Handling Forms in Angular Apps

Why Do We Need Angular's Help?

- Angular is a **single-page application (SPA)**, meaning there's no submitting to servers
- Instead, Angular handles it with its own HTTP services
- Angular gives us a **JavaScript object representation** of the form
  - This makes it simple to retrieve its contents, see the form's state, and modify it

Template-Driven vs Reactive Approach

- **Template-Driven Approach**
  - Angular infers the Form Object from the DOM
  - Greater resembles native JavaScript forms
- **Reactive Approach**
  - The form is created programmatically and synchronized with the DOM
  - We define the form's structure in **TypeScript**, present it with **HTML**, and **manually** stitch it together
  - Much greater control

Template-Driven: Creating the Form and Registering the Controls

- First, ensure you're importing **FormsModule** inside of your **AppModule** file
  - Automates the creation of Angular **Form** elements when a form is detected → Essentially a **form** selector
- Second, we must add the **ngModel** directive and **name** as attributes to the **input fields**
  - No **[(ngModel)]** (like with two-way data-binding), just **ngModel**
  - **name** is used to refer to the input field

Template-Driven: Submitting and Using the Form

- **NOTE:** The default behavior of buttons inside of forms is **submit**
  - We should include the **(ngSubmit) event** to **form** so Angular decides what happens when the form submission is triggered
- To fetch the form's content, we can insert a **local reference** inside **form**
  - To receive data in an object notation, you must assign the **local reference** to **ngForm** → **#localref="ngForm"**
  - By passing that **local reference** into our (**ngSubmit)="<function>"**, we can access the form and its various contents inside the component
    - The **form**'s type is **HTMLFormElement**
  - To access the form as an object, we input the form, via **(ngClick)**, as type **NgForm**
    - The form contains a field titled **values**, which holds all inputs and their values as key/value pairs

Template-Driven: Understanding Form State

- Logging the form shows us its many states
- **States** and their meanings
  - **value** → Stores user input in key/value pairs
  - **dirty** → True if the form is changed in any way
  - **disabled** → True if the form is disabled
  - **invalid** → False when there aren't any added validators
  - **touched** → Let's us know if we've clicked into fields
  - **etc** → So many more fields, so little time

Template-Driven: Accessing the Form with @ViewChild

- **@ViewChild** → Tells us data about an assigned reference
- Create a variable in the component of type **ngForm** that has the **@ViewChild("<localRef>")** decorator
- This lets us access our form without needing to use **(ngSubmit)**

Template-Driven: Adding Validation to Check User Input

- We should always validate data on the **Back-End** as the **Front-End** can be tricked
- Validating input in the **Front-End** makes matters convenient for the users
- In a template-driven approach, we can only add validators to the template
  - Add validators as an **HTML Attribute** within the inputs
  - Automatically analyzed by Angular for proper usage
- Angular naturally applies CSS classes depending on a field's validation
- Types of directive validators, native or not
  - **required** → The field must be filled-out for the form to be submitted
  - **email** → The input must match the form of a valid email; tracks on the form and control levels

Built-In Validators and Using HTML5 Validation

- Angular naturally disables HTML5 validation; however, we can reactivate it by adding **ngNativeValidate** to template controls

Template-Driven: Using the Form State

- Again, we can use the form state to alter our form
  - **[disabled]="!<local-ref>.valid"**
- We can also modify the CSS values of the classes Angular toggles on forms and elements, depending on their states
  - **.ng-invalid { border: 1px solid red; }**
  - More specific → **input.ng-invalid { border: 1px solid red; }**
  - Better, since it doesn't show at the start → **input.ng-invalid.ng-touched { border: 1px solid red; }**

Template-Driven: Outputting Validation Error Messages

- Bootstrap Approach
  - Apply a **span** below the **input** element with the **help-block** class

    - To access the control's state, add a **local reference** to the input, and set it to **ngModel**
    - Within the **span**, use a ***ngIf** to output the template if **<local-ref>.valid**
        - Again, use **<local-ref>.touched** to not display warnings upon startup

Template-Driven: Set Default Values with ngModel Property Binding

- We can **property-bind** each input element's **ngModel**
    - We can either write a default value directly into the template, or refer to a variable (not by string interpolation) from the component

Template-Driven: Using ngModel with Two-Way Binding

- We can use **two-way binding** to check or repeat user input
- **ngModel** should be two-way bound in the template → **[(ngModel)] = "<variable>"**
- **[(ngModel)]** should refer to a variable in the template
- The variable in the template can be outputted via **string interpolation** to the template

Template-Driven: Grouping Form Controls

- Groups are good for logically organizing our fields, as well as validating entire groups of inputs as opposed to purely atomic elements
- To group form controls, we should apply the **ngModelGroup** directive to element housing appropriately-grouped elements
    - **ngModelGroup** must be assigned to a **string**
    - This is the identifier of the group for when it communicates with the component
- We can also fetch the group's **JSON** by adding a **local reference** to the same housing element and setting it to **ngModelGroup**

