Convert the template-driven form to a reactive form.

The Reactive form:

- **TypeScript File:**
    - **Imports and Setup:** Uses Angular's Reactive Forms API, including FormBuilder, Validators, and FormArray.
    - **Custom Validator:** Implements a custom validator to reject names containing "test."
    - **Form Model:** Builds a reactive form with controls for name, email, and a dynamic FormArray for hobbies.
    - **Methods:** Provides methods to add new hobbies and submit the form while resetting it afterward.
- **HTML File:**
    - **Form Binding:** Uses `[formGroup]` for binding the reactive form.
    - **Input Controls:** Binds individual controls using `formControlName`, displays error messages based on control state, and dynamically handles a FormArray for hobbies.
    - **Submission:** Incorporates a submit button that is disabled until the form is valid.
- **CSS File:**
    - **Visual Enhancements:** Styles the form container, labels, inputs, error messages, and buttons to ensure a modern, clean, and responsive interface.

1. TypeScript File: user-form.component.ts

Purpose:
This file defines the reactive form's structure and behavior. It uses Angular's Reactive Forms API, including the FormBuilder to simplify form creation, a custom validator function for additional validation rules, and a FormArray to allow for a list of dynamic inputs (for example, hobbies).

Step-by-Step Instructions:

Import Necessary Modules:
Import Angular's Reactive Forms classes as well as `Component` from `@angular/core`:

```
import { Component } from '@angular/core';

import { FormBuilder, FormGroup, FormControl, Validators,
FormArray, AbstractControl } from '@angular/forms';
```

Define a Custom Validator:
Create a function that validates a control. For example, a simple custom validator that ensures the name does not contain the word “test”:

```
function forbiddenNameValidator(control: AbstractControl):
{ [key: string]: any } | null {

  const forbidden = /test/i.test(control.value);
  return forbidden ? { 'forbiddenName': { value:
control.value } } : null;
}
```

Setup the Component Decorator:
Define the component as a standalone component and include ReactiveFormsModule (if using Angular standalone components, otherwise include it in your module). Also point to the external HTML and CSS files:

```
@Component({

  selector: 'app-user-form',
  standalone: true,
  imports: [],
  templateUrl: './user-form.component.html',
  styleUrls: ['./user-form.component.css']
})
```

1. **Build the Form in the Component Class:**
   Use FormBuilder in the constructor to create the form.

- Create a FormGroup with controls for `name` and `email` (including built-in Validators such as `required` and `email`).
- Apply the custom validator on the `name` control.
- Create a FormArray (e.g., for “hobbies”) that starts empty but can be added to dynamically.

```
export class UserFormComponent {
  userForm: FormGroup;

  constructor(private fb: FormBuilder) {
    this.userForm = this.fb.group({
      name: ['', [Validators.required,
forbiddenNameValidator]],
      email: ['', [Validators.required, Validators.email]],
      hobbies: this.fb.array([]) // FormArray for dynamic
hobbies
    });
  }

  // Getter to access hobbies form array in the template
  get hobbies(): FormArray {
    return this.userForm.get('hobbies') as FormArray;
  }

  // Function to add a new hobby to the form array
  addHobby(): void {
    this.hobbies.push(this.fb.control('',
Validators.required));
  }

  // Submit handler that logs the form value and resets the
form
  onSubmit(): void {
    if (this.userForm.valid) {
      console.log('Form Submitted!', this.userForm.value);
```

Below is the continuation of the detailed instructions for converting your template-driven form into a reactive form using FormBuilder, a custom validator, and a FormArray. This guide explains the remaining parts—the HTML and CSS files—along with code hints and detailed instructions.

2. HTML File: user-form.component.html

Purpose:
This template connects to the reactive form defined in your TypeScript file. It binds form controls, displays validation errors, and provides dynamic handling of a list of inputs (for example, hobbies).

Step-by-Step Instructions & Code Hints:

1. **Binding the FormGroup:**
   - Use Angular's reactive form binding with the `[formGroup]` directive on the `<form>` element.
   - Set up the `(ngSubmit)` event to trigger your `onSubmit()` method.

   Example hint:

```
<form [formGroup]="userForm" (ngSubmit)="onSubmit()"
novalidate>
  <!-- Form controls will go here -->
</form>
```

2. **Name Field with Custom Validation:**
   - Bind the input for the name field using `formControlName="name"`.
   - Display an error message if the field is invalid. You can check for both built-in and custom errors.

   Code hint:

```
<div>
  <label for="name">Name:</label>
  <input id="name" type="text" formControlName="name" /
>
  <!-- Error message for required or forbidden name -->
  <div *ngIf="userForm.get('name')?.touched &&
userForm.get('name')?.invalid">
```

```
    <span *ngIf="userForm.get('name')?.errors?.
['required']">Name is required.</span>
    <span *ngIf="userForm.get('name')?.errors?.
['forbiddenName']">Name cannot contain "test".</span>
  </div>
</div>
```

3. **Email Field with Built-In Validators:**

   - Bind the email field using `formControlName="email"`.
   - Display validation errors for required or invalid email format.

