>
  Trigger Name Errors
</button>
<input type="submit" />
</form>
);
};

```

### Single Field errors

```

import * as React from "react";
import { useForm } from "react-hook-form";

type FormInputs = {
  lastName: string;
};

const App = () => {
  const { register, handleSubmit, setError, formState: { errors } } = useForm<FormInputs>({
    criteriaMode: 'all',
  });

  const onSubmit = (data: FormInputs) => console.log(data);

  React.useEffect(() => {
    setError("lastName", {
      types: {
        required: "This is required",
        minLength: "This is minLength"
      }
    });
  }, [setValue])

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <label>Last Name</label>
      <input {...register("lastName")} />
      {errors.lastName && errors.lastName.types && (
        <p>{errors.lastName.types.required}</p>
      )}
      {errors.lastName && errors.lastName.types && (
        <p>{errors.lastName.types.minLength}</p>
      )}
      <input type="submit" />
    </form>);
};

```

**clearErrors:** (name?: string | string[]) => void

This function can manually clear errors in the form.

## Props

Type	Description	Example
undefined	Remove all errors.	<code>clearErrors()</code>
string	Remove single error.	<code>clearErrors("yourDetails.firstName")</code>
string[]	Remove multiple errors.	<code>clearErrors(["yourDetails.lastName"])</code>

- undefined: reset all errors
- string: reset the error on a single field or by key name.

```
register('test.firstName', { required: true });
register('test.lastName', { required: true });
clearErrors('test'); // will clear both errors from test.firstName and test.lastName
clearErrors('test.firstName'); // for clear single input error
```

- string[]: reset errors on the given fields

## Rules

- This will not affect the validation rules attached to each inputs.
- This method doesn't affect validation rules or isValid formState.

## Example

```
import * as React from "react";
import { useForm } from "react-hook-form";

type FormInputs = {
  firstName: string;
  lastName: string;
  username: string;
};

const App = () => {
  const { register, formState: { errors }, handleSubmit, clearErrors } = useForm<FormInputs>();

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

```

return (
  <form onSubmit={handleSubmit(onSubmit)}>
    <input {...register('firstName', { required: true })} />
    <input {...register('lastName', { required: true })} />
    <input {...register('username', { required: true })} />
    <button type="button" onClick={() => clearErrors("firstName")}>
      Clear First Name Errors
    </button>
    <button
      type="button"
      onClick={() => clearErrors(["firstName", "lastName"])}
    >
      Clear First and Last Name Errors
    </button>
    <button type="button" onClick={() => clearErrors()}>
      Clear All Errors
    </button>
    <input type="submit" />
  </form>
);
};

```

**setValue:** (name: string, value: unknown, config?: Object) => void

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.

## Props

Name	Type	Description
name	string	<p>Target a single field by name.</p> <p>When used with field array.</p> <p>You can use methods such as <a href="#">replace</a> or <a href="#">update</a> for field array, however, they will cause the component to unmount and remount for the targeted field array.</p> <pre>const { update } = useFieldArray({ name: 'array' });</pre> <pre>// unmount fields and remount with updated value update(0, { test: '1', test1: '2' })</pre> <pre>// will directly update input value setValue('array.0.test1', '1'); setValue('array.0.test2', '2');</pre> <p>It will not create a new field when targeting a non-existing field.</p>

			<pre>const { replace } = useFieldArray({ name: 'test' })  // ✗ doesn't create new input setValue('test.101.data')  // ✔ work on refresh entire field array replace([{data: 'test'}])</pre>
value		unknown	The value for the field. This argument is required and can not be undefined.
options	shouldValidate	boolean	<p>Whether to compute if your input is valid or not (subscribed to <code>errors</code>).</p> <p>Whether to compute if your entire form is valid or not (subscribed to <code>isValid</code>).</p> <p>This option will update <code>touchedFields</code> at the specified field level not the entire form touched fields.</p> <pre>setValue('name', 'value', { shouldValidate: true })</pre>
	shouldDirty	boolean	<p>Whether to compute if your input is dirty or not against your <code>defaultValues</code> (subscribed to <code>dirtyFields</code>).</p> <p>Whether to compute if your entire form is dirty or not against your <code>defaultValues</code> (subscribed to <code>isDirty</code>).</p> <p>This option will update <code>dirtyFields</code> at the specified field level not the entire form dirty fields.</p> <pre>setValue('name', 'value', { shouldDirty: true })</pre>
	shouldTouch	boolean	<p>Whether to set the input itself to be touched.</p> <pre>setValue('name', 'value', { shouldTouch: true })</pre>

## Rules

- Only the following conditions will trigger a re-render:
  - When an error is triggered or corrected by a value update
  - When `setValue` cause state update, such as dirty and touched.
- It's recommended to target the field's name rather than make the second argument a nested object.

```
setValue('yourDetails.firstName', 'value'); // ✔ performant
setValue('yourDetails', { firstName: 'value' }); // less performant

register('nestedValue', { value: { test: 'data' } }); // register a nested value input
setValue('nestedValue.test', 'updatedData'); // ✗ failed to find the relevant field
setValue('nestedValue', { test: 'updatedData' } ); // ✔ setValue find input and update
```

- It's recommended to register the input's name before invoking `setValue`. To update the entire Field Array, make sure the `useFieldArray` hook is being executed first.

**Important:** use replace from `useFieldArray` instead, update entire field array with `setValue` will be removed in the next major version.

```
// you can update an entire Field Array,
setValue('fieldArray', [{ test: '1' }, { test: '2' }]); // ✅

// you can setValue to a unregistered input
setValue('notRegisteredInput', 'value'); // ✅ prefer to be registered

// the following will register a single input (without register invoked)
setValue('resultSingleNestedField', { test: '1', test2: '2' }); // 🤔

// with registered inputs, the setValue will update both inputs correctly.
register('notRegisteredInput.test', '1')
register('notRegisteredInput.test2', '2')
setValue('notRegisteredInput', { test: '1', test2: '2' }); // ✅ sugar syntax to setValue twice
```

## Examples

### 1) Basic

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

const App = () => {
  const { register, setValue } = useForm({
    firstName: ""
  });

  return (
    <form>
      <input {...register("firstName", { required: true })} />
      <button onClick={() => setValue("firstName", "Bill")}>
        setValue
      </button>
      <button
        onClick={() =>
          setValue("firstName", "Luo", {
            shouldValidate: true,
            shouldDirty: true
          })
        }
      >
        setValue options
      </button>
    </form>
  );
};
```

```

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

type FormValues = {
  string: string;
  number: number;
  object: {
    number: number;
    boolean: boolean;
  };
  array: {
    string: string;
    boolean: boolean;
  }[];
};

export default function App() {
  const { setValue } = useForm<FormValues>();

  setValue("string", "test");
  // function setValue<"string", string>(name: "string", value: string, shouldValidate?: boolean |
undefined): void
  setValue("number", 1);
  // function setValue<"number", number>(name: "number", value: number, shouldValidate?: boolean |
undefined): void
  setValue("number", "error");

  return <form />;
}

```

## 2) Dependant fields

```

import * as React from "react";
import { useForm } from "react-hook-form";

type FormValues = {
  a: string;
  b: string;
  c: string;
};

export default function App() {
  const { watch, register, handleSubmit, setValue, formState } = useForm<
  FormValues
>({
  defaultValues: {
    a: "",
    b: "",
    c: ""
  }
});

```



