Seneca
SCHOOL OF INFORMATION AND COMMUNICATIONS TECHNOLOGY

BTI425

Web Programming for Apps and Services

Validation introduction for Angular template-driven forms

In this document, you are introduced to validation capabilities in Angular template-driven forms.

Template reference variables

Recently, you learned to configure the `<form>` element in a very specific way:

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

In an Angular (HTML) template, an attribute that begins with a hash symbol (# aka pound sign character) is a *template reference variable*. In the Angular docs, it states:

> *A **template reference variable** is often a reference to a DOM element within a template.*
> *It can also be a reference to an Angular component or directive or a web component.*
> *Use the hash symbol (#) to declare a reference variable.*
> *You can refer to a template reference variable anywhere in the template.*

What this means is that we can use `f` as a reference to the form. We'll see this later.

Why this matters

In the next section, you will learn how to get access to a form element's (i.e. a control) state or condition, which is the first validation-related task.

Track form element/control state

The Angular docs cover this topic well. Here, we will provide you with actionable implementation tasks.

Add some CSS rules

Add these new CSS rules to the project's root `styles.css` source code file:

```
/* Classes that help with template-driven forms */

.ng-valid[required], .ng-valid.required  {
  border-left: 5px solid #42A948; /* green */
}

.ng-invalid:not(form)  {
  border-left: 5px solid #a94442; /* red */
}
```

Test them

In a form that has two or more text input elements, add the `required` attribute. Then run the app, and notice that the left side of the input element has a red-coloured border when the element's value is empty/null, and a green-coloured border when not empty. Nice.

Add validation attributes to elements

The Angular docs cover this topic well. Here, we will provide you with actionable implementation tasks.

> *Don't ignore its suggestion to follow the link to the MDN document on native HTML form validation.*
> *It's a good review, and we will use some of its guidance in our forms.*

Think about what validation you want or need

This is an important task. It must be done on a per-element basis.

For example, in this week's code example, in the "add new" form, there are two text input elements, for "name" and "job". The "name" element is `required`. Here's what it looks like:

```
<input class="form-control" id="name" name="name" [(ngModel)]="newUse
```

To add validation, follow this recipe:

1. Decide what to validate
2. Add the appropriate attribute(s)
3. Add a *template reference variable*

Let's talk about #1 and #2: We already have the `required` attribute. For character length/size, let's also add minimum and maximum attributes (`minlength` and `maxlength`).

For #3, we need access to the element later on in the template. The guidance or suggestion is that the name of the template reference variable can match the value that you used for the `id` and/or `name` attributes above (i.e. `name`).

Here's what it looks like after the changes:

```
<input class="form-control" id="name" name="name"
  [(ngModel)]="newUser.name" required minlength="3"
  maxlength="100" #name="ngModel" autofocus>
```

Now what? Let's add some UI to show an error, as you saw in the Angular docs:

Add UI to show validation errors

Just below the `<input...` element, add a new container to show error messages. Here's an example, and we'll explain it below:

```
<div class="form-group">
  <label class="control-label" for="name">Name:</label>
  <input class="form-control" id="name" name="name" [(ngModel)]="newU
    maxlength="100" #name="ngModel" autofocus>

  <!-- Validation error area -->
  <div *ngIf='name.invalid && (name.dirty || name.touched)' class='al
    <!-- The code below is situation-dependent - think before coding
    <div *ngIf='name.errors?.required'>Name is required, 3 to 100 cha
    <div *ngIf='name.errors?.minlength'>Name must be at least 3 chara
  </div>
</div>
```

Notice that the validation error(s) wrapper `div` will appear only when there's a problem. The first `ngIf` takes care of that.

Notice the use of the `name` *template reference variable*. This is how we get access to that element from another element.

Notice also that the `name` element has properties that hold useful information, like `invalid` (i.e. true or false) and `touched`. We also have access to an `errors` property, which we'll use next.

Inside the validation error(s) wrapper `div`, one or more nested containers hold a very specific message, depending on the error.

Notice the first nested container:

```
<div *ngIf='name.errors?.required'>Name is required, 3 to 100 charact
```

> *Notice that the `errors` property is declared as optional with a question mark. This is recommended.*

We interpret this as follows: If the "name" element's errors property has a "required" property value of "true", then show some error message text.

Other validation using `pattern` and a regex

Above, you were reminded about some of the standard HTML form control validation attributes, and learned how a few of them (but not all) can be used in Angular template-driven forms.

A common task, in a form, is to validate a URL or an email address. Can that be done? Yes, using the standard `pattern` attribute.

For example, assume that you want the form user to enter a URL. Provide an input field, and a regex as suggested below (some detail has been omitted):

```
<input id="website" name="website" [(ngModel)]="website" pattern="(ht
```

The regex provided here ensures that the input data must include the scheme prefix, and that it matches most common URLs.

If you add a template reference variable, then you can notify the form user of problems via the (for example) `website.errors?.pattern` property.

Ensure that the whole/entire form is valid

One more task must be done, and it's simple. We ensure that the whole/entire form is valid before it can be submitted.

That means disabling the "save" (or whatever) button until the form is valid. Here's how to do it: Add a `disabled` attribute to the submit button:

```
<button class="btn btn-primary" type="submit" [disabled]="!f.form.val
```

Ah, here we are using the `f` *template reference variable*, and checking whether it's valid. Remember (above) we configured our `form` element with a *template reference variable*?

How and why does this work?

The `form` element is a container for many other elements, including `input` elements and this `button` element.

Angular is managing the form. The validity of a contained element (true/false) bubbles up to provide validity info for the entire form. After ALL contained elements are valid, the form's valid state changes to true. That's how the button is enabled/disabled.

What's next?

In the near future, we may learn some additional form and data validation techniques beyond this simple text input element coverage (e.g. item selection, etc.).