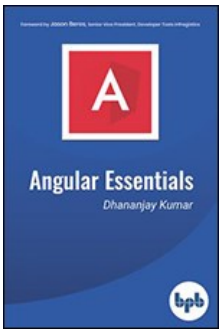

Angular Essentials: The Essential Guide to Learn Angular

by Dhananjay Kumar

Chapter 7: Template Driven Forms

In this chapter, you will learn about Forms in Angular. Following topics will be covered in this chapter:

- Template-Driven Forms
- ngModel, [ngModel],[(ngModel)]
- Binding form to ngForm
- Submitting the form
- Handling validation
- Resetting the form

Template- Driven Forms

You create forms to accept user inputs. A form is created for various purposes, such as:

- To login a user
- To sign up a user
- To submit a help request
- To place an order
- To schedule a meeting

A form in combination of other HTML controls, a form may contains,

1. Input elements
2. Check boxes
3. Radio buttons
4. Buttons

To work with user inputs, Angular provides us different types of forms. They are as follows:

- Template Driven Form
- Reactive Form

Purpose of both these forms is same, with few basic differences.

In Template-driven forms, you create all validation logics, form controls in the template. To create a template-driven form, you almost do not need any code in the template class. Template-driven forms are normal forms on which you use Angular directives to enable Angular features such as two-way data binding, change notification, validations, and so on.

Let us create a Login Form as template-driven form. First step required is to import **`FormsModule`** in the **`AppModule`** as shown in the *code listing 7.1.* I am importing **`FormsModule`** in **`AppModule`** and on purpose showing you the entire code of **`AppModule`.**

Code Listing 7.1

```
import { BrowserModule } from '@angular/platform-browser';
import  { NgModule  }  from '@angular/core';
import  {FormsModule}  from '@angular/forms';

import  { AppComponent  }  from './app.component';
@NgModule({
   declarations:   [
      AppComponent
```

```
    ],
    imports:    [
        BrowserModule,  FormsModule
    ],
    providers:    [],
    bootstrap:    [AppComponent]
})
export class AppModule {    }
```

In the **`AppModule`**, we have imported **`FormModule`** and passed it in the imports array.

We are going to create a template-driven form for Login operation. To model login operation, let us create a model class, which will hold information about login operation. The login model class is a plain class with no behavior (methods) and contains only properties.

Code Listing 7.2

```
export class Login {
constructor(
    public email:  string,
    public password:  string,
    public rememberpassword?: boolean
    )    {     }
}
```

Login class is anemic model that has only properties but not the behavior. There are two required properties and one optional property in the **Login** class.

Next, let us create a vanilla HTML form to accept user input for login functionality. You can create a form as shown in the *code listing 7.3.* I have used bootstrap classes to make form little bit more immersive.

Code Listing 7.3

```
<div class="container">
    <h1>Login Form</h1>
    <form>
        <div class="form-group">
         <label for="email">Email</label>
        <input   type="text" class="form-control" id="email"
required>
      </div>
      <div class="form-group">
         <label for="password">Password</label>
              <input   type="password"   class="form-control"
id="password" required>
      </div>
      <div class="form-check">
         <input type="checkbox"  class="form-check-input"
id="rpassword">
               <label       class="form-check-label"
for="rpassword">remember password</label>
      </div>
   <button type="submit" class = "btn btn-success">Login</
button>
   </form>
</div>
```

You will get a HTML form rendered as shown in the *figure 7.1*.

Figure 7.1

We have set up the form, next we need to convert above HTML form to Angular template-driven form. For that, very first create model object in thecomponentclass.

We already have a **`Login`** model class; now let us create an object of **`Login`** class,which we will bind to the form. The object of Login class will act as model of the form. You can create Login model object instance as shown in *code listing 7.4.*

Code Listing 7.4

```
model: Login;
       constructor()   {
       this.model = new Login('a@abc.com',   'password123',
true );
     }
```

We have model object in place, now bind it to the input elements of the form to enable them as Angular template-driven form elements. We can bind email property of model object to email input at the form as shown in the *code listing 7.5.*

Code Listing 7.5

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

There are two important properties set:

1. name
2. ngModel

To work with template-driven form, you must set name property of input element. The name attribute is used as the key in **`FormGroup`** array for a particular element.

`ngModel, [ngModel],[(ngModel))`

ngModel is used to bind input element with model object. There are three options to bind input element with model object.

`ngModel`

You can use **`ngModel`** as shown in *code listing 7.6.* In this case, **`ngModel`** will use name attribute to create key in **`ngForm`** object. In **`ngModel`**, input element is not set with any initial value.

Code Listing 7.6

```
      <input    type="text"
            ngModel
            name='email'
            class="form-control" required>
[ngModel]
```

You can use **`[ngModel]`** to create one-way binding. When **`ngModel`** is set as property binding, it will set initial value of input element with model object.

Code Listing 7.7

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

In the form, initial value of email input element will be set to model object's email property.

`[(ngModel)]`

You can use **`[(ngModel)]`** to create two-way data binding in between model object and input element. This option will set initial value of element and whenever element is updated, it will update model object also.