```

const onSubmit = (data: FormValues) => console.log(data);
const [a, b] = watch(["a", "b"]);

React.useEffect(() => {
  if (formState.touchedFields.a && formState.touchedFields.b && a && b) {
    setValue("c", `${a} ${b}`);
  }
}, [setValue, a, b, formState]);

return (
  <form onSubmit={handleSubmit(onSubmit)}>
    <input {...register("a")} placeholder="a" />
    <input {...register("b")} placeholder="b" />
    <input {...register("c")} placeholder="c" />
    <input type="submit" />

    <button
      type="button"
      onClick={() => {
        setValue("a", "what", { shouldTouch: true });
        setValue("b", "ever", { shouldTouch: true });
      }}
    >
      trigger value
    </button>
  </form>
);
}

```

**setFocus:** (name: string, options: SetFocusOptions) => void

This method will allow users to programmatically focus on input. Make sure input's ref is registered into the hook form.

## Props

Name		Type	Description
name		string	A input field name to focus
options	shouldSelect	boolean	Whether to select the input content on focus. COPY const { setFocus } = useForm() setFocus("name", { shouldSelect: true })

## Rules

- This API will invoke focus method from the ref, so it's important to provide ref during register.

- Avoid calling `setFocus` right after `reset` as all input references will be removed by `reset` API.

## Examples

```
import * as React from "react";
import { useForm } from "./src";

type FormValues = {
  firstName: string;
};

export default function App() {
  const { register, handleSubmit, setFocus } = useForm<FormValues>();
  const onSubmit = (data: FormValues) => console.log(data);
  renderCount++;

  React.useEffect(() => {
    setFocus("firstName");
  }, [setFocus]);

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register("firstName")} placeholder="First Name" />
      <input type="submit" />
    </form>
  );
}
```

**getValues:** (payload?: string | string[]) => Object

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.

## Props

Type	Description
<code>undefined</code>	Returns the entire form values.
<code>string</code>	Gets the value at path of the form values.
<code>array</code>	Returns an array of the value at path of the form values.

Example

The example below shows what to expect when you invoke `getValues` method.

<code>&lt;input {...register('root.test1')} /&gt;</code>
<code>&lt;input {...register('root.test2')} /&gt;</code>

Name	Output
getValues()	{ root: { test1: '', test2: '' } }
getValues("yourDetails")	{ test1: '', test2: '' }
getValues("yourDetails.firstName")	{ test1: '' }
getValues(["yourDetails.lastName"])	[ ' ' ]

## Rules

- Disabled inputs will be returned as undefined. If you want to prevent users from updating the input and still retain the field value, you can use `readOnly` or disable the entire `<fieldset />`. Here is an [example](#).
- It will return `defaultValues` from `useForm` before the **initial** render.

## Example

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

type FormInputs = {
  test: string
  test1: string
}

export default function App() {
  const { register, getValues } = useForm<FormInputs>();

  return (
    <form>
      <input {...register("test")} />
      <input {...register("test1")} />

      <button
        type="button"
        onClick={() => {
          const values = getValues(); // { test: "test-input", test1: "test1-input" }
          const singleValue = getValues("test"); // "test-input"
          const multipleValues = getValues(["test", "test1"]);
          // ["test-input", "test1-input"]
        }}
      >
        Get Values
      </button>
    </form>
  );
}
```

+ types

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

// Flat input values
type Inputs = {
  key1: string;
  key2: number;
  key3: boolean;
  key4: Date;
};

export default function App() {
  const { register, getValues } = useForm<Inputs>();

  getValues();

  return <form />;
}

// Nested input values
type Inputs1 = {
  key1: string;
  key2: number;
  key3: {
    key1: number;
    key2: boolean;
  };
  key4: string[];
};

export default function Form() {
  const { register, getValues } = useForm<Inputs1>();

  getValues();
  // function getValues(): Record<string, unknown>
  getValues("key1");
  // function getValues<"key1", unknown>(payload: "key1"): string
  getValues("key2");
  // function getValues<"key2", unknown>(payload: "key2"): number
  getValues("key3.key1");
  // function getValues<"key3.key1", unknown>(payload: "key3.key1"): unknown
  getValues<string, number>("key3.key1");
  // function getValues<string, number>(payload: string): number
  getValues<string, boolean>("key3.key2");
  // function getValues<string, boolean>(payload: string): boolean
  getValues("key4");
  // function getValues<"key4", unknown>(payload: "key4"): string[]

  return <form />;
}
```

**getFieldState:** (name: string, formState?: Object) => ({isDirty, isTouched, invalid, error})

This method is introduced in react-hook-form ([v7.25.0](#)) to return individual field state. It's useful in case you are trying to retrieve nested field state in a typesafe way.

## Props

Name	Type	Description
name	string	registered field name.
formState	object	This is an optional prop, which is only required if formState is not been read/subscribed from the useForm, useFormContext or useFormState. COPY <pre>const methods = useForm(); // not subscribed to any formState const { error } = getFieldState('firstName', methods.formState) // It is subscribed now and reactive to error state updated  const { formState: { errors } } = useForm() // errors are subscribed and reactive to state update getFieldState('firstName') // return updated field error state</pre>

## Return

Name	Type	Description
isDirty	boolean	field is modified. Condition: subscribe to dirtyFields.
isTouched	boolean	field has received a focus and blur event. Condition: subscribe to touchedFields.
invalid	boolean	field is not valid. Condition: subscribe to errors.
error	undefined   FieldError	field error object. Condition: subscribe to errors.

## Rules

- name needs to match a registered field name.

```
getFieldState('test');

getFieldState('test'); // ☒ register input and return field state
getFieldState('non-existent-name'); // ☐ will return state as false and error as undefined
```

- getFieldState works by subscribing to the form state update, and you can subscribe to the formState in the following ways:
  - You can subscribe at the useForm, useFormContext or useFormState. This is will establish the form state subscription and getFieldState second argument will no longer be required.

