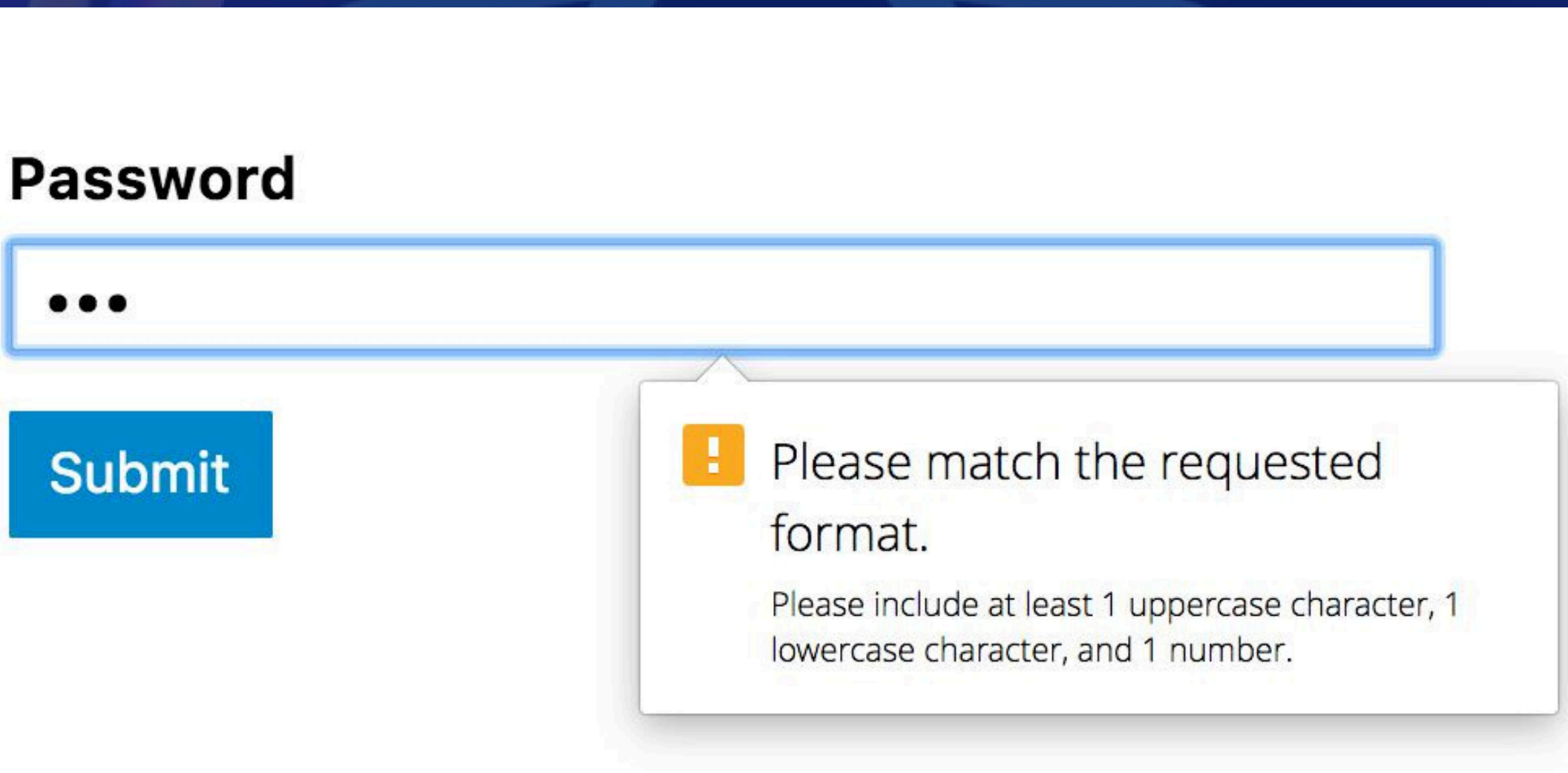
Day 13

Form Validation In React



A Comprehensive Guide

@oluwakemi Oluwadahunsi

Introduction

Handling form validation in React.js is an important aspect of developing reliable, user-friendly applications.

It ensures that users **input the correct data** in the desired format, **prevents incorrect submissions**, and improves the overall user experience.

React.js provides flexibility for handling form validation, offering built-in techniques and third-party libraries to manage simple to complex validation requirements.

