# Exercise: Implement Form Logic in the UserForm Component

## Objective:

Given a pre-structured `UserForm` component, your goal is to complete the logic for the input change handling and form submission. This exercise aims to enhance your understanding of state management, event handling, and form validation in React.

## Initial Component:

You are provided with a `UserForm` component that contains the structure for a user registration form. The form has three fields: username, email, and password. Along with this, the component uses state to manage form data ( `formData` ) and any errors that may occur during form validation ( `errors` ).

## Requirements:

1. **Input Change Handling ( `handleInputChange` function) Overview:**
   The `handleInputChange` function is responsible for updating the component's state when a user types into or modifies any input field in the form. Its two main objectives are:
2. To update the form data based on the user's input.
3. To clear any error messages associated with the field being edited.

**Instructions:**

1. **Capture the Event:**
   - The function takes the event as an argument.
2. **Extract Relevant Data:**
   - From the event object, extract the `name` and `value` properties of the target input element.
   - The `name` corresponds to the attribute of the input field (like 'username', 'email', or 'password'), and the `value` is the current content of that input field.
3. **Update Form Data:**

- Use the component's state update function `setFormData` to modify the state.
- Maintain the previous state values while updating the particular field's value that the user is editing. This ensures that only the current field's data changes while the other fields' data remain intact.
- To achieve this, spread out the previous state and then override the value for the specific field using the extracted `name` as the key and the extracted `value` as its value.

```
setFormData(prevState => ({
  ...prevState,
  [name]: value
}));
```

4. **Clear Associated Errors:**

   - After updating the form data, it's a good practice to clear any validation error messages associated with the field being edited. This is to ensure that as a user is correcting their input, they don't continue to see the old error message.
   - Use the component's state update function `setErrors` to modify the error state.
   - Similarly, spread out the previous error state to maintain the existing errors for other fields.
   - Then, set the error for the field being edited to `null`, indicating there's no error for that field at the moment.

```
setErrors(prevState => ({
  ...prevState,
  [name]: null
}));
```

5. **Form Submission Handling (`handleSubmit` function)**

   **Overview:**
   The `handleSubmit` function is triggered when the user attempts to submit the form. Its primary roles are to validate the user's input data and, if everything is valid, proceed with the desired submission action. Otherwise, it should inform the user about any validation errors.

   **Instructions:**

   1. **Prevent Default Form Submission:**

- As the first action, prevent the default behavior of the form submission. This ensures that the page doesn't reload or navigate away, which is the standard behavior for HTML forms.

2. **Initialize Validation Variables:**

   - Set up an initial flag variable `isValid` that indicates the form's validity as `true`. This flag will be used later to decide if the form can be successfully submitted or not.
   - Create an empty errors object `newErrors` that will store any validation error messages and will be set as the state of `errors` state.

3. **Username Validation:**

   - Check the length of the username entered by the user.
   - If the username length is less than the required minimum (e.g., less than 3 characters), then:
     - Update the errors object to include a message about the username's error.
     - Change the form's validity flag to `false`.

```
if (formData.username.length < 3) {
  newErrors.username = "Username must be at least 3 characters long.";
  isValid = false;
}
```

4. **Email Validation:**

   - Define the criteria for a valid email format. Use a regular expression.
   - Test the user's email input against this criteria.
   - If the email does not match the valid format, then:
     - Update the errors object to include a message about an email error, similar to what you did in the previous step.
     - Set the form's validity flag to `false`.

5. **Password Validation:**

   - Define the criteria for a valid password. This often includes requirements like a minimum length, the inclusion of uppercase and lowercase letters, numbers, and special characters. Again, use a regular expression for this.

- Test the user's password input against this criteria.
- If the password does not meet the requirements, then:
  - Update the errors object to include a message about the required password format, similar to what you did in the previous steps.
  - Set the form's validity flag to `false`.

6. **Update Errors State:**

   - Now that validation checks are complete, set the component's `errors` state with the constructed errors object. This will display the validation messages to the user, if any are present.

```
setErrors(newErrors);
```

7. **Successful Submission Action:**

   - After all validations are done, check the form's validity flag.
   - If the form is valid (i.e., the validity flag remains `true`), Alert a message declaring the successful form submission. In a real life scenario, this step could involve sending data to a server, navigating to a new page, or simply displaying a success message.

# Constraints:

- Do not modify the component structure.
- Assume all the necessary imports and component declarations are provided.

# Hints:

- For email and password validation, use regular expressions ( `regex` ).
- This exercise is a refactor of the same exercise in the DOM section.

# Evaluation Criteria:

- Correctness: Ensure that your logic behaves as described.
- Code Quality: Write clean, readable, and concise code.