```
const { register, formState: { isDirty } } = useForm()
register('test');
getFieldState('test'); // ✓
```

```
const { isDirty } = useFormState();
register('test');
getFieldState('test'); // ✓
```

```
const { register, formState: { isDirty } } = useFormContext();
register('test');
getFieldState('test'); // ✓
```

- When form state subscription is not setup, you can pass the entire formState as the second optional argument by following the example below:

```
const { register } = useForm()
register('test');
const { isDirty } = getFieldState('test'); // ✗ formState.isDirty is not subscribed at useForm

const { register, formState } = useForm()
const { isDirty } = getFieldState('test', formState); // ✓ formState.isDirty subscribed

const { formState } = useFormContext();
const { touchedFields } = getFieldState('test', formState); // ✓ formState.touchedFields subscribed
```

## Example

```
import * as React from "react";
import { useForm } from "react-hook-form";

export default function App() {
  const {
    register,
    getFieldState,
    formState: { isDirty, isValid }
  } = useForm({
    mode: "onChange",
    defaultValues: {
      firstName: ""
    }
  });

  // you can invoke before render or within the render function
  const fieldState = getFieldState("firstName");
```

```

return (
  <form>
    <input {...register("firstName", { required: true })} />
    <p>{getFieldState("firstName").isDirty && "dirty"}</p>
    <p>{getFieldState("firstName").isTouched && "touched"}</p>
    <button type="button" onClick={() => console.log(getFieldState("firstName"))}>
      field state
    </button>
  </form>
);
}

```

**trigger:** (name?: string | string[]) => Promise<boolean>

Manually triggers form or input validation. This method is also useful when you have dependant validation (input validation depends on another input's value).

## Props

Name	Type	Description	Example
name	undefined	Triggers validation on all fields.	trigger()
	string	Triggers validation on a specific field value by name	trigger("yourDetails.firstName")
	string[]	Triggers validation on multiple fields by name.	trigger(["yourDetails.lastName"])
shouldFocus	boolean	Should focus the input during setting an error. This only works when the input's reference is registered, it will not work for custom register as well.	trigger('name', { shouldFocus: true })

## Rules

Isolate render optimisation only applicable for targeting a single field name with string as payload, when supplied with array and undefined to trigger will re-render the entire formState.

## Example

```

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

type FormInputs = {
  firstName: string
  lastName: string
}

export default function App() {
  const { register, trigger, formState: { errors } } = useForm<FormInputs>();

  return (
    <form>
      <input {...register("firstName", { required: true })} />
      <input {...register("lastName", { required: true })} />
      <button type="button" onClick={() => { trigger("lastName"); }}>Trigger</button>
      <button type="button" onClick={() => { trigger(["firstName", "lastName"]); }}>Trigger
Multiple</button>
      <button type="button" onClick={() => { trigger(); }}>Trigger All</button>
    </form>
  );
}

```

## control: Object

This object contains methods for registering components into React Hook Form.

### Rules

**Important:** do not access any of the properties inside this object directly. It's for internal usage only.

### Example

```

import React from "react";
import { useForm, Controller } from "react-hook-form";
import { TextField } from "@material-ui/core";

type FormInputs = {
  firstName: string
}

function App() {
  const { control, handleSubmit } = useForm<FormInputs>();
  const onSubmit = (data: FormInputs) => console.log(data);

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <Controller
        as={TextField}
        name="firstName"

```



```
        control={control}
        defaultValue=""
    />

    <input type="submit" />
</form>
);
}
```

# useController

React hooks for controlled component

## useController:

```
(props?: UseControllerProps) => { field: object, fieldState: object,
formState: object }
```

This custom hook powers Controller. Additionally, it shares the same props and methods as Controller. It's useful for creating reusable Controlled input.

## Props

The following table contains information about the arguments for useController.

Name	Type	Required	Description
name	<u>FieldPath</u>	✓	Unique name of your input.
control	<u>Control</u>		<u>control</u> object provided by invoking useForm. Optional when using FormProvider.
defaultValue	unknown		Important: Can not apply undefined to defaultValue or defaultValues at useForm. You need to either set defaultValue at the field-level or useForm's defaultValues. undefined is not a valid value. If your form will invoke reset with default values, you will need to provide useForm with defaultValues.
rules	Object		Validation rules in the same format for register, which includes: required, min, max, minLength, maxLength, pattern, validate <u>CODESANDBOX</u> <sup>TS</sup> rules={{ required: true }}

Name	Type	Required	Description
shouldUnregister	boolean = false		Input will be unregistered after unmount and defaultValues will be removed as well. Note: this prop should be avoided when using with useFieldArray as unregister function gets called after input unmount/remount and reorder.

## Return

The following table contains information about properties which useController produces.

Object Name	Name	Type	Description
field	onChange	(value: any) => void	A function which sends the input's value to the library. It should be assigned to the onChange prop of the input and value should not be <b>undefined</b> . This prop update <u>formState</u> and you should avoid manually invoke <u>setValue</u> or other API related to field update.
field	onBlur	() => void	A function which sends the input's onBlur event to the library. It should be assigned to the input's onBlur prop.
field	value	unknown	The current value of the controlled component.
field	name		
string			Input's name being registered.
field	ref		
React.Ref			A ref used to connect hook form to the input. Assign ref to component's input ref to allow hook form to focus the error input.
fieldState	invalid	boolean	Invalid state for current input.
fieldState	isTouched	boolean	Touched state for current controlled input.

Object Name	Name	Type	Description
fieldState	isDirty	boolean	Dirty state for current controlled input.
fieldState	error	object	error for this specific input.
formState	isDirty	boolean	<p>Set to <code>true</code> after the user modifies any of the inputs.</p> <p>Make sure to provide all inputs' <code>defaultValues</code> at the <code>useForm</code>, so hook form can have a single source of truth to compare whether the form is dirty.</p> <p><b>COPY</b></p> <pre>const {   formState: { isDirty,     dirtyFields },   setValue, } = useForm({ defaultValues: {   test: "" } });  // isDirty: true setValue('test', 'change')</pre> <p>// isDirty: false because there  <code>getValues() === defaultValues</code>  <code>setValue('test', '')</code>  File typed input will need to be managed at the app level due to the ability to cancel file selection and <a href="#">FileList</a> object.</p>
formState	dirtyFields	object	<p>An object with the user-modified fields.</p> <p>Make sure to provide all inputs' <code>defaultValues</code> via <code>useForm</code>, so the library can compare against the <code>defaultValues</code>.</p> <p>Dirty fields will <b>not</b> represent as <code>isDirty</code> <code>formState</code>, because dirty fields are marked field dirty at field level rather the entire form. If you want to determine the entire form state use <code>isDirty</code> instead.</p>
formState	touchedFields	object	An object containing all the inputs the user has interacted with.
formState	defaultValues	object	The value which has been set at <code>useForm</code> 's <code>defaultValues</code> or updated <code>defaultValues</code> via <code>reset</code> API.
	isSubmitted	boolean	<p>Set to <code>true</code> after the form is submitted.</p> <p>Will remain <code>true</code> until the <code>reset</code> method is invoked.</p>
formState	isSubmitSuccessful	boolean	Indicate the form was successfully submitted without any <code>Promise</code> rejection or <code>Error</code> been

Object Name	Name	Type	Description
			thrown within the <code>handleSubmit</code> callback.
formState	isSubmitting	boolean	true if the form is currently being submitted. false otherwise.
formState	submitCount	number	Number of times the form was submitted.
formState	isValid	boolean	Set to true if the form doesn't have any errors. <code>isValid</code> is affected by mode at <code>useForm</code> . This state is only applicable with <code>onChange</code> , <code>onTouched</code> , and <code>onBlur</code> mode. <code>setError</code> has no effect on <code>isValid</code> formState, <code>isValid</code> will always derived via the entire form validation result.
formState	isValidating	boolean	Set to true during validation.
formState	errors	object	An object with field errors. There is also an <code>ErrorMessage</code> component to retrieve error message easily.

## Examples

### Textfield

