Ministry of SMEs and Startups

KOSME Korea SMEs and Startups Agency

LIKELION

JOBKOREA

LESSON 09 Form Handling (React Form Hook)

WEEK 02

Introduction to Form Handling

- **What is Form Handling?**
  - Managing user input in forms, including validation and submission.
- **Why Important?**
  - Essential for user interaction in web applications.
- **Official Reference**
  - React Hook Form Documentation (https://react-hook-form.com)

Challenges with Native Form Handling

- **Manual State Management**
  - Using useState for each input is tedious and error-prone.
- **Performance Issues**
  - Frequent re-renders on every input change.

```
export default function NativeFormHandling() {
  const [name, setName] = React.useState('');
  return (
    <div>
      <input value={name} onChange={(e) => setName(e.target.value)} />
    </div>
  );
}
```

What is React Hook Form?

- **Definition**
  - A lightweight library for managing forms in React with hooks.
- **Key Features**
  - Minimal re-renders, easy validation, uncontrolled components.
- **Benefits**
  - Simplifies form handling, improves performance.

Installing React Hook Form

- ❖ **Installation Command**
  - ➢ npm install react-hook-form
- ❖ **Basic Setup**
  - ➢ Import useForm hook to start building forms.
- ❖ **Official Guide**
  - ➢ React Hook Form Get Started (https://react-hook-form.com/get-started)

Basic Usage of useForm

❖ **useForm Hook**

➢ Provides form state and methods like **register**, **handleSubmit**.

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

export default function BasicUsage() {
  const { register, handleSubmit } = useForm();
  const onSubmit = (data) => {
    console.log(data);
  };
  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register('name')} />
      <button type="submit">Submit</button>
    </form>
  );
}
```

Uncontrolled vs. Controlled Components

- **Controlled Components**
  - Inputs tied to state, managed by React.
- **Uncontrolled Components**
  - Inputs managed by DOM, used by React Hook Form for efficiency.
- **Why Uncontrolled?**
  - Reduces re-renders, better for large forms.

Handling Form Submission

❖ **handleSubmit**

Wraps submission logic, passes form data to callback.

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

export default function BasicUsage() {
  const { register, handleSubmit } = useForm();
  const onSubmit = (data) => {
    console.log(data);
  };
  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register('name')} />
      <button type="submit">Submit</button>
    </form>
  );
}
```

Form Validation Basics

❖ **Built-in Validation**

Use register options like **required**, **minLength**.

```
export default function BasicUsage() {
  const { register, handleSubmit } = useForm();
  const onSubmit = (data) => {
    console.log(data);
  };
  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register('name', { required: true, minLength: 3 })} />
      <button type="submit">Submit</button>
    </form>
  );
}
```

Displaying Validation Errors

❖ **formState.errors**

Tracks validation errors for each field.

```tsx
interface IFormInput {
  name: string;
}

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

  const onSubmit = (data) => {
    console.log(data);
  };
  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register('name', { required: true, minLength: 3 })} />
      {errors.name && <p>{errors.name.message}</p>}
      <button type="submit">Submit</button>
    </form>
  );
}
```

Custom Validation

❖ **Custom Rules**

Use validate function in register for custom logic

```
interface IFormInput {
  email: string;
}

export default function CustomValidation() {
  const { register, handleSubmit } = useForm<IFormInput>();
  const onSubmit = (data) => console.log(data);
  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input
        {...register('email', {
          validate: (value) => value.includes('@') || 'Invalid email',
        })}
      />
      <button type="submit">Submit</button>
    </form>
  );
}
```

Ministry of SMEs and Startups KOSME Korea SMEs and Startups Agency LIKELION JOBKOREA