Code Listing 7.8

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

I would recommend using **`ngModel`** as it will set initial value of input element and will not update model object.

Binding Form to `ngForm`

Next, we have to bind the HTML form with **`ngForm`** directive such that it would work as Angular template-driven form.

Angular **ngForm** directive enables form elements with additional features. It mainly performs following tasks on form elements, which has **`ngModel`** and name attribute set:

- It monitors properties of form elements
- It monitors validity of form elements
- It raises notification if value changes of form elements

`ngForm` also has **`valid`** property which is set to **true,** on if all the form elements validity is true.

You can bind form to **`ngForm`** directive using template reference variable, as shown in *code listing 7.9.*

Code Listing 7.9

```
   <form novalidate #loginform='ngForm'>
   </form>
```

By using **`ngForm`** directive, **`ngModel`** property binding, and name attribute a HTML Form is converted to Angular template-driven form.

Putting everything together, template-driven login form should look like *code listing 7.10.*

Code Listing 7.10

```
<div class="container">
    <h1>Login Form</h1>
    <form novalidate #loginform='ngForm'>
      <div class="form-group">
           <label for="email">Email</label>
        <input   type="text"    [(ngModel)]='model.email'
name='email'  class="form-control" required>
   </div>
   <div class="form-group">
       <label for="password">Password</label>
         <input   type="password"    [(ngModel)]='model.
password' class="form-control" name="password" required
minlength="4">
   </div>
   <div class="form-check">
      <input type="checkbox"  class="from-check-input"
[(ngModel )]='model.rememberpassword'  name="rpassword">
              <label      class="form-check-label"
for="rpassword">remember password</label>
       </div>
   <button type="submit" Class="btn btn-success">Login</
button>
  </form>
</div>
```

We have used **`ngForm, ngModel`** on DOM as shown in the *figure 7.2*.

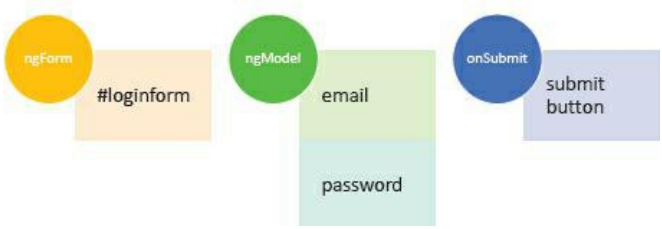

Figure 7.2

Submitting the Form

To submit template driven form, you need to use **`ngSubmit`** event on the form. You can bind **`ngSubmit`** event to a function in the component class. To submit form modify form as shown in *code listing 7.11.*

Code Listing 7.11

```
<form       novalidate      #loginform='ngForm'
(ngSubmit)='onSubmit(loginform)'>
  <!-- other elements    -->
 </form>
```

On the component class **`onSubmit`** function will look like *code listing 7.12.*

Code Listing 7.12

```
onSubmit(loginform)   {
     console.log(loginform.value);
     console.log(loginform.status);
   }
```

Template-driven form has value and status properties. The value is an object, which contains all form elements as properties and status is Boolean, which will be true if form is valid.

You can also disable **`Submit`** button, if form is invalid by setting [disabled] property of button to **`form.invalid.`**

Putting everything together **`template-driven`** Login form will be as shown in *code listing 7.13.*

Code Listing 7.13

```
<form      novalidate      #loginform='ngForm'
  (ngSubmit)='onSubmit(loginform)'>
   <div class="form-group">
      <label for="email">Email</label>
         <input   type="text"    [(ngModel)]='model.email'
name='email'  class="form-control" required> </div>
   <div class="form-group">
      <label for="password">Password</label>
         <input   type="password"    [(ngModel)]='model.
password' class="form-control" name="password" required
minlength="4">
   </div>
   <div class="form-check">
      <input type="checkbox"  class="form-check-input"
[(ngModel)]='model.rememberpassword'  name="rpassword">
               <label      class="form-check-label"
for="rpassword">remember password</label>
   </div>
   <button [disabled]='loginform.invalid' type="submit"
class="btn btn-success">Login</button>
   </form>
```

Component class with Login model and **`onSubmit`** function will be as shown in *code listing 7.14.*

Code Listing 7.14

```
import  {  Component  }  from '@angular/core';
import  {  Login }  from './login';

@Component({
   selector:   'app-root',
   templateUrl:   './app.component.html',
   styleUrls:   ['./app.component.css']
})
export class AppComponent {
   model:  Login;
   constructor()   {
   this.model = new Login('a@abc.com',   'password123',
true );
   }
   onSubmit(loginform)   {
     console.log(loginform.value);
      console.log(loginform.status);
   }
}
```

Handling Validation in Template Driven Form

You need to validate form controls, and show error messages on validation. Angular template-driven form tracks state of a control and updates CSS class associated with it. When you use **`ngModel`** in the form, it performs following tasks:

- Two- way data binding
- Find whether user touched the control
- Track state of the controls
- Update validation CSS associated with the controls

Angular associates various CSS classes to controls basis on the state as shown in *figure 7.3*.