Code hint:

```
<div>
  <label for="email">Email:</label>
  <input id="email" type="email"
formControlName="email" />
  <div *ngIf="userForm.get('email')?.touched &&
userForm.get('email')?.invalid">
    <span *ngIf="userForm.get('email')?.errors?.
['required']">Email is required.</span>
    <span *ngIf="userForm.get('email')?.errors?.
['email']">Enter a valid email.</span>
  </div>
</div>
```

4. **Dynamic FormArray (Hobbies):**

   - Use a container (e.g., `<div>`) and the `*ngFor` directive to iterate over the FormArray controls.
   - Bind each hobby input with `formControlName` (index-based when inside the `FormArray`).
   - Include a button to add a new hobby.

Code hint:

```
<div formArrayName="hobbies">
  <label>Hobbies:</label>
```

```
  <div *ngFor="let hobby of hobbies.controls; let i =
index">
    <input [formControlName]="i" placeholder="Enter
hobby" />
    <!-- Optionally add a remove button for each hobby
-->
  </div>
  <button type="button" (click)="addHobby()">Add
Hobby</button>
</div>
```

5. **Submit Button:**
   - Add a submit button that triggers the form submission.
   - The button can be disabled if the form is invalid.

   Code hint:

```
<button
type="submit" [disabled]="userForm.invalid">Submit</
button>
```

3. CSS File: user-form.component.css

Purpose:
This file styles the reactive form to make it visually appealing and user-friendly.

Step-by-Step Instructions & Code Hints:

1. **Form Container Styling:**
   - Style the `<form>` element to set max-width, center it on the page, add padding, borders, and a subtle shadow.

   Code hint:

```
form {
  max-width: 500px;
  margin: 2rem auto;
```

```
  padding: 2rem;
  border: 1px solid #ccc;
  border-radius: 8px;
  box-shadow: 0 4px 6px rgba(0, 0, 0, 0.1);
  background-color: #fff;
}
```

2. **Labels and Input Fields:**
    - Ensure labels are clearly visible, inputs are full-width, and add spacing.

Code hint:

```
label {
  display: block;
  margin-bottom: 0.5rem;
  font-weight: bold;
  color: #333;
}

input {
  width: 100%;
  padding: 0.75rem;
  margin-bottom: 1rem;
  border: 1px solid #ddd;
  border-radius: 4px;
  box-sizing: border-box;
}

input:focus {
  border-color: #007bff;
  outline: none;
}
```

3. **Error Messages:**
    - Style error messages with a distinct color and smaller font size.

Code hint:

```
.error {
  color: #d9534f;
  font-size: 0.85rem;
  margin-top: -0.5rem;
  margin-bottom: 1rem;
}
```

4. **Button Styling:**
    - Style the submit button to be modern and responsive, including hover and disabled states.

Code hint:

```
button {
  width: 100%;
  padding: 0.75rem;
  background-color: #007bff;
  color: #fff;
  border: none;
  border-radius: 4px;
  cursor: pointer;
  transition: background-color 0.3s ease;
}

button:hover:not(:disabled) {
  background-color: #0056b3;
}

button:disabled {
  background-color: #a0c4ff;
  cursor: not-allowed;
}
```

Create the Email Matcher Validator

We will update three different files:

1. **Custom Validator:**
   - Create an `emailMatchValidator` function in your **custom-validators.ts** file.
2. **Component Update:**
   - Add an `emailGroup` in your reactive form with `email` and `confirmEmail` fields.
   - Apply `emailMatchValidator` as a group-level validator.
3. **Template Update:**
   - Update your template to display both email fields and an error message if they don't match

First, add the custom validator function in your custom validators file (for example, **custom-validators.ts**). This function compares the values of two form controls (typically `email` and `confirmEmail`) and returns an error if they do not match.

Instructions:

- Open your **custom-validators.ts** file (or create it if it doesn't exist).
- Import the necessary types from `@angular/forms`.
- Create a validator function using the `ValidatorFn` type. This function will receive an `AbstractControl` (the parent form group) and check if the two email fields match.
- If the emails do not match, return an error object (e.g. `{ emailMismatch: true }`); otherwise, return `null`.

Code Example:

```
import { ValidatorFn, AbstractControl } from '@angular/
forms';

export const emailMatchValidator: ValidatorFn = (control:
AbstractControl): { [key: string]: any } | null => {
  const email = control.get('email');
  const confirmEmail = control.get('confirmEmail');

  // If either control is missing, no error is returned.
  if (!email || !confirmEmail) {
    return null;
  }
```

```
  // If the emails do not match, return an error.
  return email.value !== confirmEmail.value ?
{ emailMismatch: true } : null;
};
```

2. Update Your Component's Form Group

Next, update your reactive form in your component (for example, **user-form.component.ts**) to include both `email` and `confirmEmail` fields. Then apply the custom validator to the group containing these controls.