Default Values

❖ **Setting Defaults**

Use defaultValues in useForm to pre-fill form.

```tsx
interface IFormInput {
  name: string;
  age: number;
}

export default function DefaultValues() {
  const { register, handleSubmit } = useForm<IFormInput>({
    defaultValues: { name: 'John Doe', age: 25 },
  });
  const onSubmit: SubmitHandler<IFormInput> = (data) => console.log(data);
  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register('name')} />
      <input type="number" {...register('age')} />
      <button type="submit">Submit</button>
    </form>
  );
}
```

Resetting Form

❖ **reset Method**

Clears or resets form to default values.

```tsx
interface IFormInput {
  name: string;
  age: number;
}

export default function ResettingForm() {
  const { register, handleSubmit, reset } = useForm<IFormInput>();
  const onSubmit: SubmitHandler<IFormInput> = (data) => {
    console.log(data);
    reset();
  };
  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register('name')} />
      <input type="number" {...register('age')} />
      <button type="submit">Submit</button>
    </form>
  );
}
```

Watching Form Values

❖ **watch Method**

Tracks input changes in real-time.

```
interface IFormInput {
  name: string;
}

export default function WatchingFormValues() {
  const { register, watch } = useForm<IFormInput>();
  const name = watch('name');
  return (
    <div>
      <input {...register('name')} />
      <p>Current Value: {name}</p>
    </div>
  );
}
```

Handling Complex Forms

❖ **Nested Fields**

Use dot notation for nested objects (e.g., **user.name**).

```
interface IFormInput {
  user: IUser;
}

interface IUser {
  name: string;
  age: number;
}

export default function NestedFields() {
  const { register, handleSubmit } = useForm<IFormInput>();
  const onSubmit: SubmitHandler<IFormInput> = (data) => console.log(data);
  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register('user.name')} />
      <input {...register('user.age')} />
      <button type="submit">Submit</button>
    </form>
  );
}
```

Array Fields

❖ **useFieldArray**

Manages dynamic arrays of fields (e.g., list of items).

```tsx
interface Item {
  value: string;
}

interface FormValues {
  items: Item[];
}

export default function ArrayFields() {
  const { register, handleSubmit, control } = useForm<FormValues>();
  const { fields, append } = useFieldArray({ control, name: 'items' });

  const onSubmit = (data: FormValues) => console.log(data);
  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      {fields.map((field, index) => (
        <input key={field.id} {...register(`items.${index}.value`)} />
      ))}
      <button type="button" onClick={() => append({ value: '' })}>
        Add Item
      </button>
      <button type="submit">Submit</button>
    </form>
  );
}

```

Form Context with Controller

❖ **Controller**

Wraps third-party components for form integration.

```
import { Controller, useForm } from 'react-hook-form';
import { Select } from 'antd';

export default function FormContextWithController() {
  const { control, handleSubmit } = useForm();
  const onSubmit = (data) => console.log(data);
  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <Controller
        name="color"
        control={control}
        render={({ field }) => {
          return <Select options={[{ value: 'red', label: 'Red' }]} {...field} />;
        }}
      />
      <button type="submit">Submit</button>
    </form>
  );
}
```

Ministry of SMEs and Startups KOSME Korea SMEs and Startups Agency LIKELION JOBKOREA

Schema Validation with Yup

❖ **What is Yup?**

A schema validation library, integrates with React Hook Form.

```tsx
import { useForm, type SubmitHandler } from 'react-hook-form';
import { yupResolver } from '@hookform/resolvers/yup';
import * as yup from 'yup';

const schema = yup.object({
  email: yup.string().email('Invalid email').required('Email is required'),
});

export default function SchemaValidation() {
  const {
    register,
    handleSubmit,
    formState: { errors },
  } = useForm<IFormInput>({
    resolver: yupResolver(schema),
  });

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

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register('email')} />
      {errors.email && <p>{errors.email.message}</p>}
      <button type="submit">Submit</button>
    </form>
  );
}
```

