Seneca
SCHOOL OF INFORMATION AND
COMMUNICATIONS TECHNOLOGY

BTI425

Web Programming for Apps and Services

Angular forms introduction

In recent weeks, we have had a straight-line topic treatment of components, routing, and services. However, we have only lightly covered the topic of *user interaction*, but now it’s time to do that better, now that we have a good foundation on which to build.

As the official Angular documentation states, “Forms are the mainstay of business applications. You use forms to log in, submit a help request, place an order, book a flight, schedule a meeting, and perform countless other data-entry tasks.”

Topic coverage plan

- First, we describe the three ways to do forms in Angular. We will use only one, *Template-driven Forms*.
- Next, we refresh our memory by showing a simple and standard HTML Form.
- Then, we show how to configure an Angular app to use HTML forms.
- Once the Angular app configured properly, we will add some data in a Component to be used with the form.
- Finally, we will show the changes required to the form as well as each of the standard

form elements to “bind” our data (using “two-way” binding) and “submit” the form.

Once this is compete, a brief summary of the highlights and dev tips are presented.

Three ways to do forms in Angular

Angular offers *three* ways to do forms:

1. Template-driven (covered in this course)
2. Reactive
3. Dynamic

In this course, we will work only with *Template-driven Forms*.

Template-driven Forms

This approach takes advantage of your knowledge of, and skills with an HTML template in a component.

It builds upon your experience with one-way read-only data binding (using curly braces syntax `{{ expression }}`, or its square-bracket form for element attributes, `[attr]='expression'`), by going further with two-way data binding.

Reactive Forms

This approach features more programming in the component class, where each element of the form is explicitly declared, configured, and managed. This is done mostly in the (TypeScript) component class source code file.

We will NOT work with Reactive Forms in this course. After understanding and working with Template-driven Forms, you will be able to learn what you need to, if or when you need to, work with Reactive Forms.

Dynamic Forms

This approach is interesting, in that metadata on the data model is used to generate forms dynamically. This replaces the cycle of editing in Template-driven Forms, where we go back-and-forth between the component’s class code and its HTML template, when developing a form.

As above, we will NOT work with Dynamic Forms in this course. After understanding and working with Template-driven Forms, you will be able to learn what you need to, if or when you need to, work with Dynamic Forms.

Starting point - “standard” HTML Form, *without* Angular

Here’s a simple form, in pure HTML5, which features all of the most typical form elements, ie:

- input (type: “text”, “checkbox”, “radio”)
- textarea
- select (single / “multiple”)

It also uses the Bootstrap “forms” classes, ie “form-group” and “form-control” for formatting:

```
<form action="/path/to/handler" method="post">

  <div class="form-group">
    <label class="control-label" for="name">Full Name:</label>
    <input type="text" class="form-control" id="name" name="name" req
  </div>

  <div class="form-group">
    <label class="control-label" for="description">Description:</labe
    <textarea class="form-control" id="description" name="description
  </div>

  <div class="form-group">
    <label class="control-label" for="ownedTransportation">Owned Tran
    <select multiple class="form-control" id="ownedTransportation" na
      <option value="C">Car</option>
      <option value="B">Bus</option>
      <option value="M">Motorcycle</option>
      <option value="H">Helicopter</option>
    </select>
  </div>

  <div class="form-group">
    <label class="control-label" for="favouriteTransportation">Favour
    <select class="form-control" id="favouriteTransportation" name="f
      <option value="C">Car</option>
      <option value="B">Bus</option>
      <option value="M">Motorcycle</option>
      <option value="H">Helicopter</option>
    </select>
  </div>

  <div class="form-group">
    <label for="" class="control-label">Has a driver's license?</labe
    <div class="checkbox">
      <label class="control-label" for="driverLicence">
        <input type="checkbox" id="driverLicence" name="driverLicence
    </div>
  </div>

  <div class="form-group">
    <label for="" class="control-label">Vehicle usage:</label>
    <div class="radio">
      <label class="control-label" for="vehicleUseBusiness">
        <input type="radio" id="vehicleUseBusiness" name="vehicleUse"
      </label>
    </div>
    <div class="radio">
      <label class="control-label" for="vehicleUsePleasure">
        <input type="radio" id="vehicleUsePleasure" name="vehicleUse"
      </label>
    </div>
    <div class="radio">
      <label class="control-label" for="vehicleUseOther">
        <input type="radio" id="vehicleUseOther" name="vehicleUse" va
      </label>
    </div>
  </div>

  <button class="btn btn-primary" type="submit">Create</button>

</form>
```

It's possible that you have written hundreds of these forms. It's a very well-understood process.

