Enhancing React Forms with the **react-hook-form** Hook and **yup** Schema (Optional)

We are going to enhance our `ContactForm.js` with `react-hook-form` and `yup`. The benefits we get are:

- Less code for managing form state and validation.

- All validation rules are defined in one place (the Yup schema), improving code organization and maintainability.

- Yup's schema-based approach is *declarative,* allowing you to define validation rules in a clear and concise way.

- Yup handles complex validation scenarios with built-in methods.

- The yupResolver integrates Yup with React Hook Form easily.
  Step 1: In addition to the libraries used in the Forms lab, install dependencies react-hook-form, @hookform/resolvers, and yup:

```
install react-hook-form @hookform/
resolvers yup
```

- **Step 2: Create validationSchema.js to centralize validation rules:**
    1. Define your validation schema using Yup in this file.
    2. Specify the *shape* of your form data with the `shape()` method, and
       the validation rules for each field.
    3. Set custom error messages.

```
// validationSchema.js
import * as yup from 'yup';

const schema = yup.object().shape({
  name: yup

    .string()
    .required('Name is required')
    .min(2, 'Name must be at least 2
characters')
    .max(50, 'Name must be at most 50
characters'),

  email: yup
    .string()
    .email('Invalid email format')
    .required('Email is required'),

  message: yup
    .string()
    .required('Message is required')
    .min(10, 'Message must be at least 10
characters')
    .max(500, 'Message must be at most
500

characters'),
});

export default schema;
```

Step 3: Update ContactForm.js:

1. Import `useForm, yupResolver`, and your schema file. The schema could have been defined in the ContactForm file, but it is more reusable if

defined in another file.

2. Use `useForm` with the resolver option to integrate Yup.

3. Register your input fields using register without specifying validation
rules directly. Remove validation rules from individual `register` calls in your form component. The validation logic is now centralized in the Yup schema, making your form component cleaner and more focused on rendering the UI.

4. Handle form submission using `handleSubmit`.

5. Display error messages from the `errors` object.

```
import React from 'react';
import { useForm } from 'react-hook-
form';
import { yupResolver } from '@hookform/
resolvers/yup';
import schema from './validationSchema';
import TextInput from './TextInput';
import SubmitButton from './
SubmitButton';
import { Box } from '@mui/material';
const ContactForm = () => {
  const {
    register,
    handleSubmit,
    formState: { errors },
  } = useForm({
    resolver: yupResolver(schema),
});

  const onSubmit = (data) => {
    // Simulate an API call
    setTimeout(() => {
      alert(JSON.stringify(data, null,
2));
    }, 500);
};


  return (
    <form
onSubmit={handleSubmit(onSubmit)}>
      <Box mb={2}>
        <TextInput
          {...register('name')}
          label="Name"
          error={!!errors.name}
helperText={errors?.name?.message}
/> </Box>

      <Box mb={2}>
        <TextInput
          {...register('email')}
          label="Email"
          error={!!errors.email}

helperText={errors?.email?.message}
/> </Box>

      <Box mb={2}>
        <TextInput
          {...register('message')}
          label="Message"
          multiline
          rows={4}
          error={!!errors.message}

helperText={errors?.message?.message}
/> </Box>

      <SubmitButton>Submit</SubmitButton>
    </form>
); };

export default ContactForm;
```

Enhancing React Forms with Formik and yup Schema (Optional) Step 1: Understanding When to use Formik

Formik and React Hook Form are popular libraries for managing forms in React. Here are the benefits of using Formik over React Hook Form:

- Formik is easier to learn and use, especially for developers new to form management libraries. The API is intuitive, for example `initialValues`, `validationSchema`, and `handleSubmit`.

- Like `react-form-hook`, Formik integrates with Yup for schema-based validation, making it easy to define and manage complex validation rules. Code is cleaner and more maintainable.

- Formik handles form state internally, abstracting away the complexities of managing input values, errors, and touched fields, reducing boilerplate code.

- Formik is more *opinionated*, but this makes it easier to learn.

- Formik has a large and active community, with extensive documentation,
tutorials, and third-party integrations available.
React Hook Form Advantages:

- React Hook Form is known for exceptional performance for large and complex forms. It minimizes re-renders and optimizes state updates, leading to a smoother user experience.

- React Hook Form is less opinionated and can integrate more easily with other libraries.

- React Hook Form has a smaller bundle size compared to Formik.
Step 2: In addition to the libraries used in the Forms lab, install Formik and Yup in Your Application

```
npm install formik yup
```

Step 3: Let's convert the original ContactForm to use Formik

1. Note that in this exercise, the schema and ContactForm are in the same file.

2. Import Formik, Form, and Field from Formik

3. Define a validationSchema using Yup to specify the validation rules for each field.

4. Create the `Formik` component:

- `initialValues`: Sets the initial values for the form fields.
- `validationSchema`: Attaches the validation schema.
- `onSubmit`: Handles the form submission.

5. `Form` and `Field`:

  - `Form`: Renders the actual form element. The **Formik `Form` component** plays a crucial role in managing and rendering forms within a React application using the Formik library. The `Form` component creates the underlying structure for your form and establishes a context that provides access to Formik's state and helper functions to its child components (like `Field, ErrorMessage`, etc.). When the user submits the form, Formik’s `onSubmit` function is triggered. This function receives the current form values and can be used to perform actions like sending data to an API, validating the form, or updating application state. The `Form` component works with Formik's validation to handle and display error messages. If validation errors occur, the `Form` component ensures that error messages are appropriately displayed near the relevant fields.

- `Field`: Connects each input field to Formik's state management and validation. Formik's **`Field`** component handles the integration with
your custom input components, passing down the necessary props
like **`value`**, **`onChange`**, **`onBlur`**, **`error`**, and **`helperText`**.

- `component`: Specifies the component to render for the field (in this
case, our custom `TextInput`).