In this detailed guide, we'll go through:

- **Why form validation is important in React**
- **Different types of form validation**
- **Various methods to handle form validation in React**
- **Handling Asynchronous validation**
- **Best Practices for form validation in React**

@oluwakemi Oluwadahunsi

Why Form Validation is Important in React

Form validation prevents users from submitting incorrect, incomplete, or insecure data. This is essential for:

- **Data Integrity:** Ensuring that submitted data is consistent, structured, and in the expected format.
- **Security:** Preventing malicious input like SQL injection or script attacks.
- **User Experience:** Guiding users to correct errors in real-time, providing a smooth and interactive form experience.
- **Avoiding Errors:** Ensuring the application doesn’t break due to incorrect input.

@oluwakemi Oluwadahunsi

Types of Form Validation

Form validation in React can be broadly categorized into three types:

1. **Client-Side Validation:** This involves validating the input on the client side before sending it to the server. It ensures immediate feedback and prevents unnecessary server requests if the data is invalid.

2. **Server-Side Validation:** This happens on the backend, ensuring that even if client-side validation is bypassed (e.g., by disabling JavaScript), the data is validated correctly before being processed.

3. **Real-Time Validation:** Inputs are validated immediately as the user interacts with the form, offering real-time feedback.

@oluwakemi Oluwadahunsi

Methods for Handling Form Validation in React

There are several ways to handle form validation in React:

- Manual Validation (using React's state and logic)
- Regular Expressions (RegEx)
- Third-Party Libraries (like Formik, React Hook Form, and Yup for schema validation)
- Validation Using HTML5 Constraints

@oluwakemi Oluwadahunsi

1. Manual Validation in React

Manual validation involves handling the form state and validating the inputs through custom logic without external libraries.

React's built-in **useState** and **useEffect** hooks are powerful tools for managing form validation manually. This approach is often used for small-scale forms where the requirements are simple.

Example: Simple Email and Password Validation

```jsx
import React, { useState } from 'react';

function SignupForm() {
  // State to store form data
  const [formData, setFormData] = useState({
    email: '',
    password: '',
  });

  const [errors, setErrors] = useState({});

  const validateForm = () => {
    let newErrors = {};

    if (!formData.email) {
      newErrors.email = 'Email is required';
    }

    if (!formData.password) {
      newErrors.password = 'Password is required';
    } else if (formData.password.length < 8) {
      newErrors.password = 'Password must be at least 8 characters long';
    }
    setErrors(newErrors);

    return Object.keys(newErrors).length === 0;
  };

  // Handle form submission
  const handleSubmit = (e) => {
    e.preventDefault();

    if (validateForm()) {
      console.log('Form submitted successfully');
      // Submit form logic here
    }
  };

  // Handle input changes
  const handleChange = (e) => {
    setFormData({
      ...formData,
      [e.target.name]: e.target.value,
    });
  };
```

state to store form data

state to store errors

function to validates inputs

email validation

password validation

updates error state

function that performs submission logic

handles input changes

continued on next page

```jsx
return (
    <form onSubmit={handleSubmit}>
      <div>
        <label>Email</label>
        <input
          type="text"
          name="email"
          value={formData.email}
          onChange={handleChange}
        />
        {errors.email && <span style={{ color: 'red' }}>{errors.email}</span>}
      </div>

      <div>
        <label>Password</label>
        <input
          type="password"
          name="password"
          value={formData.password}
          onChange={handleChange}
        />
        {errors.password && <span style={{ color: 'red' }}>{errors.password}</span>}
      </div>
      <button type="submit">Submit</button>
    </form>
  );
}
export default SignupForm;
```

email input

password input

displays validation errors

In this SignupForm component above:

- **formData** is the state object that holds the current values (initial state) of the form fields (email and password), in this case, it's empty.

@oluwakemi Oluwadahunsi

- **errors** is used to store any validation errors that occur when the form is submitted.

- For the **email validation**, it simply checks if the **email field is empty**. If it is, an error message ('Email is required') is set.

- The password validation ensures that the **password field is not empty** ('Password is required') and that it is **at least 8 characters long** ('Password must be at least 8 characters long').

- When the form is submitted, the **handleSubmit** function calls **validateForm**. If **validateForm** returns **true** (meaning there are no errors), the form submission logic can proceed. Otherwise, the form will show the validation errors.

- The form shows any **validation errors** dynamically by checking the errors state and displaying the corresponding error messages below each input field.