> *Reference info:*
> *Review the docs about HTML forms and native form controls.*

Get the code example

As you work through this document, get the `forms-intro` code example from the repo.

Configuring an Angular app to use HTML forms

Before making any changes to the form, we must add the Angular forms-handling bits to the project. In the documentation's Revise *app.module.ts* section, we do a task with two related steps:

1. Import the FormsModule
2. Add FormsModule to the "imports" array

Configure a data model for the form

Next, we *always assume* that an Angular form is backed by a data model. For typical scenarios, the data model can be defined in the same source code file as the component class that interacts with the form. The data model's values are *made available to* the form when it is built and rendered, and *updated by* the form during user interaction and submission.

For example, consider the following component class. It contains all the data that is required to populate our "standard" HTML form, including some classes that define the "shape" of the data model, as well as some sample data that we can use to "bind" to our form:

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

class Driver{
    name: string;
    password: string;
    description: string;
    ownedTransportation: string[];
    favouriteTransportation: string;
    driverLicence: boolean;
    skillLevel: number;
    vehicleUse: string;
}

class Option{
  value: string;
  text: string;
}

@Component({
  selector: 'app-driver',
  templateUrl: './driver.component.html',
  styleUrls: ['./driver.component.css']
})
export class DriverComponent implements OnInit {

  constructor() { }

  // the data that will be used in the form
  driverData: Driver;

  // Define the preset list of "transportation" options
  transportationList: Option[] = [
    {value: "C", text: "Car"},
    {value: "B", text: "Bus"},
    {value: "M", text: "Motorcycle"},
    {value: "H", text: "Helicopter"}
  ];

  ngOnInit() {

    // Populate the "driverData" with some static data (this would no
    this.driverData = {
      name: "Richard Hammond",
      password: "mysecret!",
      description: "Richard is a motor vehicle enthusiast",
      ownedTransportation: ["C", "M"],
      favouriteTransportation: "M",
      driverLicence: true,
      skillLevel: 8,
      vehicleUse: "pleasure"
    };

  }
}
```

To briefly explain, we define a “Driver” class that will represent the type of data that we will be “binding” to our form so that it can be modified. We also define a generic “Option” class, which is simply defining what our “Options” will look like, ie `{value: "C", text: "Car"}` - this can be used as an “option” in an `<select>` list, or the value / label used in a radio button.

> *For a full discussion of data models for forms, study this:*
> *Angular Forms Data Models*

Data binding between the *model* and *form*, introduction

We have three ways to bind the data model to form elements/controls:

1. One way, from *model* to *form*
2. One way, from *form* to *model*
3. Two way

This topic is covered in more detail in the Angular docs.

1. One way, from *model* to *form*

Assume we have a property in the data model that must appear in the view. In the past, to display it *as content in an element*, you typically used *interpolation* syntax:
`{{ foo.barText }}`

However, to bind *to an attribute* of a form element/control, we must use the other square-brackets format. For example, set the element's `title` attribute text:

```
<input [title]="foo.barText" ...
```

> *The value of the binding can be a data property, or an expression, or a call to a method in the component.*

2. One way, from *form* to *model*

Typically used to handle an event (click, change, focus, blur, etc.) that happens in a form element/control. Use the parenthesis format. The value of the binding is typically a method in the component. For example, to detect when an element/control has received focus:

```
<input (focus)="fooStartBar() ...
```

3. Two way

Typically used to bind the *value* of the form element/control with a property in the data model. For example, this will bind an input element/control with a property, so that changes to either will be bound and synchronized:

```
<input [(ngModel)]="foo.barText" ...
```

> *The value of the binding can be a data property, or an expression, or a call to a method in the component.*

More about two-way data binding

After coding the component, we can begin to update the original "standard" HTML form to work directly with the data, using "two-way binding" syntax:

> *Often, we want to both display a data property and update that property when the user makes changes.*
>
> *On the element side that takes a combination of setting a specific element property and listening for an element change event.*
>
> *Angular offers a special two-way data binding syntax for this purpose, `[(x)]`. The `[(x)]` syntax combines the brackets of property binding, `[x]`, with the parentheses of event binding, `(x)`.*
>
> *Reference info: Two-way binding [(…)]*

The target of the two-way binding is the **NgModel** directive. Every time we have a form element that we wish to "bind" to our data model, we use the syntax:

```
[(ngModel)]="componentProperty"
```

For example, let's see how we can update each of our form element types in our "Simple" form using this syntax, paired with the "DriverComponent" data:

input (type="text")

```
<input type="text" class="form-control" name="name" [(ngModel)]="driv
```

Here, we simply add the "two-way" binding syntax with ngModel to reference the "driverData.name" property.

input (type="password")

