Template Driven Forms

In this lecture we'll be converting the model driven form we've been building so far in this section into a *template driven* form.

Learning Objectives

- Know the differences and similarities between template driven forms and model driven forms.
- How to use the `ngModel` directive to link template input controls to properties on the component.
- How to implement form validation in the template driven approach.
- How to submit and reset a form in the template driven approach.

Overview

The key in understanding the *template driven* approach is that it *still* uses the same models as the *model driven* approach. In the template driven approach Angular creates the models, the `FormGroups` and `FormControls`, for us via directives we add to the template.

That's why in this course we teach the model driven approach first. So you'll have a good knowledge of the underlying model structure that is still present in template driven forms.

Template Drive Forms are just Model Driven Form but *driven* by directives in the the template versus code in the component.
In template driven we use directives to create the model.
In model driven we create a model on the component and then use directives to map elements in the template to our form model.

Form setup

We create a basic form component, exactly the same as the model form component we started with in this section, with a basic form template, a dynamic select box and a simple component like so:

```
@Component({
    selector: 'template-form',
    templateUrl: `
<form novalidate>
  <fieldset>
    <div class="form-group">
      <label>First Name</label>
      <input type="text"
             class="form-control">
    </div>

    <div class="form-group">
      <label>Last Name</label>
      <input type="text"
             class="form-control">
    </div>
  </fieldset>

  <div class="form-group">
    <label>Email</label>
    <input type="email"
           class="form-control">
  </div>

  <div class="form-group">
    <label>Password</label>
    <input type="password"
           class="form-control">
  </div>

  <div class="form-group">
    <label>Language</label>
    <select class="form-control">
      <option value="">Please select a language</option>
      <option *ngFor="let lang of langs"
              [value]="lang">
              {{lang}}
      </option>
    </select>
  </div>
</form>
`
})
class TemplateFormComponent implements OnInit {

    langs:string[] = [
        'English',
        'French',
        'German',
    ];

    ngOnInit() {
    }
}
```

Remove all the `formGroup`, `formGroupName`, `formControl` and `formControlName` directives from our template, these are from the `ReactiveFormsModule` and used for model driven forms.

Directives

The directives we need to build template driven forms are in the `FormsModule` not the

`ReactiveFormsModule`, so lets import that and add it to our `NgModule` as an import and remove the `ReactiveFormsModule`.

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

One of the directives pulled in via the `FormsModule` is called `NgForm`.

This directive has a selector which matches the `<form>` tag.

So just by adding `FormsModule` to our `NgModule` imports our template form is already associated wth an instance of the `NgForm` directive.

This instance of `ngForm` is *hidden* but we can expose it with a local template reference variable attached to the form element, like so:

```
<form #f="ngForm"> ... </form>
```

Now we can use the variable `f` in our template and it will point to our instance of the `ngForm` directive.

One of the things the `ngForm` directive does is create a top level `FormGroup` and lets us call functions as if it was an instance of a `FormGroup`.

If you remember one of the properties on a `FormGroup` was value, this is an object representation of all the forms controls.

So just like model driven forms we can output that to screen simply with a pre tag and the json pipe, like so:

```
<pre>{{f.value | json}}</pre>
```

If you ran this code now all that would get printed out is `{}`. The forms value is an empty object, even if you started typing into the input fields the value would not update.

This is because the `ngForm` directive doesn't *automatically* detect all the controls that exist inside the `<form>` tag it's linked to. So although it's created a top level `FormGroup`, it's empty.

We need go through and *explicitly* register each template control with the `ngForm` directive. `ngForm` will then create a `FormControl` instance and map that to the `FormGroup`.

In model driven forms we added `formControlName` directives to map template form controls to existing model form controls on the component.

In template driven forms we need Angular to create the model form controls for us for each template form control. To do that we need to do two things to each template form control:

1. Add the `NgModel` directive

2. Add the `name` attribute.

The `NgModel` directive creates the `FormControl` instance to manage the template form control and the `name` attribute tells the `NgModel` directive what *key* to store that `FormControl` under in the parent `FormGroup`, like so:

```
<input name="foo" ngModel>
```

is equivalent to:

```
let model = new FormGroup({
  "foo": new FormControl()
});
```

After adding `ngModel` to our template email input control, like so:

```
<div class="form-group">
  <label>Email</label>
  <input type="email"
         class="form-control"
         name="email"
         ngModel>
</div>
```

We can also see that `f.value` now shows the value of the email field, like so:

```
{
  "email": "asim@codecraft.tv"
}
```

If we now added the `name` attribute and ngModel directive to all of our template form controls we would see `f.value` print out:

```
{
  "firstName": "",
  "lastName": "",
  "email": "",
  "password": "",
  "language": ""
}
```

This isn't exactly the same as before, in our model driven form we wanted to group `firstName` and `lastName` into a nested `FormGroup` called `name`.