    - This can be used later for outputting the validation status of the form group

Template-Driven: Handling Radio Buttons

- We should write an array of all possible values in the component
- In the template, create a **div** with a class **radio** and ***ngFor** that iterates over the array of values
- Inside of the **div** should be a **label** of type **radio**
    - The label should use **string interpolation** with the current value variable of the ***ngFor** loop to attach the correct label
    - We can use **property-binding** with **ngModel** to set a default value
    - It must also have the **[value]** attribute/directive to match correct values

Template-Driven: Setting and Patching Form Values

- It may be nice to automatically populate an input field
- First method
    - Apply the **this.<view-child-form-name>.setValue()** method
    - Takes in **JSON** that represents the whole form with the ideal data
    - Bad as it overwrites existing data
- Second method
    - Apply the **patchValue()** method
    - Done using **this.<form-name>.form.patchValue({})**
    - Takes in **JSON**, but only include the fields you wish to change

Template-Driven: Using Form Data

- To extract form data, you must follow the **value** property of your defined form
- Example → **this.<form-name>.value.<field-name>**

Template-Driven: Resetting Forms

- First method → Call **this.<form-name>.reset()** to reset the whole form
- Second method → Use the **setValue()** method for every individual field

Introduction to the Reactive Approach

- The form is created programmatically and synchronized with the DOM

Reactive: Setup

- In Angular, a form is merely a group of controls and their contents
- **Reactive** forms are of type **FormGroup**
- Must import **ReactiveFormsModule** under **imports** in **AppModule**

Reactive: Creating a Form in Code

- We must instantiate our **FormGroup** in the component
  - It takes a JavaScript **object** as an argument
  - Controls are **key/value** pairs listed in the object
    - Every value is a **new FormControl**
      - First argument is the default value
      - Second argument pertains to validators
      - Third argument concerns potentially asynchronous validators

Reactive: Syncing HTML and Form

- Naturally, Angular doesn't know that component forms relate to template forms
- To sync them, we must include certain **directives** into the template
  - In the form, include the **[formGroup]** directive, and assign it to the **FormGroup** in the component
  - To get the inputs, we must include the **formControlName** directive onto the appropriate form inputs with their assigned control values

Reactive: Submitting the Form

- We must still include **(ngSubmit)** to the form
- We don't need to add a local reference because we created the form independently

Reactive: Adding Validation

- The **second argument** for each form control pertains to **validators**
- This argument should be an array of **Validators** objects
  - Each object can invoke a specific validation using **dot-notation**
    - Don't add the parentheses

Reactive: Getting Access to Controls

- To access controls, we use the **get** method
  - Example → **signUpForm.get('<control-name/path>')**

Reactive: Grouping Controls

- We can group controls by nesting **FormGroups**
- To refer to the controls within a nested **FormGroup**, we must use **dot-notation**
- We have to group inputs together in the template as well. including the **formGroupName** directive on the housing **div**
  - Assign this value to the nested **FormGroup** name in the component

Reactive: Arrays of Form Controls (FormArray)

- **Form Arrays** hold arrays of controls
- They accept an **array** as an argument
  - Empty defaults to no starting controls
- Must add the **formArrayName** directive (assigned to the corresponding name in the form) to the **div** housing the array in the template
- To reference the **FormArray** elsewhere in the component, you must prepend the **getter** with **<FormArray>**
- To add the control, merely push it to the array
- Example → **(<FormArray>this.signUpForm.get('hobbies')).push(<new-control>);**

Reactive: Creating Custom Validators

- **Validators** are merely functions that Angular executes when it checks a form control's validity
- Every validator receives the control to check as an argument → **<validator-name>(<parm>: FormControl)**
- Validators must also return **JavaScript** objects
  - **Keys** can be interpreted as a **string**
  - **Values** must be **booleans**

- If the validator **fails**, then return a **JavaScript** object containing the **failure code** (string) and **true** (boolean)
  - Else, you must return **null** → **NOT FALSE!**
- To apply the validator, add the function to the list of validators
- **this.<validator-name>.bind(this)**

Reactive: Using Error Codes

- We can check if a field has a validation error by looking at the specific validator's error log
- Example → **signUpForm.get('username').errors['nameIsForbidden']**

Reactive: Creating a Custom Async Validator

- **Asynchronous validators** await a response before determining if a value is valid
- Similar to synchronous validators, it takes a **FormControl** as an argument
- Unlike synchronous validators, it returns a **Promise** or **Observable**
- You must instantiate a **new Promise**, resolve it properly, and return it
  - Similar to synchronous validators, you resolve a JavaScript **object** with an **error name** and **boolean** value for **failed** validation, and **null** for **passed** validation

Reactive: Reacting to Status or Value Changes

- Status and value changes are both **observables**
- We can **subscribe** to them to determine their values

Reactive: Setting and Patching Values

- We can set values using the **<form-name>.setValue()** method
- It takes a JavaScript object that matches the structure of the form
- **<form-name>.patchValue()** is similar
  - This updates part of the form
- Can **reset** the form with **<form-name>.reset()**