@oluwakemi Oluwadahunsi

Pros for using RegEx:

- **Full control:** You have complete control over how and when to validate the form.
- **Customization:** Easily customizable, allowing for any kind of validation logic specific to your use case.
- **No external dependencies:** You don’t need to install or depend on any third-party libraries.
- **Lightweight:** No extra package bloat.

Cons:

- **Boilerplate:** It can require a lot of repetitive code, especially for complex forms.
- **Prone to errors:** Writing validation logic manually can lead to bugs or inconsistent validation behavior.
- **Difficult to manage complex forms:** For large forms with many fields and rules, manual validation can become difficult to manage.

@oluwakemi Oluwadahunsi

2. Validation Using RegEx

Regular Expressions (RegEx) provide a powerful and flexible way to handle form validation by defining patterns that input data must match.

It is widely used to validate formats such as email addresses, phone numbers, passwords, zip codes, and more.

In React, integrating RegEx for form validation can be done either **manually** or **alongside other validation methods**.

Basic RegEx Validation (Manual) Example:

```jsx
import React, { useState } from "react";

function RegExFormValidation() {
  const [email, setEmail] = useState("");
  const [password, setPassword] = useState("");
  const [errors, setErrors] = useState({});

  // RegEx Patterns for Validation
  const emailPattern = /^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/;
  const passwordPattern = /^(?=.*[A-Za-z])(?=.*\d)[A-Za-z\d]{6,}$/;

  // Function to Validate Email and Password
  const validate = () => {
    let validationErrors = {};

    // Email Validation
    if (!emailPattern.test(email)) {
      validationErrors.email = "Invalid email format!";
    }

    // Password Validation (At least 6 characters, one letter, one number)
    if (!passwordPattern.test(password)) {
      validationErrors.password = "Password must be 6+ characters, including at least one letter and one number.";
    }

    return validationErrors;
  };

  // Handle Form Submission
  const handleSubmit = (e) => {
    e.preventDefault();
    const errors = validate();

    if (Object.keys(errors).length > 0) {
      setErrors(errors);
    } else {
      // Proceed with form submission
      alert("Form Submitted Successfully!");
    }
  };
```

states to store form data and errors

Regular Expression patterns

validation function

email validation

password validation

function that performs submission logic

updates error state

continued on next page

```jsx
return (
    <form onSubmit={handleSubmit}>
      <div>
        <label>Email:</label>
        <input
          type="text"
          value={email}
          onChange={(e) => setEmail(e.target.value)}
        />
        {errors.email && <p>{errors.email}</p>}
      </div>

      <div>
        <label>Password:</label>
        <input
          type="password"
          value={password}
          onChange={(e) => setPassword(e.target.value)}
        />
        {errors.password && <p>{errors.password}</p>}
      </div>

      <button type="submit">Submit</button>
    </form>
  );
}

export default RegExFormValidation;
```

email input

password input

displays validation errors

In the Example above:

- The **validate** function checks both the **email** and **password** using predefined RegEx patterns.

@oluwakemi Oluwadahunsi

- For the **email**, it ensures that the input matches a valid email format (characters followed by '@' and a **valid domain** like .com, .net, etc).

- For the **password**, the **RegEx** checks that it has **at least 6 characters**, contains **at least one letter**, and **one number**. If any of these conditions are not met, an **error message** is returned for each invalid field.

- If validation fails, the corresponding **error message** is displayed under the input field using the **setErrors** state, allowing users to see what they need to correct.

- When the user submits the form, it first runs the validation. If there are errors, the form won't be submitted, and the errors will be shown. If no errors are found, it performs the form logic.

@oluwakemi Oluwadahunsi

Pros for using RegEx:

- **Efficient for simple patterns:** Regex is useful for checking simple input patterns like email addresses, phone numbers, or passwords.

- **Highly customizable:** You can create patterns to validate nearly any format or type of input.

Cons:

- **Hard to read and maintain:** Regex can be difficult to understand, especially for beginners or when creating complex patterns.

- **Not a complete solution:** Regex is great for format validation but not for other types of validation, like ensuring a password meets multiple security criteria (e.g., min length, special characters, etc.).

- **Debugging is tricky:** Troubleshooting incorrect regex patterns can be time-consuming.

@oluwakemi Oluwadahunsi

3. Validation Using Libraries

Third-party libraries like **Formik** and **React Hook Form** simplify handling form validation, especially for **complex and large-scale** forms. These libraries provide built-in methods to manage form state, validation, and submission.