Instructions:

- In your component, use `FormBuilder` to create a new `FormGroup` (often nested) for your email fields. This group should contain two controls: one for the primary email and one for confirmation.
- Attach the `emailMatchValidator` as a group-level validator on this nested group.
- Update your overall form to include this email group.

Code Example:

```
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormGroup, Validators, FormArray }
from '@angular/forms';
import { forbiddenNameValidator, emailMatchValidator } from
'./custom-validators';

@Component({
  selector: 'app-user-form',
  standalone: true,
  imports: [],
  templateUrl: './user-form.component.html',
  styleUrls: ['./user-form.component.css']
})
export class UserFormComponent implements OnInit {
  userForm: FormGroup;

  constructor(private fb: FormBuilder) {
```

```
    // Create the form with a nested emailGroup that uses
the emailMatchValidator.
    this.userForm = this.fb.group({
      name: ['', [Validators.required,
forbiddenNameValidator]],
      emailGroup: this.fb.group({
        email: ['', [Validators.required,
Validators.email]],
        confirmEmail: ['', [Validators.required,
Validators.email]]
      }, { validators: emailMatchValidator }),
      hobbies: this.fb.array([])
    });
  }

  ngOnInit(): void {
    // (Optional) Disable or monitor other parts of the
form as needed.
  }

  // Getter for the hobbies FormArray.
  get hobbies(): FormArray {
    return this.userForm.get('hobbies') as FormArray;
  }

  addHobby(): void {
    this.hobbies.push(this.fb.control('',
Validators.required));
  }

  removeHobby(index: number): void {
    this.hobbies.removeAt(index);
  }

  onSubmit(): void {
    if (this.userForm.valid) {
      console.log('Form Submitted!', this.userForm.value);
      this.userForm.reset();
      // Optionally, clear out the hobbies array:
      while (this.hobbies.length) {
```

```
        this.hobbies.removeAt(0);
      }
    } else {
      console.log('Form is invalid');
    }
  }
}
```

3. Update the Template to Display the Email Confirmation Field

Finally, modify your component's HTML template (for example, **user-form.component.html**) to include inputs for both email fields. Also, display an error message if the emails don't match.

Instructions:

- Wrap the email inputs in a `<div>` that uses `formGroupName="emailGroup"` to bind them to the nested form group.
- Add inputs for `email` and `confirmEmail` using `formControlName`.
- Display a message if the `emailGroup` has the `emailMismatch` error.

Code Example:

```
<form [formGroup]="userForm" (ngSubmit)="onSubmit()"
novalidate>
  <!-- Name Field -->
  <div>
    <label for="name">Name:</label>
    <input id="name" type="text" formControlName="name" />
    <div *ngIf="userForm.get('name')?.touched &&
userForm.get('name')?.invalid" class="error">
      <span *ngIf="userForm.get('name')?.errors?.
['required']">Name is required.</span>
      <span *ngIf="userForm.get('name')?.errors?.
['forbiddenName']">
        Name cannot contain "admin" or "test".
      </span>
    </div>
  </div>
```

```
  <!-- Email Group -->
  <div formGroupName="emailGroup">
    <div>
      <label for="email">Email:</label>
      <input id="email" type="email"
formControlName="email" />
      <div *ngIf="userForm.get('emailGroup.email')?.touched
&& userForm.get('emailGroup.email')?.invalid"
class="error">
        <span
*ngIf="userForm.get('emailGroup.email')?.errors?.
['required']">Email is required.</span>
        <span
*ngIf="userForm.get('emailGroup.email')?.errors?.
['email']">Enter a valid email.</span>
      </div>
    </div>
    <div>
      <label for="confirmEmail">Confirm Email:</label>
      <input id="confirmEmail" type="email"
formControlName="confirmEmail" />
      <div
*ngIf="userForm.get('emailGroup.confirmEmail')?.touched &&
userForm.get('emailGroup.confirmEmail')?.invalid"
class="error">
        <span
*ngIf="userForm.get('emailGroup.confirmEmail')?.errors?.
['required']">Confirmation is required.</span>
        <span
*ngIf="userForm.get('emailGroup.confirmEmail')?.errors?.
['email']">Enter a valid email.</span>
      </div>
    </div>
    <!-- Error for mismatched emails -->
    <div *ngIf="userForm.get('emailGroup')?.errors?.
['emailMismatch']" class="error">
      Emails do not match.
    </div>
  </div>
```

```
  <!-- Hobbies and Submit Button -->
  <div formArrayName="hobbies">
    <!-- (hobbies code goes here) -->
  </div>

  <button
type="submit" [disabled]="userForm.invalid">Submit</button>
</form>
```

-