```
import * as React from "react";
import { useForm, useController, UseControllerProps } from "react-hook-form";

type FormValues = {
  FirstName: string;
};

function Input(props: UseControllerProps<FormValues>) {
  const { field, fieldState } = useController(props);

  return (
    <div>
      <input {...field} placeholder={props.name} />
      <p>{fieldState.isTouched && "Touched"}</p>
      <p>{fieldState.isDirty && "Dirty"}</p>
      <p>{fieldState.invalid ? "invalid" : "valid"}</p>
    </div>
  );
}

export default function App() {
  const { handleSubmit, control } = useForm<FormValues>({
    defaultValues: {
```

```

    FirstName: ""
  },
  mode: "onChange"
});
const onSubmit = (data: FormValues) => console.log(data);

return (
  <form onSubmit={handleSubmit(onSubmit)}>
    <Input control={control} name="FirstName" rules={{ required: true }} />
    <input type="submit" />
  </form>
);
}

```

```

import * as React from "react";
import { useController, useForm } from "react-hook-form";

const Checkboxes = ({ options, control, name }) => {
  const { field } = useController({
    control,
    name
  });
  const [value, setValue] = React.useState(field.value || []);

  return (
    <>
      {options.map((option, index) => (
        <input
          onChange={(e) => {
            const valueCopy = [...value];

            // update checkbox value
            valueCopy[index] = e.target.checked ? e.target.value : null;

            // send data to react hook form
            field.onChange(valueCopy);

            // update local state
            setValue(valueCopy);
          }}
          key={option}
          checked={value.includes(option)}
          type="checkbox"
          value={option}
        />
      ))}
    </>
  );
};

export default function App() {
  const { register, handleSubmit, control } = useForm({

```

```

    defaultValues: {
      controlled: [],
      uncontrolled: []
    }
  });
  const onSubmit = (data) => console.log(data);

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <section>
        <h2>uncontrolled</h2>
        <input {...register("uncontrolled")} type="checkbox" value="A" />
        <input {...register("uncontrolled")} type="checkbox" value="B" />
        <input {...register("uncontrolled")} type="checkbox" value="C" />
      </section>

      <section>
        <h2>controlled</h2>
        <Checkboxes
          options={["a", "b", "c"]}
          control={control}
          name="controlled"
        />
      </section>
      <input type="submit" />
    </form>
  );
}

```

## Tips

- It's important to be aware of each prop's responsibility when working with external controlled components, such as MUI, AntD, Chakra UI. Its job is to spy on the input, report, and set its value.
  - **onChange**: send data back to hook form
  - **onBlur**: report input has been interacted (focus and blur)
  - **value**: set up input initial and updated value
  - **ref**: allow input to be focused with error
  - **name**: give input an unique name

It's fine to host your state and combined with useController.

```

const { field } = useController();
const [value, setValue] = useState(field.value);

onChange={(event) => {
  field.onChange(parseInt(event.target.value)) // data send back to hook form
  setValue(event.target.value) // UI state
}}

```

Do not **register** input again. This custom hook is designed to take care of the registration process.

```
const { field } = useController({ name: 'test' })

<input {...field} /> // ✓
<input {...field} {...register('test')} /> // ✗ double up the registration
```

It's ideal to use a single `useController` per component. If you need to use more than one, make sure you rename the prop. May want to consider using `Controller` instead.

```
const { field: input } = useController({ name: 'test' })
const { field: checkbox } = useController({ name: 'test1' })

<input {...input} />
<input {...checkbox} />
```

## Controller: Component

React Hook Form embraces uncontrolled components and native inputs, however it's hard to avoid working with external controlled component such as [React-Select](#), [AntD](#) and [MUI](#). This wrapper component will make it easier for you to work with them.

## Props

The following table contains information about the arguments for `useController`.

Name	Type	Required	Description
name	<u>FieldPath</u>	✓	Unique name of your input.
control	<u>Control</u>		<u>control</u> object is from invoking <code>useForm</code> . Optional when using <code>FormProvider</code> .
render	Function		This is a <u>render prop</u> . A function that returns a React element and provides the ability to attach events and value into the component. This simplifies integrating with external controlled components with non-standard prop names. Provides <code>onChange</code> , <code>onBlur</code> , <code>name</code> , <code>ref</code> and <code>value</code> to the child component, and also a <code>fieldState</code> object which contains specific input state.

```

<Controller
  control={control}
  name="test"
  render={({
    field: { onChange, onBlur, value, name, ref },
    fieldState: { invalid, isTouched, isDirty, error },
    formState,
  }) => (
    <Checkbox
      onBlur={onBlur} // notify when input is touched
      onChange={onChange} // send value to hook form
      checked={value}
      inputRef={ref}
    />
  )}
/>

```

