# Forms Without the Fuss

*React Hook Form in Action*

METHOD

# Forms Are Easy Until They Aren't

- `useState` for every field
- Custom validation logic scattered around
- One large `handleSubmit` block

```
const [email, setEmail] = useState('');
const [error, setError] = useState('');

const handleSubmit = (e) => {
  e.preventDefault();
  if (!email.includes('@')) setError('Invalid email');
};
```

# It Gets Messy Fast

- Validation lives far from the input
- You manually track errors and touched state
- Code grows quickly with little reusability

```
const [touched, setTouched] = useState(false);
const handleBlur = () => setTouched(true);
```

# There’s a Better Way... React Hook Form!

- One hook to manage form state
- Input registration is declarative
- Built-in validation and tracking

# `useForm()` — Your Form's Core

- Sets up internal form state
- Returns helpers like `register`, `handleSubmit`, and `formState`
- Accepts `defaultValues`, validation mode, and more

```
const { register, handleSubmit } = useForm({ defaultValues: { email: '' } });
```

# `register()` — Connecting Native Inputs

- Binds input to the form
- Handles `value`, `onChange`, and `ref`
- Supports inline validation rules

```
<input {...register('email', { required: true })} />
```

# Validation Is Built In

- Rules like `required`, `pattern`, `minLength`, etc.
- Works inline or with external schema
- Errors managed inside `formState`

```
<input
  {...register('email', {
    pattern: { value: /.+@.+/, message: 'Invalid email' },
  })}
/>
```

# `formState` — Centralized Status

- Tracks `errors`, `isDirty`, `isValid`, `touchedFields`, etc.
- Drives validation display and form state logic
- Keeps UI reactive without extra state

```
const { errors, isDirty } = formState;
```

# Touched and Dirty Fields

- `touchedFields` tracks interaction (focus + blur)
- `dirtyFields` tracks changes from default values
- Useful for conditionally showing feedback or enabling buttons

```
if (touchedFields.email && dirtyFields.email) {
  /* ... */
}
```

METHOD

# Not All Inputs Work with `register`

- Custom components like `Select`, `Radio.Group`, `DatePicker`
- Don’t emit native input events
- Need special handling for RHF compatibility

```
<Select {...register('framework')} /> // ❌ not supported
```

# Use `Controller` for Custom Components

- Wraps non-native inputs
- Manages `value`, `onChange`, and validation
- Required for most UI libraries

```
<Controller name="framework" control={control} render={({ field }) => <Select {...field} />} />
```

# Share Form State with `FormProvider`

- Enables access to form methods across nested components
- No need to pass props manually
- Use `useFormContext()` to connect

```
<FormProvider {...methods}>
  <MyFormChild />
</FormProvider>
```

# `watch()` — Observe Field Values

- Monitor live form input
- Useful for previews, conditional rendering, and debug tools
- Can watch individual fields or the full form

```
const email = watch('email');
```

# RHF in Production

- Strong TypeScript support
- Easily composable for larger forms
- Works with any component library

```
type FormValues = { email: string; message: string };
```

# Alright, Let’s See the Form

- Live demo of the full feedback form
- Built with Mantine + React Hook Form
- Everything shown so far, applied in a real-world example

# Resources & Links

- 📚 Docs: react-hook-form.com
- 🔗 GitHub: github.com/react-hook-form/react-hook-form
  ⭐ 42.8k stars, 🍴 2.2k forks, 🧱 ~12kB min+gzip
- 💻 Demo repo: https://github.com/neilgamb/react-hook-form-preso