Form Submission States

❖ **formState**

Tracks states like **isSubmitting**, **isSubmitted**.

```
export default function FormSubmissionStates() {
  const {
    register,
    handleSubmit,
    formState: { isSubmitting },
  } = useForm<IFormInput>();

  const onSubmit = (data) => {
    console.log(data);
  };
  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register('name', { required: true, minLength: 3 })} />
      <button type="submit"> {isSubmitting ? 'Submitting...' : 'Submit'}</button>
    </form>
  );
}
```

Performance Optimization

- **Minimize Re-renders**
  - Uncontrolled inputs reduce unnecessary updates.
- **Use mode Option**
  - mode: 'onChange' for real-time validation.
- **Reference**
  - React Hook Form Performance

Handling File Inputs

❖ **File Input**

Use register with type="file".

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

export default function HandlingFileInputs() {
  const { register, handleSubmit } = useForm();
  const onSubmit = (data) ⇒ console.log(data.file[0].name);
  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input type="file" {...register('file')} />
      <button type="submit">Submit</button>
    </form>
  );
}
```

Integrating with APIs

❖ **API Submission**

Send form data to a backend API.

```
interface IFormInput {
  name: string;
}

export default function IntegratingWithAPIs() {
  const { register, handleSubmit } = useForm<IFormInput>();
  const onSubmit: SubmitHandler<IFormInput> = async (data) => {
    const res = await fetch('/api/submit', {
      method: 'POST',
      body: JSON.stringify(data),
    });
    console.log(await res.json());
  };
  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register('name')} />
      <button type="submit">Submit</button>
    </form>
  );
}

```

Custom Input Components

❖ **Reusable Inputs**

Create custom inputs compatible with React Hook Form

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

function CustomInput({ register, name, ...props }) {
  return <input {...register(name)} {...props} />;
}

export default function CustomInputComponents() {
  const { register, handleSubmit } = useForm();
  const onSubmit = (data) => console.log(data);
  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <CustomInput register={register} name="name" placeholder="Name" />
      <button type="submit">Submit</button>
    </form>
  );
}

```

Form with Conditional Fields

❖ **Dynamic Fields**

Show/hide fields based on user input.

```tsx
interface IFormInput {
  hasPet?: boolean;
  petName?: string;
}

export default function FormWithConditionalFields() {
  const { register, handleSubmit, watch } = useForm<IFormInput>();
  const hasPet = watch('hasPet');
  const onSubmit: SubmitHandler<IFormInput> = (data) => console.log(data);
  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input type="checkbox" {...register('hasPet')} />
      {hasPet && <input {...register('petName')} />}
      <button type="submit">Submit</button>
    </form>
  );
}
```

Error Handling Strategies

❖ **Centralized Errors**

Use formState.errors for global error display.

```
export default function CentralizedErrors() {
  const {
    register,
    handleSubmit,
    formState: { errors },
  } = useForm();
  const onSubmit = (data) ⇒ console.log(data);
  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register('name', { required: 'Required' })} />
      {Object.keys(errors).length > 0 && <p>Form has errors, please check inputs</p>}
      <button type="submit">Submit</button>
    </form>
  );
}
```

Advanced: Form Persistance

❖ **Persisting Form Data**

Save form state to localStorage for recovery.

```tsx
export default function PersistingFormData() {
  const storedFormData = localStorage.getItem('formData');
  const { register, handleSubmit } = useForm<IFormInput>({
    defaultValues: storedFormData ? JSON.parse(storedFormData) : {},
  });
  const onSubmit: SubmitHandler<IFormInput> = (data) => {
    localStorage.setItem('formData', JSON.stringify(data));
    console.log(data);
  };
  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register('name')} />
      <button type="submit">Submit</button>
    </form>
  );
}
```

Common Pitfalls

- **Overusing watch**
  - Causes performance issues with large forms.
- **Missing Validation**
  - Always define rules to prevent invalid submissions.
- **Ignoring TypeScript**
  - Use types for better maintainability.