```
<input type="password" class="form-control" name="password" [(ngModel
```

Similar to above, but with a different input type.

textarea

```
<textarea class="form-control" name="description" [(ngModel)]="driver
```

This is very similar to the **input** example above, ie: we simply add the two-way data binding to ngModel with the correct Component property.

select / select multiple

```
<select multiple class="form-control" name="ownedTransportation" [(ng
        <option *ngFor = "let t of transportationList" [value]="t.val
</select>
```

```
<select class="form-control" name="favouriteTransportation" [(ngModel
          <option *ngFor = "let t of transportationList" [value]="t.v
</select>
```

The above two examples are practically identical, the only differences are the property that they’re binding to and the “multiple” attribute.

You will notice that our `[(ngModel)]` binding syntax has not changed, however the method for displaying the `<option>` elements is different. Here, we use the standard `*ngFor` structural directive, but we have added a **value** property that we can and must set.

Since both the “ownedTransportation” and “favouriteTransportation properties use the “value” of the transportation, we must use “transportation.value” as the “value” for the `<option>` elements, to correctly bind to the lists.

input (type=”checkbox”)

```
<input type="checkbox" name="driverLicence" [(ngModel)]="driverData.d
```

Once again, nothing special here. We simply bind to ngModel as before.

input (type=”range”)

```
<input type="range" min="1" max="10" class="form-control" name="skill
```

This one is interesting, because it renders a slider in the UI. We set the two-way binding to the `skillLevel` value, and also set the current rendered `value` to the same.

input (type=”radio”)

```
<div class="radio">
  <label class="control-label" for="vehicleUseBusiness">
    <input type="radio" name="vehicleUse" [(ngModel)]="driverData.veh
  </label>
</div>
<div class="radio">
  <label class="control-label" for="vehicleUsePleasure">
    <input type="radio" name="vehicleUse" [(ngModel)]="driverData.veh
  </label>
</div>
<div class="radio">
  <label class="control-label" for="vehicleUseOther">
    <input type="radio" name="vehicleUse" [(ngModel)]="driverData.veh
  </label>
</div>
```

Here, we must place identical ngModel binding on each “radio” button with the same “name” attribute.

As a rule of thumb, whenever you would like to “read from” / “write to” a form using **two-way** binding, always bind to ngModel on a form element that would typically have a “name” property.

“Live” or *real time* data model updates

Above, it was stated that we *always assume* that an Angular form is backed by a data model.

We write one or more classes in the component class source code file, which describe(s) the shape of the data on the form.

The data model appears as one or more objects - properties - in the component class.

The effect of two-way data binding is that data is *always* in sync, between the user interface (template) and the data model (code).

This realization is *very important*. Subtle, simple-sounding, but very important.

It means that we do not have to worry about losing data, moving data to-and-from the UI and code, and other similar data move/copy tasks. The binding just works, and we can rely on it.

It also means that we can perform tasks on the data "live" and in *real time*, as the user interacts with the form. We will do this in the near future.

From before, all of our data for "Richard Hammond" should be correctly rendered in the form. As a way to inspect/test that the two-way binding is working, you can add the following line somewhere below the form:

```
{{ driverData | json }}
```

This will show you how your driverData "data model" is being updated with every change you make in the form!

Handling the Form "Submission"

What about a "submit" task? Do we need one? Typically, we do, as it's a way for the user to indicate that they are done with the form.

To handle a form submission event, we add the event handler "ngSubmit" to our `<form>` element:

```
<form (ngSubmit)='onSubmit()' #f='ngForm'>
```

The above will enable the `<button type='submit'...` element to execute the method named "onSubmit". We will have to write that method.

Summary of the big ideas

While there are *many* new ideas, some new syntax, plenty of Angular parts (directives, classes, etc.), working with forms at a beginner level is not hard. Use the guidance at the top of this document to quickly build success.

Here are the important "big ideas":

Enable forms for the entire app by adding an import in the app module. It's a one-time task per app. Doing this enables Angular to do its magic whenever a form is declared and used.

Always plan on using a data model - often known or referred to as a *view model* - when using a form.

> *If you're thinking that we don't really need a view model when displaying a form for the first time, we would suggest that is wrong.*
> *In almost all situations, we want to push some data to the form, to make it a better user interaction experience.*
> *For example, items in a select list, or default or initial/starter values for some of the form elements.*

Write a function that will handle form submit. It can do anything you want it to do.

Two-way data-binding is a huge feature. Use it when appropriate. Remember it is still appropriate to use one-way read-only data binding to pull in values (curly braces). And, it is still appropriate to define event handlers on form elements (parentheses) if you need special behaviours.

Happy programming! Enjoy!