Using Formik and Yup

Formik and **Yup** are popular libraries used to simplify form handling and validation in React.

Formik is a powerful library that provides a **higher-level API** for handling forms, validation, and submission. You can use it in combination with Yup for **schema-based** validation.

Example with Formik and Yup:

```jsx
import React from 'react';
import { Formik, Form, Field, ErrorMessage } from 'formik';
import * as Yup from 'yup';

const validationSchema = Yup.object({
  email: Yup.string()
    .email('Invalid email format')
    .required('Email is required'),
  password: Yup.string()
    .min(6, 'Password must be at least 6 characters long')
    .required('Password is required'),
});

function FormikForm() {
  return (
    <Formik
      initialValues={{ email: '', password: '' }}
      validationSchema={validationSchema}
      onSubmit={(values) => {
        console.log('Form data', values);
      }}
    >
      <Form>
        <div>
          <label>Email</label>
          <Field type="email" name="email" />
          <ErrorMessage name="email" component="span" />
        </div>

        <div>
          <label>Password</label>
          <Field type="password" name="password" />
          <ErrorMessage name="password" component="span" />
        </div>

        <button type="submit">Submit</button>
      </Form>
    </Formik>
  );
}

export default FormikForm;
```

imports necessary components

email validation

password validation

assigning validation schema provided by yup

email input field

password field

updates error state

@oluwakemi Oluwadahunsi

- **Formik** manages the form's state and submission. It uses **initialValues** to define the starting values for the form fields (**email** and **password**).
- Yup provides the validation schema (validationSchema) to enforce rules:
  - **Email:** Must be a valid email format (**Yup.string().email()**). If the email format is invalid, it shows the error "**Invalid email format**." The field is also marked as required (**Yup.string().required()**), so it will show "Email is required" if left empty.
  - **Password:** Must be **at least 6 characters** long (**Yup.string().min(6)**), and it is required (**Yup.string().required()**), displaying appropriate messages for violations.
- The **Formik Field** component binds the **input fields** (email and password) to **Formik's internal state**, automatically tracking their values and changes. The **ErrorMessage** component is used to display validation errors next to the fields whenever they violate the Yup validation rules.

@oluwakemi Oluwadahunsi

Pros for using Libraries:

- These libraries **abstract away** most of the **repetitive logic** involved in form handling and validation.

- Libraries like Yup allow for schema-based validation, which is particularly useful for complex **forms with multiple fields and validation rules**.

- Efficiently handles complex forms, multiple validation types, and reusable field components.

Cons:

- Although these libraries simplify form validation, there is a learning curve, especially when using advanced features.

- For small or simple forms, using a library might be **overkill** compared to manual validation.

- You're adding an extra dependency to your project, which may lead to **additional bundle size**.

@oluwakemi Oluwadahunsi

4. Validation Using HTML5 Constraints

Using **HTML5 form validation** provides a quick and basic way to validate forms without custom logic.

For Example:

```html
<form>
  <label htmlFor="email">Email:</label>
  <input type="email" id="email" required />
  <span>Enter a valid email address</span>

  <label htmlFor="password">Password:</label>
  <input type="password" id="password" minLength="6" required />
  <span>Password must be at least 6 characters</span>

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

Required Fields: Use the **required** attribute to ensure that the field must be filled out.

Pattern Matching: The **type="email"** input automatically checks for a valid email format.

@oluwakemi Oluwadahunsi

Pros for using HTML5 validation:

- **No JavaScript required:** Validation is handled entirely by the browser.
- **No JavaScript required:** Validation is handled entirely by the browser.
- **Simple to implement:** It’s straightforward to use and requires minimal coding effort.

Cons:

- **Limited customization:** You have less control over how validation errors are displayed and managed.
- **Not cross-browser consistent:** Some browsers might implement HTML5 form validation differently, leading to inconsistent user experiences.
- **Complex validations:** Handling complex validation rules (e.g., custom validation rules or cross-field validations) is difficult.
- **User experience:** Native validation messages are not as customizable, and the UX might not match the rest of your app.

@oluwakemi Oluwadahunsi

Handling Asynchronous Validation

Handling asynchronous validation in React forms is important when you need to **validate data that requires external checks**, such as ensuring a username is unique or an email address is already registered.

This type of validation often involves making a request to an external server to check the validity of the input.

Example of an Asynchronous validation with hooks:

@oluwakemi Oluwadahunsi

```jsx
import React, { useState, useEffect, useRef } from 'react';