We can do the same in template driven forms with the `ngModelGroup` directive, lets add that to the parent `fieldset` element of our `firstName` and `lastName` template form controls, like so:

```
<fieldset ngModelGroup="name"> ... </fieldset>
```

We can now see that the output of `f.value` matches what we have before

```
{
  "name": {
    "firstName": "",
    "lastName": ""
  },
  "email": "",
  "password": "",
  "language": ""
}
```

We have now mirrored the model we created in the model driven form using template driven forms, with `FormControls` and a nested `FormGroup`.

Two way data binding

Another feature of the `ngModel` directive is that it lets us setup *two way* data binding between a template form control and a variable on our component.

So when the user changes the value in the template form control the value of the variable on the component automatically updates and when we change the variable on the component the template form control automatically updates.

The syntax for using the `ngModel` directive in this way is a little bit different, let’s set this up for our email field. First we add a string property called `email` on our component so we have somewhere to store the email, like so:

```
class TemplateFormComponent implements OnInit {
    email: string; ①
    langs:string[] = [
        'English',
        'French',
        'German',
    ];

    ngOnInit() {
    }
}
```

① We add an `email` property so we can store the email the user enters on the component.

Then we setup *two way* data binding by changing our email `ngModel` directive to:

```
<input ... [(ngModel)]="email" >
```

The `[( )]` syntax is a combination of the syntax for input property binding `[]` and output event binding `()`

The long form of writing the above would be:

```
<input ... [ngModel]="email" (ngModelChange)="email = $event" >
```

But the `[()]` syntax is shorter and clearly shows we are implementing *two way* data binding on this input control.

Domain model

In Angular we typically won't data bind to a simple string or object on our component but a *domain model* we've defined via a class, lets create one for our Signup form called Signup.

```
class Signup {
  constructor(public firstName:string = '',
              public lastName:string = '',
              public email:string = '',
              public password:string = '',
              public language:string = '') {
  }
}
```

Then on our component we replace our `email` property with:

```
model: Signup = new Signup();
```

Now lets bind all our input controls to our model directly, like so:

```
<input ... [(ngModel)]="model.email" >
```

Validation

In the model driven approach we defined the validators via code in the component.

In the template driven approach we define the validators via `directives` and HTML5 attributes in our template itself, lets add validators to our form template.

All the fields apart from the language were *required*, so we'll just add the `required` attribute to those input fields, like so:

```
<input type="email"
           class="form-control"
           name="email"
           [(ngModel)]="model.email"
           required>
```

The email field also had a *pattern* validator, we can add that via an attribute as well, like so:

```
<input type="email"
     class="form-control"
     name="email"
     [(ngModel)]="model.email"
     required
     pattern="[^ @]*@[^ @]*">
```

The password field also had a *min length* validator, we can add that via an attribute also, like so:

```
<input type="password"
       class="form-control"
       name="password"
       [(ngModel)]="model.password"
       required
       minlength="8">
```

The `attributes` we are adding to add validation to our control are parts of the standard HTML5 specification. They are built-in to HTML5 and not part of Angular.

Validation styling

Similar to model driven forms we can access each model form controls state by going through the top level form group.

The `ngForm` directive makes the top level `FormGroup` available to us via the `.form` property, so we can show the *valid*, *dirty* or *touched* state of our email field like so:

```
<pre>Valid? {{f.form.controls.email?.valid}}</pre>
<pre>Dirty? {{f.form.controls.email?.dirty}}</pre>
<pre>Touched? {{f.form.controls.email?.touched}}</pre>
```

The ? is called the *elvis* operator, it means:

"Only try to call the property on the right of ? if the property on the left of ? is ***not*** *null"*

So if form.controls.email was null or undefined it would not try to call form.controls.email.valid (which would throw an error).

In template driven forms the controls can sometimes be null when Angular is building the page, so to be safe we use the *elvis* operator. We don't need to use this in model driven forms since the models are created already in our component by the time the HTML form is shown on the page.

So again similar to model driven forms we can use this in conjunction with the ngClass directive and the validation classes from twitter bootstrap to style our form to give visual feedback to the user when it's invalid.

Lets add validation styling to our email field, like so:

```
<div class="form-group"
     [ngClass]="{
      'has-danger': f.form.controls.email?.invalid && (f.form.controls.email?.dirty ||
f.form.controls.email?.touched),
      'has-success': f.form.controls.email?.valid && (f.form.controls.email?.dirty ||
f.form.controls.email?.touched)
 }">
  <label>Email</label>
  <input type="email"
         class="form-control"
         name="email"
         [(ngModel)]="model.email"
         required
         pattern="[^ @]*@[^ @]*">
</div>
```

The above code displays a red border round the input control when it's invalid and a green border when it's valid.

Writing shorter validation expressions

The NgForm directive does provide us with a shortcut to the controls property so we can write f.controls.email?.valid instead of f.form.controls.email?.valid.

But both are still pretty *wordy*, and if we wanted to get access to a nested form control like firstName it can become even more cumbersome, f.controls.name.firstName?.valid.

Using the ngModel directive however provides us with a *much* shorter alternative.

We can get access to the instance of our ngModel directive by using a *local template reference* *variable*, like so:

```
<input ... [(ngModel)]="model.email" #email="ngModel"> </input>
```

Then in our template we can use our local variable `email`.