```

<Controller
  render={({
    field: { onChange, onBlur, value, name, ref },
    fieldState: { invalid, isTouched, isDirty, error },
  }) => (
    <TextField
      value={value}
      onChange={onChange} // send value to hook form
      onBlur={onBlur} // notify when input is touched
      inputRef={ref} // wire up the input ref
    />
  )}
  name="TextField"
  control={control}
  rules={{ required: true }}
/>

```

defaultValue	unknown	<p><b>Important:</b> Can not apply undefined to defaultValue or defaultValues at useForm.</p> <p>You need to either set defaultValue at the field-level or useForm's defaultValues. undefined is not a valid value.</p> <p>If your form will invoke reset with default values, you will need to provide useForm with defaultValues.</p> <p>Calling onChange with undefined is not valid. You should use null or the empty string as your default/cleared value instead.</p>
rules	Object	<p>Validation rules in the same format for <u>register options</u>, which includes:</p> <p>required, min, max, minLength, maxLength, pattern, validate</p> <p><u>CODESANDBOX<sup>TS</sup></u></p> <pre>rules={{ required: true }}</pre>
shouldUnregister	boolean = false	<p>Input will be unregistered after unmount and defaultValues will be removed as well.</p>



			<b>Note:</b> this prop should be avoided when using <code>useFieldArray</code> as <code>unregister</code> function gets called after input unmount/remount and reorder.
--	--	--	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------

## Return

The following table contains information about properties which `Controller` produces.

Object Name	Name	Type	Description
field	onChange	<code>(value: any) =&gt; void</code>	A function which sends the input's value to the library. It should be assigned to the <code>onChange</code> prop of the input and value should not be <b>undefined</b> . This prop update <code>formState</code> and you should avoid manually invoke <code>setValue</code> or other API related to field update.
field	onBlur	<code>() =&gt; void</code>	A function which sends the input's <code>onBlur</code> event to the library. It should be assigned to the input's <code>onBlur</code> prop.
field	value	unknown	The current value of the controlled component.
field	name		
string			Input's name being registered.
field	ref		
React.Ref			A ref used to connect hook form to the input. Assign <code>ref</code> to component's input ref to allow hook form to focus the error input.
fieldState	invalid	boolean	Invalid state for current input.
fieldState	isTouched	boolean	Touched state for current controlled input.
fieldState	isDirty	boolean	Dirty state for current controlled input.
fieldState	error	object	error for this specific input.
formState	isDirty	boolean	Set to <code>true</code> after the user modifies any of the inputs.

Object Name	Name	Type	Description
			<p>Make sure to provide all inputs' defaultValues at the useForm, so hook form can have a single source of truth to compare whether the form is dirty.</p> <pre>COPY const {   formState: { isDirty, dirtyFields },   setValue, } = useForm({ defaultValues: { test: "" } });  // isDirty: true setValue('test', 'change')  // isDirty: false because there getValues() === defaultValues setValue('test', '')</pre> <p>File typed input will need to be managed at the app level due to the ability to cancel file selection and <a href="#">FileList</a> object.</p>
formState	dirtyFields	object	<p>An object with the user-modified fields. Make sure to provide all inputs' defaultValues via useForm, so the library can compare against the defaultValues.</p> <p>Dirty fields will <b>not</b> represent as isDirty formState, because dirty fields are marked field dirty at field level rather the entire form. If you want to determine the entire form state use isDirty instead.</p>
formState	touchedFields	object	An object containing all the inputs the user has interacted with.
formState	defaultValues	object	The value which has been set at <a href="#">useForm</a> 's defaultValues or updated defaultValues via <a href="#">reset</a> API.
	isSubmitted	boolean	Set to true after the form is submitted. Will remain true until the reset method is invoked.
formState	isSubmitSuccessful	boolean	Indicate the form was successfully submitted without any Promise rejection or Error been thrown within the handleSubmit callback.
formState	isSubmitting	boolean	true if the form is currently being submitted. false otherwise.

Object Name	Name	Type	Description
formState	submitCount	number	Number of times the form was submitted.
formState	isValid	boolean	Set to true if the form doesn't have any errors. <b>isValid</b> is affected by mode at <a href="#">useForm</a> . This state is only applicable with <code>onChange</code> , <code>onTouched</code> , and <code>onBlur</code> mode. <code>setError</code> has no effect on <code>isValid</code> formState, <code>isValid</code> will always derived via the entire form validation result.
formState	isValidating	boolean	Set to true during validation.
formState	errors	object	An object with field errors. There is also an <a href="#">ErrorMessage</a> component to retrieve error message easily.

## Examples

```
import ReactDatePicker from "react-datepicker";
import { TextField } from "@material-ui/core";
import { useForm, Controller } from "react-hook-form";

type FormValues = {
  ReactDatePicker: string;
}

function App() {
  const { handleSubmit, control } = useForm<FormValues>();

  return (
    <form onSubmit={handleSubmit(data => console.log(data))}>
      <Controller
        control={control}
        name="ReactDatePicker"
        render={({ field: { onChange, onBlur, value, ref } }) => (
          <ReactDatePicker
            onChange={onChange} // send value to hook form
            onBlur={onBlur} // notify when input is touched/blur
            selected={value}
          />
        )}
      />

      <input type="submit" />
    </form>
  );
}
```

## Tips

- It's important to be aware of each prop's responsibility when working with external controlled components, such as MUI, AntD, Chakra UI. Controller acts as a "spy" on your input by reporting and setting value.
  - **onChange**: send data back to hook form
  - **onBlur**: report input has been interacted (focus and blur)
  - **value**: set up input initial and updated value
  - **ref**: allow input to be focused with error
  - **name**: give input an unique name

The following codesandbox demonstrate the usages:

- [MUI and other components](#)
- [Chakra UI components](#)

Do not register input again. This component is made to take care of the registration process.

```
<Controller
  name="test"
  render={({ field }) => {
    // return <input {...field} {...register('test')} />; ✗ double up the registration
    return <input {...field} />; // ✓
  }}
/>
```

Customise what value gets sent to hook form by transforming the value during onChange.

```
<Controller
  name="test"
  render={({ field }) => {
    // sending integer instead of string.
    return <input {...field} onChange={(e) => field.onChange(parseInt(e.target.value))} />;
  }}
/>
```

# useFormContext

React Context API for hook form

**useFormContext:** `Function`

This custom hook allows you to access the form context. `useFormContext` is intended to be used in deeply nested structures, where it would become inconvenient to pass the context as a prop.

## Return

This hook will return all the `useForm` return methods and props.

```
const methods = useForm()

<FormProvider {...methods} /> // all the useForm return props

const methods = useFormContext() // retrieve those props
```

## Rules

You need to wrap your form with the `FormProvider` component for `useFormContext` to work properly.

## Example

```
import React from "react";
import { useForm, FormProvider, useFormContext } from "react-hook-form";

export default function App() {
  const methods = useForm();
  const onSubmit = data => console.log(data);

  return (
    <FormProvider {...methods} > // pass all methods into the context
      <form onSubmit={methods.handleSubmit(onSubmit)}>
        <NestedInput />
        <input type="submit" />
      </form>
    </FormProvider>
  );
}

function NestedInput() {
  const { register } = useFormContext(); // retrieve all hook methods
  return <input {...register("test")} />;
}
```

# FormProvider

A component to provide React Context

This component will host context object and allow consuming component to subscribe to context and use useForm props and methods.

## Props

This following table applied to `FormProvider`, `useFormContext` accepts no argument.

Name	Type	Description
<code>...props</code>	Object	<code>FormProvider</code> requires all <code>useForm</code> methods.

## Rules

- Avoid using nested `FormProvider`

## Example

```
import React from "react";
import { useForm, FormProvider, useFormContext } from "react-hook-form";

export default function App() {
  const methods = useForm();
  const onSubmit = data => console.log(data);

  return (
    <FormProvider {...methods}> // pass all methods into the context
      <form onSubmit={methods.handleSubmit(onSubmit)}>
        <NestedInput />
        <input type="submit" />
      </form>
    </FormProvider>
  );
}

function NestedInput() {
  const { register } = useFormContext(); // retrieve all hook methods
  return <input {...register("test")} />;
}
```

# useWatch

React Hook for subscribe to input changes

## useWatch:

```
({ control?: Control, name?: string, defaultValue?: unknown, disabled?: boolean }) => object
```

Behaves similarly to the watch API, however, this will isolate re-rendering at the custom hook level and potentially result in better performance for your application.

## Props

Name	Type	Description
name	<code>string   string[]   undefined</code>	Name of the field.
control	<code>Object</code>	<code>control</code> object provided by <code>useForm</code> . It's optional if you are using <code>FormContext</code> .
defaultValue	<code>unknown</code>	default value for <code>useWatch</code> to return before the initial render. Note: the first render will always return <code>defaultValue</code> when it's supplied.
disabled	<code>boolean = false</code>	Option to disable the subscription.
exact	<code>boolean = false</code>	This prop will enable an exact match for input name subscriptions.

## Return

Example	Return
<code>useWatch('inputName')</code>	<code>unknown</code>
<code>useWatch(['inputName1'])</code>	<code>unknown[]</code>
<code>useWatch()</code>	<code>{[key:string]: unknown}</code>

## Rules

- The initial return value from `useWatch` will always return what's inside of `defaultValue` or `defaultValues` from `useForm`.
- The only difference between `useWatch` and `watch` is at the root (`useForm`) level or the custom hook level update.
- `useWatch`'s execution order matters, which means if you update a form value before the subscription is in place, then the value updated will be ignored.

