LET’S CREATE A TEMPLATE-DRIVEN CONTACT FORM

Start with a new project, we will merge these files later with routing.

1. TypeScript File: app.component.ts

Purpose:
This file contains the logic for your Angular component. It defines the data model, handles form submission, and resets the form once the data is submitted.

Key Steps & Instructions:

- **Importing Modules:**
    - Import `FormsModule` from `@angular/forms` because it provides the directives and services necessary for template-driven forms.

- **Component Class & Data Model:**
    - The class (`UserFormComponent`) contains a `user` object that holds the properties for form fields (name and email).
    - This object is used with Angular's two-way binding so that any changes in the form inputs automatically update the model and vice versa.

- **Form Submission Handling:**
    - The `onSubmit(form: any)` method is called when the form is submitted.
    - Inside this method, the code checks if the form is valid (using `form.valid`), and if so:
        - It logs the user data to the console.
        - It calls `form.reset()` to clear all form fields.
        - Optionally, it resets the `user` object to ensure that the component's data model is also cleared.
    - If the form is invalid, it logs an appropriate message.
    -

Code Hints:

- **Import and Component Setup:**
  Start by importing Angular's `Component` and `FormsModule`.

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

In the `@Component` decorator, set `standalone: true` and add `FormsModule` to the `imports` array:

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [FormsModule],
  // ... (templateUrl and styleUrls)
})
```

- **Defining the Data Model:**
  Create a `user` object that holds properties for your form fields:

```
user = {
  name: '',
  email: ''
};
```

- **Form Submission Method:**
  Write an `onSubmit` method that first checks if the form is valid, logs the data, then resets the form:

```
onSubmit(form: any): void {
  if (form.valid) {
    console.log('Form Submitted!', this.user);
    form.reset(); // Clears the form fields
    this.user = { name: '', email: '' }; // Optionally
reset the model
  } else {
    console.log('Form is invalid');
  }
}
```

2. HTML File: app.component.html

Purpose:
This file defines the structure and layout of the form that users interact with. It binds the UI elements to the data model and sets up validations.

Key Steps & Instructions:

- **Form Setup:**
  - Begin with a `<form>` element that has a template reference variable (e.g., `#userForm="ngForm"`). This variable gives you access to the form's properties (like its validity status).
  - Use the `(ngSubmit)` event binding to link the form submission to the `onSubmit` method defined in the TypeScript file. The `novalidate` attribute is included to prevent the browser's default validation so that Angular's validations can take over.
- **Input Fields with Data Binding and Validation:**
  - **Name Field:**
    - Create a label and an `<input>` element for the name.
    - Use Angular's two-way binding (`[(ngModel)]`) to bind the input to `user.name`.
    - Add the `required` attribute to ensure that the field must be filled.
    - Define a local template variable (e.g., `#name="ngModel"`) to track the state of this input.
    - Provide an error message that shows up when the field is invalid and has been interacted with (using an `*ngIf` directive). Note: the directives need to be imported with CommonModule.
  - **Email Field:**
    - Similarly, create a label and `<input>` for the email.
    - Bind this field to `user.email` with `[(ngModel)]` and enforce validations using `required` and Angular's built-in `email` validator.
    - Use another local variable (e.g., `#email="ngModel"`) to monitor this control's state and display a corresponding error message when needed.
- **Submit Button:**
  - Add a `<button>` element with a type of `submit`.
  - Use the Angular binding `[disabled]="userForm.invalid"` to disable the button if the form is invalid. This prevents submission until all required validations pass.

Code Hints:

- **Setting Up the Form:**

  Start with a `<form>` element that uses a template reference variable to access the form state:

```
<form
#userForm="ngForm" (ngSubmit)="onSubmit(userForm)"
novalidate>
  <!-- form fields go here -->
</form>
```

- **Creating Input Fields with Validation:**

  Name Input:
  Use `[(ngModel)]` to bind to `user.name` and include the `required` attribute:

```
<input type="text" name="name" required
[(ngModel)]="user.name" #name="ngModel" />
```

  And provide an error hint that appears when validation fails:

```
<div *ngIf="name.invalid && name.touched">Name is
required.</div>
```

  Email Input:
  Similarly, bind the email field and use Angular's built-in validators:

```
<input type="email" name="email" required email
[(ngModel)]="user.email" #email="ngModel" />

<div *ngIf="email.invalid && email.touched">A valid
email is required.</div>
```

Submit Button:

Add a submit button that is disabled if the form is invalid:

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

3. CSS File: app.component.css

Purpose:
This file adds visual styling to the form, making it more user-friendly and visually appealing.

Key Steps & Instructions:

- **Form Container Styling:**
  - Define styles for the `<form>` element to control its width, padding, margin (centering it on the page), and add a border, border-radius, and box-shadow for depth.
  - Set a background color to differentiate the form area from the rest of the page.
- **Label and Input Styling:**
  - Style `<label>` elements to ensure they are clearly visible, with proper spacing and font weight.
  - Set styles for `<input>` fields to use full width, include padding for easier interaction, and a light border with a smooth border-radius.
  - Define a focus state for inputs so that their border changes color when the user is interacting with them. This improves usability and accessibility.
- **Error Message Styling:**
  - Apply a red color and adjust the font size for error messages. This makes it clear when a field has an issue.
  - Add margin adjustments so that the error messages are spaced well relative to the inputs.
- **Submit Button Styling:**

- Style the submit button to use a consistent color scheme (for example, a blue background with white text).
- Ensure the button takes the full width of its container for a modern look.
- Add transition effects and a hover state to improve user interaction.
- Style the disabled state so users see a clear visual indicator when the button is not clickable.