// Simulate an API call to check if the username is taken
const checkUsername = async (username) => {
  const existingUsers = ['grace_jans', 'grace_jans'];
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (existingUsers.includes(username)) {
        reject('Username is already taken');
      } else {
        resolve('Username is available');
      }
    }, 1000);
  });
};

function AsyncValidationWithHooks() {
  const [username, setUsername] = useState('');
  const [error, setError] = useState('');
  const [isSubmitting, setIsSubmitting] = useState(false);
  const [isChecking, setIsChecking] = useState(false);
  const isMounted = useRef(true);

  useEffect(() => {
    return () => {
      isMounted.current = false;
    };
  }, []);

  // Handle username validation asynchronously
  const handleUsernameChange = (e) => {
    setUsername(e.target.value);
    setError('');
    setIsChecking(true);

    // Delay validation to simulate typing and reduce API calls
    const validateUsername = async () => {
      try {
        await checkUsername(e.target.value);
        if (isMounted.current) {
          setError('');
        } catch (err) {
          if (isMounted.current) {
            setError(err);
          }
        } finally {
          if (isMounted.current) {
            setIsChecking(false);
          }
        }
      };

      // Run the asynchronous validation
      validateUsername();
    };
```

continued on next page

```jsx
const handleSubmit = (e) => {
    e.preventDefault();

    // Avoid submission if still checking or if there's an error
    if (isChecking || error) {
      return;
    }

    setIsSubmitting(true);
    console.log('Form Submitted:', { username });
    setIsSubmitting(false);
  };

  return (
    <form onSubmit={handleSubmit}>
      <div>
        <label>Username:</label>
        <input
          type="text"
          value={username}
          onChange={handleUsernameChange}
          placeholder="Enter username"
        />
        {isChecking && <p>Checking username...</p>}
        {error && <p style={{ color: 'red' }}>{error}</p>}
      </div>

      <button type="submit" disabled={isSubmitting || isChecking || !!error}>
        {isSubmitting ? 'Submitting...' : 'Submit'}
      </button>
    </form>
  );
}

export default AsyncValidationWithHooks;
```

@oluwakemi Oluwadahunsi

- We use **useState** to track the **username**, any error messages (like if the username is already taken), **isSubmitting** to handle the form submission state, and **isChecking** to manage the validation state when checking for username availability.

- The **handleUsernameChange** function is triggered every time the user types in the input field. It resets the error message and sets **isChecking** to true, indicating that the asynchronous validation is in progress.

- A **validateUsername** function is created inside **handleUsernameChange** to simulate an asynchronous request (via the **checkUsername** function) that checks if the **username** is available.

- If the **API** call succeeds, the error is cleared, and if it fails (the username is taken), the error message is set. **isChecking** is set to false once the validation process completes, regardless of success or failure.

@oluwakemi Oluwadahunsi

- We use **useRef** to track whether the component is still **mounted** during asynchronous operations. This prevents React from trying to update the state if the component unmounts before the API call completes, which would otherwise lead to **memory leaks**.

- The **handleSubmit** function checks if **isChecking** or error exists. If either is true, the form won't be submitted. If the username passes validation, the form is successfully submitted, and **isSubmitting** is set to true to indicate the submission process.

The **Asynchronous** validation is a complex process that involves both front and backend logics. This is **not beginner-friendly** so, I will advice not to start with this method of validation if you are just starting out with React.

@oluwakemi Oluwadahunsi

Best Practices for Form Validation in React

1. Minimize Re-Renders: Use libraries like React Hook Form to minimize form re-renders and improve performance.

2. Usability: Ensure that forms are easy to interact with, with clear labels, large clickable areas, and responsive designs.

3. Accessibility: Show Clear and Accessible Error Messages: Error messages should be concise, informative, and accessible to all users, including those using screen readers by using ARIA attributes.

4. Give Immediate Feedback: Always provide real-time feedback for validation errors, making the form more interactive and user-friendly.

@oluwakemi Oluwadahunsi

I hope you found this material useful and helpful.

Remember to:

Like

Save for future reference

&

Share with your network, be helpful to someone 👌

@oluwakemi Oluwadahunsi

Hi There!

Thank you for reading through

Did you enjoy this knowledge?

💼 Follow my LinkedIn page for more work-life balancing and Coding tips.

🌐 LinkedIn: Oluwakemi Oluwadahunsi

kodemaven-portfolio.vercel.app