```
setValue('test', 'data');
useWatch({ name: 'test' }); // ✗ subscription is happened after value update, no update received

useWatch({ name: 'example' }); // ✔ input value update will be received and trigger re-render
setValue('example', 'data');
```

- useWatch's result is optimised for render phase instead of useEffect's deps, to detect value updates you may want to use an external custom hook for value comparison.

## Examples

### 1) form

```
import React from "react";
import { useForm, useWatch } from "react-hook-form";

interface FormInputs {
  firstName: string;
  lastName: string;
}

function FirstNameWatched({ control }: { control: Control<FormInputs> }) {
  const firstName = useWatch({
    control,
    name: "firstName", // without supply name will watch the entire form, or ['firstName', 'lastName'] to watch both
    defaultValue: "default" // default value before the render
  });

  return <p>Watch: {firstName}</p>; // only re-render at the custom hook level, when firstName changes
}

function App() {
  const { register, control, handleSubmit } = useForm<FormInputs>();

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

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <label>First Name:</label>
      <input {...register("firstName")} />
      <input {...register("lastName")} />
      <input type="submit" />

      <FirstNameWatched control={control} />
    </form>
  );
}
```



## 2) Advance field array

```
import React from "react";
import { useWatch } from "react-hook-form";

function totalCal(results) {
  let totalValue = 0;

  for (const key in results) {
    for (const value in results[key]) {
      if (typeof results[key][value] === "string") {
        const output = parseInt(results[key][value], 10);
        totalValue = totalValue + (Number.isNaN(output) ? 0 : output);
      } else {
        totalValue = totalValue + totalCal(results[key][value], totalValue);
      }
    }
  }

  return totalValue;
}

export const Calc = ({ control, setValue }) => {
  const results = useWatch({ control, name: "test" });
  const output = totalCal(results);

  // isolated re-render to calc the result with Field Array
  console.log(results);

  setValue("total", output);

  return <p>{output}</p>;
};
```

# useFormState

Subscribe to form state update

**useFormState:** `(( { control: Control } ) => FormState`

This custom hook allows you to subscribe to each form state, and isolate the re-render at the custom hook level. It has its scope in terms of form state subscription, so it would not affect other useFormState and useForm. Using this hook can reduce the re-render impact on large and complex form application.

## Props

The following table contains information about the arguments for `useFormState`.

Name	Type	Description
control	object	control object provided by <code>useForm</code> . It's optional if you are using <code>FormContext</code> .
name	string   string[]	Provide a single input name, an array of them, or subscribe to all inputs' <code>formState</code> update.
disabled	boolean = false	Option to disable the subscription.
exact	boolean = false	This prop will enable an exact match for input name subscriptions.

## Return

Name	Type	Description
isDirty	boolean	<p>Set to <code>true</code> after the user modifies any of the inputs. Make sure to provide all inputs' <code>defaultValues</code> at the <code>useForm</code>, so hook form can have a single source of truth to compare whether the form is dirty.</p> <p><b>COPY</b></p> <pre>const {   formState: { isDirty, dirtyFields },   setValue, } = useForm({ defaultValues: { test: "" } });  // isDirty: true setValue('test', 'change')  // isDirty: false because there getValues() === defaultValues setValue('test', '')</pre> <p>File typed input will need to be managed at the app level due to the ability to cancel file selection and <code>FileList</code> object.</p>
dirtyFields	object	<p>An object with the user-modified fields. Make sure to provide all inputs' <code>defaultValues</code> via <code>useForm</code>, so the library can compare against the <code>defaultValues</code>.</p> <p>Dirty fields will <b>not</b> represent as <code>isDirty</code> <code>formState</code>, because dirty fields are marked field dirty at field level rather the entire form. If you want to determine the entire form state use <code>isDirty</code> instead.</p>
touchedFields	object	An object containing all the inputs the user has interacted with.
defaultValues	object	The value which has been set at <code>useForm</code> 's <code>defaultValues</code> or updated <code>defaultValues</code> via <code>reset</code> API.

<code>isSubmitted</code>	<code>boolean</code>	Set to <code>true</code> after the form is submitted. Will remain <code>true</code> until the <code>reset</code> method is invoked.
<code>isSubmitSuccessful</code>	<code>boolean</code>	Indicate the form was successfully submitted without any <code>Promise</code> rejection or <code>Error</code> been thrown within the <code>handleSubmit</code> callback.
<code>isSubmitting</code>	<code>boolean</code>	<code>true</code> if the form is currently being submitted. <code>false</code> otherwise.
<code>submitCount</code>	<code>number</code>	Number of times the form was submitted.
<code>isValid</code>	<code>boolean</code>	Set to <code>true</code> if the form doesn't have any errors. <code>isValid</code> is affected by <code>mode</code> at <code>useForm</code> . This state is only applicable with <code>onChange</code> , <code>onTouched</code> , and <code>onBlur</code> mode. <code>setError</code> has no effect on <code>isValid</code> formState, <code>isValid</code> will always derived via the entire form validation result.
<code>isValidating</code>	<code>boolean</code>	Set to <code>true</code> during validation.
<code>errors</code>	<code>object</code>	An object with field errors. There is also an <code>ErrorMessage</code> component to retrieve error message easily.

## Rules

Returned `formState` is wrapped with `Proxy` to improve render performance and skip extra computation if specific state is not subscribed, so make sure you deconstruct or read it before render in order to enable the subscription.

```
const { isDirty } = useFormState(); // ☒
const formState = useFormState(); // ☐ should deconstruct the formState
```

## Example

```
import * as React from "react";
import { useForm, useFormState } from "react-hook-form";

export default function App() {
  const { register, handleSubmit, control } = useForm({
    defaultValues: {
      firstName: "firstName"
    }
  });
  const { dirtyFields } = useFormState({
    control
  });
  const onSubmit = (data) => console.log(data);

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register("firstName")} placeholder="First Name" />
    </form>
  );
}
```

```
{dirtyFields.firstName && <p>Field is dirty.</p>}

<input type="submit" />
</form>
);
}
```

## ErrorMessage

An error message component to handle errors

**ErrorMessage:** `Component`

A simple component to render associated input's error message.

