**What is Template-driven form?**

In Template Driven Forms we specify behaviors/validations using directives and attributes in our template and let it work behind the scenes. All things happen in Templates hence very little code is required in the component class. This is different from the reactive forms, where we define the logic and controls in the component class.

The Template-driven forms

1. The form is set up using ngForm directive
2. controls are set up using the ngModel directive
3. ngModel also provides the two-way data binding
4. The Validations are configured in the template via directives

Template-driven forms are

1. Contains little code in the component class
2. Easier to set up

While they are

1. Difficult to add controls dynamically
2. Unit testing is a challenge

**Create the Example Application**

Use ng new to create a new application

|  |  |
| --- | --- |
| 1  2  3 | ng new tdf  --routing=true --style=css |

Run ng serve and verify if everything is installed correctly.

**Import FormsModule**

To work with Template-driven forms, we must import the FormsModule. We usually import it in root module or in a [shared module](https://www.tektutorialshub.com/angular/angular-folder-structure-best-practices/#shared-module). The FormsModule contains all the form directives and constructs for working with forms

Open the app.module.ts and add the import { FormsModule } from '@angular/forms'; to it.

And also add the FormsModule to the *imports metadata property array*

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';  import { FormsModule } from '@angular/forms';        //import FormsModule    import { AppRoutingModule } from './app-routing.module';  import { AppComponent } from './app.component';    @NgModule({    declarations: [      AppComponent    ],    imports: [      BrowserModule,      AppRoutingModule,      FormsModule                    //Add in Imports Array    ],    providers: [],    bootstrap: [AppComponent]  })  export class AppModule { } |

**HTML Form**

The first task is to build the template. The following is a regular HTML form. We enclose it in a <form> tag. We have included two text input (FirstName & LastName), a email (email), a radio button (gender), a checkbox (isMarried), and a select list (country). These are form elements.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | <form>      <p>      <label for="firstname">First Name</label>      <input type="text" id="firstname" name="firstname">    </p>      <p>      <label for="lastname">Last Name</label>      <input type="text" id="lastname" name="lastname">    </p>      <p>      <label for="email">Email </label>      <input type="text" id="email" name="email">    </p>      <p>      <label for="gender">Geneder</label>      <input type="radio" value="male" id="gender" name="gender"> Male      <input type="radio" value="female" id="gender" name="gender"> Female    </p>      <p>      <label for="isMarried">Married</label>      <input type="checkbox" id="isMarried" name="isMarried">    </p>      <p>    <label for="country">country </label>    <select name="country" id="country">      <option selected="" value=""></option>      <option [ngValue]="c.id" \*ngFor="let c of countryList">        {{c.name}}      </option>    </select>    </p>      <p>      <button type="submit">Submit</button>    </p>    </form> |

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

**Component Class**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | import { Component } from '@angular/core';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent {    title = 'Template driven forms';      countryList:country[] = [      new country("1", "India"),      new country('2', 'USA'),      new country('3', 'England')    ];  }    export class country {    id:string;    name:string;      constructor(id:string, name:string) {      this.id=id;      this.name=name;    }  } |

**ngForm**

Once, we have a form with few form elements, the angular automatically converts it into a Template-driven form. This is done by the ngForm directive.

The ngForm directive is what makes the Angular template-driven forms work. But we do not need to add it explicitly. Angular adds it automatically

When we include FormsModule, the Angular is going to look out for any <form> tag in our HTML template. Angular does this via ngForm [directive](https://www.tektutorialshub.com/angular/angular-directives/). ngForm directive automatically detects the <form> tag and automatically binds to it. You do not have to do anything on your part to invoke and bind the ngForm directive.

The ngForm does the following

1. Binds itself to the <Form> directive
2. Creates a top-level FormGroup instance
3. CreatesFormControl instance for each of child control, which has ngModel directive.
4. CreatesFormGroup instance for each of the  NgModelGroup directive.

