Angular Forms

Learn how to accept user input with Angular's FormsModule.

Overview

There are three steps when using forms with Angular. Details on each are found below.

1. Import the FormsModules into the component in order to enable the forms features.
2. Create HTML inputs in the template. For each input, also create a matching property in the TypeScript class that will track its value.
3. Provide a method for handling form submission.

Import

To use Angular Forms, you must import it per component. You will see it **Bolded and Underlined** in the code example.

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

Bind inputs to properties (two way binding)

Two way binding variables will see the values update whenever the value is changed either in the TypeScript or the HTML. Changing one will change the other.

First let's create a variable that we want to use.

```
city:string = "Detroit";
```

Once you have some variables setup, we can bind them to the HTML inputs using the ngModel directive.

A good way to remember [(ngModel)] is the term Bananas in a Box. The way it's shaped will connect to the saying.

```
<p>Current City: {{city}}</p>

<input type="text" name="city" [(ngModel)]="city" />
```

Now the city property will be updated whenever the user types into the textbox and if whenever we set the city property in code, it will also update what's in the textbox to match.

Numbers

This is very similar to the last example. The main difference is that we want to use the number data type and the number input type.

```
resultLimit:number|null = null;
```

```
<input type="number" name="resultLimit" [(ngModel)]="resultLimit" />
```

The built-in validation attributes for number inputs work here as well, such as min and max.

```
<input type="number" name="resultLimit" [(ngModel)]="resultLimit"
min="0" max="100" />
```

Usage with Select/Options

For <select> elements, we typically need both a variable for binding the value and an array of options.

Here I've created a list of choices and a variable to hold their choice.

```
  flavors:string[] = [
    "Mocha",
    "Latte",
    "Black"
  ];

  choice:string = "";
```

In the HTML I've created a display for the choice as well as a Select/Options to allow them to choose the flavor. Notice how the ngModel is tied to the choice while the options are tied to the array.

```
<p>Current Drink Choice: {{ choice }}</p>

<select name="choice" [(ngModel)]="choice">
    @for (f of flavors; track $index) {
        <option [value]="f">{{ f }}</option>
    }
</select>
```

Checkbox

Checkboxes are best saved for boolean values. If Angular detects a boolean, it will set the value to either true or false based on if the checkbox is clicked.

```
vegan:boolean = false;
```

Connecting a label is highly encouraged for accessibility and usability. You can tie it to the input by either (a) connecting the input's id attribute to the label's for attribute or (b) wrapping the input in the label.

```
<input type="checkbox" name="vegan" [(ngModel)]="vegan" id="veganCheck" />
<label for="veganCheck">Are you vegan?</label>

<label>
  <input type="checkbox" name="vegan" [(ngModel)]="vegan"/> Are you vegan?
</label>
```

Radio Buttons

Radio Buttons are best used for one of two things. Either selecting one of the hard coded choices or one of the values of an array.

```
milkChoice:string="";
```

Radio buttons are grouped together by sharing a common name and ngModel.

Again, the label is strongly encouraged. You can tie it to the input by connecting the input's id attribute to the label's for attribute or wrapping the input in the label.

```
<input type="radio" name="milkChoice" [(ngModel)]="milkChoice"
value="whole" id="whole"/>
<label for="whole">Whole?</label>

<input type="radio" name="milkChoice" [(ngModel)]="milkChoice"
value="oat" id="oat"/>
<label for="oat">Oat?</label>

<input type="radio" name="milkChoice" [(ngModel)]="milkChoice"
value="soy" id="soy"/>
<label for="soy">Soy?</label>
```

If you have an array of values, we can use that instead.

```
favSyrup:string="";
syrupChoices:string[] = ["Caramel","Vanilla","Peppermint"];
```

This time, we use a for loop and use property binding to help assigning everything

```
@for(s of syrupChoices; track $index) {
      <label [for]="s">{{ s }}</label>
      <input type="radio" name="favSyrup" [(ngModel)]="favSyrup"
            [value]="s" [id]="s"/>
}
```

The name attribute

When using ngModel inside a form, it is also required to include a name attribute. This does not have to match ngModel, but it must be present for an input with ngModel to work within a form.

```
<form (ngSubmit)="doTheThing()">
    <!-- Works -->
    <input name="displayName" [(ngModel)]="profile.displayName" />
    <!-- Works -->
    <input name="name" [(ngModel)]="profile.displayName" />
    <!-- Does not work -->
    <input [(ngModel)]="profile.displayName" />
</form>
```

Handling form submission

In order to handle form submission:

1. Create a method in the TypeScript class to handle the submission. Within this method, use the values from the bound properties (see above).
2. Add a <button type="submit"> to the form.
3. Wrap the inputs and button in a <form> tag.
4. Add an (ngSubmit) event handler to call the method you created.

```
// properties bound to inputs
name: string = "";
age: number = NaN;
// submit handler
```

```
submitForm():void{
  console.log(`Name is ${this.name}, age is ${this.age}.`);
}
```

```
<form (ngSubmit)="submitForm()">
    <input name="name" [(ngModel)]="name" />
    <input name="age" [(ngModel)]="age" type="number" />
    <button type="submit">Submit</button>
</form>
```

Note that we use a submit handler on the form rather than a click handler on the button. This ensures we can take advantage of the form's built-in features including HTML5 validation and quick submission using the ENTER key.