```
npm install @hookform/error-message
```

### Props

Name	Type	Required	Description
<b>name</b>	string	✓	Name of the field.
<b>errors</b>	object		errors object from React Hook Form. Optional if you are using FormProvider.
<b>message</b>	string   React.ReactElement		Inline error message.
<b>as</b>	React.ElementType   string		Wrapper component or HTML tag. For example: as="span" or as={<Text />}
<b>render</b>	(({ message: string   React.ReactElement, messages?: Object }) => any		This is a <u>render prop</u> for rendering error message or messages. Note: you need to set <code>criteriaMode</code> to 'all' for using messages.

### Examples

1) Single error message

```

import React from "react";
import { useForm } from "react-hook-form";
import { ErrorMessage } from '@hookform/error-message';

interface FormInputs {
  singleErrorInput: string
}

export default function App() {
  const { register, formState: { errors }, handleSubmit } = useForm<FormInputs>();
  const onSubmit = (data: FormInputs) => console.log(data);

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register("singleErrorInput", { required: "This is required." }} />
      <ErrorMessage errors={errors} name="singleErrorInput" />

      <ErrorMessage
        errors={errors}
        name="singleErrorInput"
        render={({ message }) => <p>{message}</p>
      />

      <input type="submit" />
    </form>
  );
}

```

## 2) Multiple error message

```

import React from "react";
import { useForm } from "react-hook-form";
import { ErrorMessage } from '@hookform/error-message';

interface FormInputs {
  multipleErrorInput: string
}

export default function App() {
  const { register, formState: { errors }, handleSubmit } = useForm<FormInputs>({
    criteriaMode: "all"
  });
  const onSubmit = (data: FormInputs) => console.log(data);

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input
        {...register("multipleErrorInput", {
          required: "This is required.",
          pattern: {
            value: /d+/,
            message: "This input is number only."
          },
        },
      />
    </form>
  );
}

```

```

      maxLength: {
        value: 10,
        message: "This input exceed maxLength."
      }
    }}}
  />
  <ErrorMessage
    errors={errors}
    name="multipleErrorInput"
    render={({ messages }) =>
      messages &&
      Object.entries(messages).map(([type, message]) => (
        <p key={type}>{message}</p>
      ))
    }
  />

  <input type="submit" />
</form>
);
}

```

# useFieldArray

React hooks for Field Array

**useFieldArray:** [UseFieldArrayProps](#)

Custom hook for working with Field Arrays (dynamic form). The motivation is to provide better user experience and performance. You can watch [this short video](#) to visualize the performance enhancement.

## Props

Name	Type	Required	Description
name	string	✓	Name of the field array. Note: Do not support dynamic name.
control	Object		control object provided by useForm. It's optional if you are using FormContext.
shouldUnregister	boolean		Whether Field Array will be unregistered after unmount.
keyName	string = id		Name of the attribute with autogenerated identifier to use as the key prop. This prop is no longer required and will be removed in the next major version.

Name	Type	Required	Description
rules	Object		<p>The same validation rules API as for <u>register</u>, which includes:  required, minLength, maxLength, validate  CODESandbox<sup>TS</sup>  useFieldArray({    rules: { minLength: 4 }  })</p> <p>In case of validation error, the root property is appended  to <code>formState.errors?.fieldArray?.root</code> of  type <code>FieldError</code>  Important: This is only applicable to <b>built-in</b> validation  only  <u>Resolvers</u> are yet to support useFieldArray root  level validation.</p>

## Examples

```
function FieldArray() {
  const { control, register } = useForm();
  const { fields, append, prepend, remove, swap, move, insert } = useFieldArray({
    control, // control props comes from useForm (optional: if you are using FormContext)
    name: "test", // unique name for your Field Array
  });

  return (
    {fields.map((field, index) => (
      <input
        key={field.id} // important to include key with field's id
        {...register(`test.${index}.value`)}
      />
    ))}
  );
}
```

## Return

Name	Type	Description
fields	<code>object &amp; { id: string }</code>	This object contains the <code>defaultValue</code> and key for your component.
append	<code>(obj: object   object[], focusOptions) =&gt; void</code>	Append input/inputs to the end of your fields and focus. The input value will be registered during this action. Important: append data is required and not partial.
prepend	<code>(obj: object   object[], focusOptions) =&gt; void</code>	Prepend input/inputs to the start of your fields and focus. The input value will be registered during this action. Important: prepend data is required and not partial.
insert	<code>(index: number, value: object   object[], focusOptions) =&gt; void</code>	Insert input/inputs at particular position and focus. Important: insert data is required and not partial.
swap	<code>(from: number, to: number) =&gt; void</code>	Swap input/inputs position.
move	<code>(from: number, to: number) =&gt; void</code>	Move input/inputs to another position.
update	<code>(index: number, obj: object) =&gt; void</code>	Update input/inputs at a particular position, updated fields will get unmount and remount. If this is not desired behavior, please use <code>setValue</code> API instead. Important: update data is required and not partial.
replace	<code>(obj: object[]) =&gt; void</code>	Replace the entire field array values.
remove	<code>(index?: number   number[]) =&gt; void</code>	Remove input/inputs at particular position, or remove all when no index provided.



## Rules

`useFieldArray` automatically generates a unique identifier named `id` which is used for `key` prop. For more information why this is required: <https://reactjs.org/docs/lists-and-keys.html#keys>

The `field.id` (and not `index`) must be added as the component key to prevent re-renders breaking the fields:

```
// ✔ correct:
{fields.map((field, index) => <input key={field.id} ... />)}

// ✘ incorrect:
{fields.map((field, index) => <input key={index} ... />)}
```

It's recommend to not stack actions one after another.

```
onClick={() => {
  append({ test: 'test' });
  remove(0);
}}

// ✔ Better solution: the remove action is happened after the second render
React.useEffect(() => {
  remove(0);
}, [remove])

onClick={() => {
  append({ test: 'test' });
}}
```

Each `useFieldArray` is unique and has its own state update, which means you should not have multiple `useFieldArray` with the same name.

Each input name needs to be unique, if you need to build checkbox or radio with the same name then use it with `useController` or `controller`.

Does not support flat field array.

When you append, prepend, insert and update the field array, the obj can't be empty object rather need to supply all your input's defaultValues.

```
append(); ✘
append({}); ✘
append({ firstName: 'bill', lastName: 'luo' }); ✔
```

## TypeScript

- when register input name, you will have to cast them as const

```
<input key={field.id} {...register(`test.${index}.test` as const)} />
```

- we do not support circular reference. Refer to this [Github issue](#) for more detail.
- for nested field array, you will have to cast the field array by its name.