We can export the ngForm instance into a local template variable using ngForm as the key (ex: #contactForm="ngForm"). This allows us to access the many properties and methods of ngForm using the template variable contactForm

Hence, update the form element as shown below.

|  |  |
| --- | --- |
| 1  2  3 | <form #contactForm="ngForm"> |

**FormControl**

The FormControl is the basic building block of the [Angular Forms](https://www.tektutorialshub.com/angular/angular-forms-fundamentals/). It represents a single input field in an [Angular form](https://www.tektutorialshub.com/angular/angular-forms-fundamentals/)**.** The [Angular Forms Module](https://www.tektutorialshub.com/angular/angular-forms-fundamentals/#angular-forms-module) binds the input element to a FormControl. We use the FormControl instance to track the value, user interaction and validation status of an individual form element. Each individual Form element is a FormControl

We have six form elements in our HTML template. They are firstName, lastname, email, gender, isMarried & country. We need to bind them to FormControl instance. We do this by using the ngModel directive. Add the ngModel directive to each control as shown below.

|  |  |
| --- | --- |
| 1 | <input type="text" name="firstname" ngModel> |

ngModel will use the name attribute to create the FormControl instance for each of the Form field it is attached.

**Submit Form**

Now have the template ready, except for the final piece i.e submitting data to the component.

We use the ngSubmit event, to submit the form data to the component class. We use the [event binding](https://www.tektutorialshub.com/angular/angular-data-binding/) (parentheses) to bind ngSubmit to OnSubmit method in the component class. When the user clicks on the submit button, the ngSubmit event will fire

|  |  |
| --- | --- |
| 3 | **<form #contactForm="ngForm" (ngSubmit)="onSubmit(contactForm)">** |

We are passing the local template variable contactForm in onSubmit method. contactForm holds the reference to the ngForm directive. We can use this in our component class to extract the data from the form fields.

**Final Template**

Our final template is as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | <form #contactForm="ngForm" (ngSubmit)="onSubmit(contactForm)">      <p>      <label for="firstname">First Name</label>      <input type="text" name="firstname" ngModel>    </p>      <p>      <label for="lastname">Last Name</label>      <input type="text" name="lastname" ngModel>    </p>      <p>      <label for="email">Email </label>      <input type="text" id="email" name="email" ngModel>    </p>      <p>      <label for="gender">Geneder</label>      <input type="radio" value="male" name="gender" ngModel> Male      <input type="radio" value="female" name="gender" ngModel> Female    </p>      <p>      <label for="isMarried">Married</label>      <input type="checkbox" name="isMarried" ngModel>    </p>      <select name="country" ngModel>      <option [ngValue]="c.id" \*ngFor="let c of countryList">        {{c.name}}      </option>    </select>      <p>      <button type="submit">Submit</button>    </p>    </form> |

**Receive Form Data**

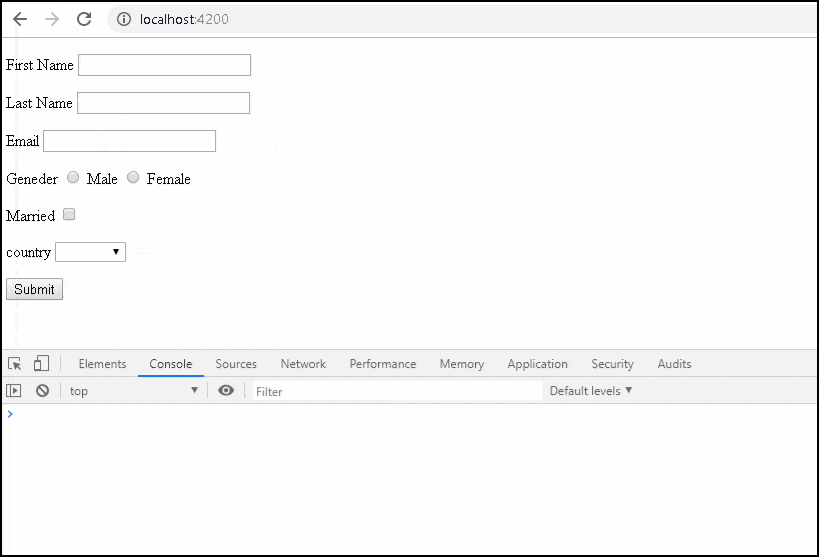
We need to receive the data in component class from our form. To do this we need to create the onSubmit method in our component class. The submit method receives the reference to the ngForm directive, which we named is as contactForm. The contactForm exposes the value method which returns the form fields as a Json object.

|  |  |
| --- | --- |
|  | onSubmit(contactForm) {      console.log(contactForm.value);    } |

You can print the value to the console using the console.log(contactForm.value)

Run the code now and enter some data into the form. Open the Developer Console in your browser and check the output, when you submit the data.

|  |  |
| --- | --- |
|  | country: "1"  firstname: "Sachin"  email:"sachin@gmail.com"  gender: "male"  isMarried: true  lastname: "Tendulkar" |