Clear the form

When you are finished accepting the form, it is common to clear the form after. Since we are binding our inputs to variables, we can just reset the variables.

```
// properties bound to inputs
name: string = "";
age: number = NaN;
// submit handler
submitForm():void{
  console.log(`Name is ${this.name}, age is ${this.age}.`);
  this.name = "";
  this.age = NaN;
}
```

Full form example

Angular goes much deeper into forms than what we are covering here. This will be a working form, but you can actually do a lot with forms. Check Advanced Forms for more details once you have mastered this style of form.

For this, I've created pet.ts

```
export interface Pet {
    name: string;
    species: string;
    age: number;
}
```

Back in the component, we created two properties. One will be an array of all pets we make and the other will be used in the form that we create pets in. I've also created a method we will call that will add the pet to the array of pets. This is buggy. We will come back to fix it.

```
AllPets:Pet[] = [];
Pet: Pet = {} as Pet;
//buggy method
addPet():void{
  this.AllPets.push(this.Pet);
}
```

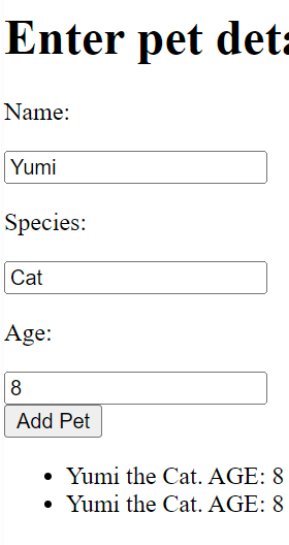
Now in the HTML, I will be creating a form. This form will contain multiple two way binded inputs and a button with a submit event attached. I've also created a display for the array.

```
<!-- Form -->
<form (ngSubmit)="addPet()">
    <h1>Enter pet details!</h1>
    <p>Name:</p>
    <input type="text" name="name" [(ngModel)]="Pet.name" />
    <p>Species:</p>
    <input type="text" name="species" [(ngModel)]="Pet.species" />
    <p>Age:</p>
    <input type="number" name="age" [(ngModel)]="Pet.age" />

    <button>Add Pet</button>
</form>
```

```
<!-- See Pets array -->
<ul>
    @for (p of AllPets; track p) {
        <li>{{p.name}} the {{p.species}}. AGE: {{ p.age }}</li>
    }
</ul>
```

Now save and try it out! It works, but as you add a second and third, you'll notice the bug right away.



The inputs are still bound to the values in the array. That's because the two way binding is connected to its memory location. To fix this, we need to create a new variable to use a different spot in memory. I'm using the spread operator to speed up the process of manually assigning the values to the new object.

```
  addPet():void{
    let newPet: Pet = {...this.Pet};
    this.AllPets.push(newPet);
    this.Pet = {} as Pet; // This will reset the form
  }
```

Now the bug should be fixed.

HTML5 validation

Modern browsers build in native validation features that handle most cases pretty well. You may consider relying on this validation. The form will not be able to submit if there are validation errors and user-friendly error messages will appear next to the invalid fields.

First, make sure you're using the right input types (text, number, email, etc.). This provides some automatic validation.

Second, add validation attributes to inputs as needed. Here are some options.

- required
- minlength
- maxlength
- min (for numbers)
- max (for numbers)
- step (for numbers) - e.g. "0.01" means the value cannot be more precise than hundredths.
- pattern - provide a regular expression

```
<p>
  <label for="price">Price</label>
  <input name="price" id="price" [(ngModel)]="price" type="number"
         required min="0" step="0.01" />
</p>
```

WARNING: Keep in mind that you always need to do full validation on the server side because even if your client-side validation is perfect, hackers can go around that and send requests to the server directly, bypassing your client-side validation.

Advanced forms

The style we use here is called template-driven forms. Another technique with more features is reactive forms. We will not cover reactive forms in this class.

- Template-driven forms (simpler, but don't scale well). Learn more: https://angular.dev/guide/forms/template-driven-forms

- Reactive Forms (more advanced, but more robust). Learn more: https://angular.dev/guide/forms/reactive-forms

Some other form considerations which we will not cover in-depth here are:

- Advanced validation
- Error handling

Additional Resources

- https://angular.dev/guide/forms/template-driven-forms - Both basic and ngForms
- https://angular.dev/guide/forms/reactive-forms - Reactive Forms