```
const { fields } = useFieldArray({ name: `test.${index}.keyValue` as 'test.0.keyValue' });
```

## Examples

### 1 use Field Array

```
import * as React from "react";
import { useForm, useFieldArray, useWatch, Control } from "react-hook-form";

type FormValues = {
  cart: {
    name: string;
    price: number;
    quantity: number;
  }[];
};

const Total = ({ control }: { control: Control<FormValues> }) => {
  const formValues = useWatch({
    name: "cart",
    control
  });
  const total = formValues.reduce(
    (acc, current) => acc + (current.price || 0) * (current.quantity || 0),
    0
  );
  return <p>Total Amount: {total}</p>;
};

export default function App() {
  const {
    register,
    control,
    handleSubmit,
    formState: { errors }
  } = useForm<FormValues>({
    defaultValues: {
```

```

    cart: [{ name: "test", quantity: 1, price: 23 }]
  },
  mode: "onBlur"
});
const { fields, append, remove } = useFieldArray({
  name: "cart",
  control
});
const onSubmit = (data: FormValues) => console.log(data);

return (
  <div>
    <form onSubmit={handleSubmit(onSubmit)}>
      {fields.map((field, index) => {
        return (
          <div key={field.id}>
            <section className={"section"} key={field.id}>
              <input
                placeholder="name"
                {...register(`cart.${index}.name` as const, {
                  required: true
                })}
                className={errors?.cart?.[index]?.name ? "error" : ""}
              />
              <input
                placeholder="quantity"
                type="number"
                {...register(`cart.${index}.quantity` as const, {
                  valueAsNumber: true,
                  required: true
                })}
                className={errors?.cart?.[index]?.quantity ? "error" : ""}
              />
              <input
                placeholder="value"
                type="number"
                {...register(`cart.${index}.price` as const, {
                  valueAsNumber: true,
                  required: true
                })}
                className={errors?.cart?.[index]?.price ? "error" : ""}
              />
              <button type="button" onClick={() => remove(index)}>
                DELETE
              </button>
            </section>
          </div>
        );
      })}

      <Total control={control} />

      <button
        type="button"

```

```

      onClick={() =>
        append({
          name: "",
          quantity: 0,
          price: 0
        })
      }
    >
      APPEND
    </button>
    <input type="submit" />
  </form>
</div>
);
}

```

## 2 Nested Form

```

import * as React from "react";
import { useForm, useFieldArray, useWatch } from "react-hook-form";

export default function App() {
  const { control, handleSubmit } = useForm();
  const { fields, append, update } = useFieldArray({
    control,
    name: 'array'
  });

  return (
    <form onSubmit={handleSubmit((data) => console.log(data))}>
      {fields.map((field, index) => (
        <Edit
          key={field.id}
          control={control}
          update={update}
          index={index}
          value={field}
        />
      ))}

      <button
        type="button"
        onClick={() => {
          append({ firstName: "" });
        }}
      >
        append
      </button>
      <input type="submit" />
    </form>
  );
}

const Display = ({ control, index }) => {

```

```

const data = useWatch({
  control,
  name: `array.${index}`
});
return <p>{data?.firstName}</p>;
};

const Edit = ({ update, index, value, control }) => {
  const { register, handleSubmit } = useForm({
    defaultValues: value
  });

  return (
    <div>
      <Display control={control} index={index} />

      <input
        placeholder="first name"
        {...register(`firstName`, { required: true })}
      />

      <button
        type="button"
        onClick={handleSubmit((data) => update(index, data))}
      >
        Submit
      </button>
    </div>
  );
};

```

### 3 Conditional field Array

```

import React from 'react';
import { useForm, useWatch, useFieldArray, Control } from 'react-hook-form';

type FormValues = {
  data: { name: string }[];
};

const ConditionField = ({
  control,
  index,
  register,
}: {
  control: Control<FormValues>;
  index: number;
}) => {

```

```

const output = useWatch({
  name: 'data',
  control,
  defaultValue: 'yay! I am watching you :)',
});

return (
  <>
    {output[index]?.name === "bill" && (
      <input {...register(`data[${index}].conditional`)} />
    )}
    <input
      {...register(`data[${index}].easyConditional`)}
      style={{ display: output[index]?.name === "bill" ? "block" : "none" }}
    />
  </>
);
};

const UseFieldArrayUnregister: React.FC = () => {
  const { control, handleSubmit, register } = useForm<FormValues>({
    defaultValues: {
      data: [{ name: 'test' }, { name: 'test1' }, { name: 'test2' }],
    },
    mode: 'onSubmit',
    shouldUnregister: false,
  });
  const { fields } = useFieldArray({
    control,
    name: 'data',
  });
  const onSubmit = (data: FormValues) => console.log(data);

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      {fields.map((data, index) => (
        <>
          <input {...register(`data[${index}].name`)} />
          <ConditionField control={control} register={register} index={index} />
        </>
      ))}
      <input type="submit" />
    </form>
  );
};

```

#### 4 Focus name/index

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

const App = () => {
  const { register, control } = useForm<{
    test: { value: string }[]>
  >({
    defaultValues: {
      test: [{ value: '1' }, { value: '2' }],
    },
  });
  const { fields, prepend, append } = useFieldArray({
    name: 'test',
    control,
  });

  return (
    <form>
      {fields.map((field, i) => (
        <input key={field.id} {...register(`test.${i}.value` as const)} />
      ))}
      <button
        type="button"
        onClick={() => prepend({ value: '' }, { focusIndex: 1 })}
      >
        prepend
      </button>
      <button
        type="button"
        onClick={() => append({ value: '' }, { focusName: 'test.0.value' })}
      >
        append
      </button>
    </form>
  );
};
```

## Tips

### *Custom Register*

You can also register inputs at Controller without the actual input. This makes `useFieldArray` quick and flexible to use with complex data structure or the actual data is not stored inside an input.

```
import { useForm, useFieldArray, Controller, useWatch } from "react-hook-form";
```

```
const ConditionalInput = ({ control, index, field }) => {  
  const value = useWatch({  
    name: "test",  
    control  
  });  
  
  return (  
    <Controller  
      control={control}  
      name={`test.${index}.firstName`}   
      render={({ field }) =>  
        value?.[index]?.checkbox === "on" ? <input {...field} /> : null  
      }  
    />  
  );  
};
```

```
function App() {  
  const { control, register } = useForm();  
  const { fields, append, prepend } = useFieldArray({  
    control,  
    name: "test"  
  });  
  
  return (  
    <form>  
      {fields.map((field, index) => (  
        <ConditionalInput key={field.id} {...{ control, index, field }} />  
      ))}  
    </form>  
  );  
}
```

### *Controlled Field Array*

There will be cases where you want to control the entire field array, which means each `onChange` reflects on the `fields` object.



```
import { useForm, useFieldArray } from "react-hook-form";

export default function App() {
  const { register, handleSubmit, control, watch } = useForm<FormValues>();
  const { fields, append } = useFieldArray({
    control,
    name: "fieldArray"
  });
  const watchFieldArray = watch("fieldArray");
  const controlledFields = fields.map((field, index) => {
    return {
      ...field,
      ...watchFieldArray[index]
    };
  });

  return (
    <form>
      {controlledFields.map((field, index) => {
        return <input {...register(`fieldArray.${index}.name` as const)} />;
      })}
    </form>
  );
}
```