Angular template-driven forms in Action

**Local Variable**

We can assign the ngForm,FormControl or FormGroup instance to a template local variable. This allows us to check the status of the form like whether the form is valid, submitted, and value of the form elements, etc

**ngForm**

We have access to the ngForm instance via the local template variable #contactForm.

|  |  |
| --- | --- |
| 1  2  3 | <form #contactForm="ngForm" (ngSubmit)="onSubmit(contactForm)"> |

Now, we can make use of some of the properties & methods to know the status of form. For Example

|  |  |
| --- | --- |
|  | <p>    <button type="submit">Submit</button>  </p>    <pre>Value : {{contactForm.value | json }} </pre>  <pre>Valid : {{contactForm.valid}} </pre>  <pre>Touched : {{contactForm.touched  }} </pre>  <pre>Submitted : {{contactForm.submitted  }} </pre> |

value: The value property returns the object containing the value of every

**FormControl**  
valid: Returns true if the form is Valid else returns false.  
touched: True if the user has entered a value in at least in one field.  
submitted: Returns true if the form is submitted. else false.

**FormControl**

Similarly, we can also get access to the FormControl instance by assigning the ngModel to a local variable as shown below

|  |  |
| --- | --- |
|  | <input type="text" name="firstname" #fname="ngModel" ngModel> |

Now, the variable #fname holds the reference to the firstname FormControl. We can then access the properties of FormControl like value, valid, isvalid, tocuhed etc

|  |  |
| --- | --- |
|  | <p>    <label for="firstname">First Name </label>    <input type="text" name="firstname" #fname="ngModel" ngModel>  </p>    <pre>Value    : {{fname.value}} </pre>  <pre>valid    : {{fname.valid}} </pre>  <pre>invalid  : {{fname.invalid}} </pre>  <pre>touched  : {{fname.touched}} </pre |

value: Returns the current value of the control  
valid: Returns true if the value is Valid else false  
invalid: True if the value is invalid else false  
touched: Returns true if the value is entered in the element

**Nested FormGroup**

The FormGroup is a collection of FormControl. It can also contain other FormGroup's.

The ngForm directive creates the top Level FormGroup behind the scene, when we use the <Form> directive.

|  |  |
| --- | --- |
|  | <form #contactForm="ngForm" (ngSubmit)="onSubmit(contactForm)"> |

We can add new [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/) using the ngModelGroup directive. Let us add street, city & Pincode form controls and group them under the address FormGroup

All you need to do is to enclose the fields inside a div element with ngModelGroup directive applied on it as shown below

|  |  |
| --- | --- |
|  | <div ngModelGroup="address">        <p>        <label for="city">City</label>        <input type="text" name="city" ngModel>      </p>        <p>        <label for="street">Street</label>        <input type="text" name="street" ngModel>      </p>      <p>        <label for="pincode">Pin Code</label>        <input type="text" name="pincode" ngModel>      </p>    </div> |

Run the App and submit. The resultant object is as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | Value : {    "firstname": "Sachin",    "lastname": "Tendulkar",    "email":"sachin@gmail.com"    "gender": "male",    "isMarried": true,    "country": "1",    "address": {      "city": "Mumbai",      "street": "Fashin Street",      "pincode": "400600"    }  } |

**Setting the Initial Value**

The form is usually pre-filled with some default data. In the case of editing, we have to show the user the current data

**Validating the Form**

Validating the form is another important task. We have covered it in Validation in template-driven form tutorial.

**Summary**

**Angular Template-driven Forms** is simpler compared to the reactive forms. The FormsModule is imported first. Then we create the HTML form. The Angular detects the <form> tag and converts the form to the Angular Form. ngModel directive added to each form element, which converts them to FormControl. Finally, submit event is subscribed via event binding.

Validating and displaying error messages are equally important. We have covered it in a separate tutorial. The following is the list of tutorials on Angular forms

Chapter-2

## Template-driven Form Validation

Validations in [Template-driven forms](https://www.tektutorialshub.com/angular/angular-template-driven-forms/) are provided by the Validation directives. The [Angular Forms Module](https://www.tektutorialshub.com/angular/angular-forms-fundamentals/) comes with several built-in validators. You can also create your own custom Validator.