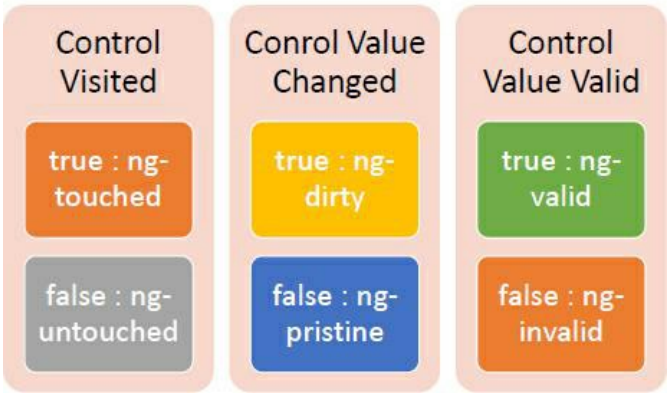

Figure 7.3

Toshow error messages using these classes, let us add some custom CSS to component. You can add CSS of *code listing 7.15* either in global CSS file or component's style.

Code Listing7.15

```
.ng-valid[required],   .ng-valid.required    {
     border-left:  5px solid #42A948;  /* green */
  }
  .ng-invalid:not(form)     {
     border-left:  5px solid #a94442;  /* red */
  }
```

Using combination of Angular template-driven form validations and CSS classes, you can show the validation error message on form as shown in the *code listing 7.16.*

Code Listing 7.16

```
<div class="form-group">
      <label for="email">Email</label>
         <input type="text" [(ngModel)]='model.email'
#email="ngModel" name='email' class="form-control"
required>
   </div>
      <div    [hidden]="email.valid    ||    email.pristine"
class="alert alert-danger">
      Email is required
   </div>
```

We are assigning **ngModel** to temp reference variable to track the changes. Also, using Bootstrap classes to make error message **div** more immersive, error message **div** displays if form control is invalid and touched as shown in the *figure 7.4*.

Figure 7.4

Reset the Form

You may have the requirement to reset the form and bind form with new model object. Angular template-driven form provides API for this as well. You canuse **reset()** method **ngForm** to reset the form as shown in *code listing 7.17.*

Code Listing 7.17

```
<button      (click)='newLogin() ;loginform.reset()'
class="btn btn-warning">Reset</button>
```

On the click event of the button, you are calling two functions:

- newLogin function
- reset method of the form

Before resetting the form, in **newLogin** function, you create new model object as shown the *code listing 7.18.* I have put null for all three properties; youcanchooseother valuesdependingonyour requirement.

Code Listing 7.18

```
newLogin()   {
      this.model = new Login(null,  null,  null);
   }
```

Putting Everything Together

Putting everything together, you can create **template-driven** login form in Angular with validations, model and track changes as shown in *code listing 7.19.*

Code Listing 7.19

```
      <form   novalidate   #loginform='ngForm'
(ngSubmit)='onSubmit(loginform)'>
   <div class="form-group">
      <label for="email">Email</label>
         <input   type="text"    [(ngModel)]='model.email'
```

```
#email="ngModel"      name='email'      class="form-control"
required>
   </div>
      <div    [hidden]="email.valid    ||    email.pristine"
class="alert alert-danger">
        Email is required
      </div>
      <div class="form-group">
         <label for="password">Password</label>
            <input   type="password"   #password="ngModel"
[(ngModel)]='model.password'   class="form-control"
name="password"
      required minlength="4">
   </div>
   <div [hidden]="password.valid || password.pristine"
class="alert alert-danger">
      Password is invalid
   </div>
   <div class="form-check">
      <input type="checkbox"  class="form-check-input"
[(ngModel)]='model.rememberpassword'  name="rpassword">
            <label       class="form-check-label"
for="rpassword">remember password</label>
   </div>
   <button [disabled]='loginform.invalid' type="submit"
class="btn btn-success">Login</button>
         <button    (click)='newLogin();loginform.reset()'
class="btn btn-warning">Reset</button>
   </form>
```

In the component class, you can have model object and function to handle submit as shown in *code listing 7.20*.

Code Listing 7.20

```
import  {  Component  }  from '@angular/core';
import  {  Login }  from './login';

@Component({
    selector:   'app-root',
    templateUrl:   './app.component.html',
    styleUrls:   ['./app.component.css']
})
export class AppComponent {
   model:  Login;
   constructor()   {
   this.model = new Login('a@abc.com', 'password123',
true );
     }
    onSubmit(loginform)   {
    console.log(loginform.value);
    console.log(loginform.status);
    }
    newLogin()   {
       this.model = new Login(null,  null,  null);
   }
}
```

Summary

In this chapter you learnt about template-driven form, validations etc. You create forms to accept user input. In this chapter you learnt about following topics:

- Template-Driven Forms
- `ngModel, [ngModel ],[(ngModel)]`
- Binding form to ngForm
- Submitting the form
- Handling validation

- Resetting the form