Since `NgModel` created the `FormControl` instance to manage the template form control in the first place, it stored a *reference* to that `FormControl` in its `control` property which we can now access in the template like so `email.control.touched`. This is such a common use case that the `ngModel` directive provides us a shortcut to the `control` property, so we can just type `email.touched` instead.

We can then shorten our validation class expression and re-write the template for our email control like so:

```
<div class="form-group"
     [ngClass]="{
      'has-danger': email.invalid && (email.dirty || email.touched), ①
      'has-success': email.valid && (email.dirty || email.touched)
 }">
  <label>Email</label>
  <input type="email"
         class="form-control"
         name="email"
         [(ngModel)]="model.email"
         required
         pattern="[^ @]*@[^ @]*"
         #email="ngModel"> ②
</div>
```

① We can now access the form control directly through the template local variable called `email`.

② We create a template local variable pointing to the instance of the ngModel directive on this input control.

So now our template is a lot less *verbose*.

As long as we named our local reference variables the same name we named our form controls in the *model driven version of this form* we can just re-use the same `ngClass` syntax, like so:

```
<div class="form-group"
       [ngClass]="{
        'has-danger': email.invalid && (email.dirty || email.touched),
        'has-success': email.valid && (email.dirty || email.touched)
    }">
```

Validation messages

As for form validation messages, we can use *exactly the same method* that we used in model driven forms. As long as we named the local reference variables the same as the form controls in the model driven approach we can use exactly the same HTML in our template driven forms, like so:

```
<div class="form-control-feedback"
     *ngIf="email.errors && (email.dirty || email.touched)">
  <p *ngIf="email.errors.required">Email is required</p>
  <p *ngIf="email.errors.minlength">Email must contain at least the @ character</p>
</div>
```

Submitting the form

Submitting a form is exactly the same in model driven forms as it is in template driven forms.

We need a submit button, this is just button with a `type="submit"` somewhere between the opening and closing `form` tags.

```
<form>
  .
  .
  .
  <button type="submit" class="btn btn-primary" >Submit</button>
</form>
```

By default this would just try to post the form to the current URL in the address bar, to hijack this process and call a function on our component instead we use the `ngSubmit` directive (which comes from the `FormsModule`).

```
<form (ngSubmit)="onSubmit()">...</form>
```

This is an output event binding which calls a function on our component called `onSubmit` when the user clicks the submit button.

However, we don't want the form submitted when the form is invalid. We can easily disable the submit button when the form is invalid, like so:

```
<button type="submit" class="btn btn-primary" [disabled]="f.invalid">Submit</button>
```

Resetting the form

In the model driven approach we reset the form by calling the function `reset()` on our `myform` model.

We need to do the same in our template driven approach but we don't have access to the underlying form model in our component. We only have access to it in our template via our local reference variable `f.form`

However, we can get a reference to the `ngForm` instance in our component code by using a `ViewChild` decorator which we covered in the section on components earlier on in this course.

This decorator gives us a reference in our component to something in our template.

First we create a property on our component to hold an instance of `NgForm`, like so:

```
form: any;
```

Then we import the `ViewChild` decorator from `@angular/core`, like so:

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

Finally we decorate our property with the `ViewChild` decorator. We pass to `ViewChild` the *name* of the local reference variable we want to link to, like so:

```
@ViewChild('f') form: any;
```

And then in our `onSubmit()` function we can just call `form.reset()` like we did in the model driven approach.

The full listing for our component is now:

```
class TemplateFormComponent {

  model: Signup = new Signup();
  @ViewChild('f') form: any;

  langs: string[] = [
    'English',
    'French',
    'German',
  ];

  onSubmit() {
    if (this.form.valid) {
      console.log("Form Submitted!");
      this.form.reset();
    }
  }
}
```

Now when we submit the form it *blanks out* all the fields and also resets the states of the form controls so any validation styling and errors reset also to the original pristine condition.

Summary

In this lecture we converted our model driven form into a template driven form.

We learnt that the template driven form still uses the same classes as the model driven form but in the template drive approach the models are created by directives in the template instead of explicitly created on the component.

The `ngForm` directive automatically attaches to `<form>` and creates a top level `FormGroup`.

We learnt about the `?` elvis operator and how we can use it when some properties in a dot chain can be undefined or null.

We learnt how to use the ngModel directive, how it creates the FormControl instance for us and how we can use it to implement two way data binding to a domain model on our component.

We learnt how to use a domain model in Angular.

We learnt how to implement validation in template driven forms as well as how to submit and reset a form.

Listing

http://plnkr.co/edit/f4Dj1ZPVJHe6kcuF3Bk2?p=preview

script.ts

```
import {
    NgModule,
    Component,
    OnInit,
    ViewChild
} from '@angular/core';
import {
    FormsModule,
    FormGroup,
    FormControl
} from '@angular/forms';
import {BrowserModule} from '@angular/platform-browser';
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';

class Signup {
  constructor(public firstName: string = '',
              public lastName: string = '',
              public email: string = '',
              public password: string = '',
              public language: string = '') {
```