## Template

Consider the following template-driven form. It has firstname, lastname, email, gender & istoc form fields.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | <form #contactForm="ngForm" (ngSubmit)="onSubmit(contactForm)">      <p>      <label for="firstname">First Name </label>      <input type="text" id="firstname" name="firstname" [(ngModel)]="contact.firstname">    </p>      <p>      <label for="lastname">Last Name </label>      <input type="text" id="lastname" name="lastname" [(ngModel)]="contact.lastname">    </p>      <p>      <label for="email">email </label>      <input type="text" id="email" name="email" [(ngModel)]="contact.email">    </p>      <p>      <label for="gender">Geneder </label>      <input type="radio" value="male" id="gender" name="gender" [(ngModel)]="contact.gender"> Male      <input type="radio" value="female" id="gender" name="gender" [(ngModel)]="contact.gender"> Female    </p>      <p>      <label for="isToc">Accept TOC</label>      <input type="checkbox" id="isToc" name="isToc" [(ngModel)]="contact.isToc">    </p>      <p>      <button type="submit">Submit</button>    </p>    </form> |

## ****Component Class****

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | import { Component, ViewChild, ElementRef, OnInit } from '@angular/core';  import { NgForm } from '@angular/forms';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',    styleUrls: ['./app.component.css']  })  export class AppComponent implements OnInit  {    title = 'Template driven forms';      @ViewChild('contactForm',null) contactForm: NgForm;      contact:contact;      ngOnInit() {        this.contact = {        firstname:"",        lastname:"",        gender:"male",        isToc:true,        email:"",      };      }      onSubmit() {      console.log(this.contactForm.value);    }    }    export class contact {    firstname:string;    lastname:string;    gender:string;    isToc:boolean;    email:string;  } |

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

## Disabling the Browser validation

First, we need to disable browser validator interfering with the Angular validator. To do that we need to add novalidate attribute on <form> element as shown below

|  |  |
| --- | --- |
| 1  2  3 | <form #contactForm="ngForm" (ngSubmit)="onSubmit(contactForm)" novalidate> |

## Built-in Validators

The Built-in validators use the HTML5 validation attributes like required, minlength, maxlength & pattern. Angular interprets these validation attributes and add the validator functions to [FormControl](https://www.tektutorialshub.com/angular/formcontrol-in-angular/) instance.

## Adding in Built-in Validators

### Required Validation

The required validator returns true only if the form control has non-empty value entered. Let us add this validator to all fields

|  |  |
| --- | --- |
| 1  2  3 | <input type="text" id="firstname" name="firstname" required [(ngModel)]="contact.firstname"> |

### Minlength Validation

This Validator requires the control value must not have less number of characters than the value specified in the validator.

For Example, minlength validator ensures that the firstname value has at least 10 characters.

|  |  |
| --- | --- |
| 1  2  3 | <input type="text" id="firstname" name="firstname" required minlength="10" [(ngModel)]="contact.firstname"> |

### Maxlength Validation

This Validator requires that the number of characters must not exceed the value of the attribute.

|  |  |
| --- | --- |
| 1  2  3 | <input type="text" id="lastname" name="lastname" required maxlength="15" [(ngModel)]="contact.lastname"> |

### Pattern Validation

This Validator requires that the control value must match the regex pattern provided in the attribute. For example, the pattern ^[a-zA-Z]+$ ensures that the only letters are allowed (even spaces are not allowed). Let us apply this pattern to the lastName

|  |  |
| --- | --- |
| 1  2  3  4 | <input type="text" id="lastname" name="lastname" required maxlength="15"      pattern="^[a-zA-Z]+$" [(ngModel)]="contact.lastname"> |

### Email Validation

This Validator requires that the control value must be a valid email address. We apply this to the email field

|  |  |
| --- | --- |
| 1  2  3 | <input type="text" id="email" name="email" required email [(ngModel)]="contact.email"> |

## Disable Submit button

Now, we have successfully added the validators. You will notice that the click submit button still submits the form.

We need to disable the submit button if our form is not valid.

Angular forms module keep track of the state of our form and each of its form elements. These states are exposed to the user through FormGroup, FormArray & FormControl objects.

We get the reference to the top-level FormGroup instance by creating a template variable and bind it to ngForm. We have already done it when we had added the #contactForm="ngForm" in our form tag.

The [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/) has a valid property, which is set to true if all of its child controls are valid. We use it to set the disabled attribute of the submit button.

|  |  |
| --- | --- |
| 1  2  3 | <button type="submit" [disabled]="!contactForm.valid">Submit</button> |

So long as contactForm.valid remains false, the submit button remains disabled.

## Displaying the Validation/Error messages

We need to provide a short and meaningful error message to the user.

Angular creates a FormControl for each and every field, which has ngModel directive applied. The FormControl exposes the state of form element like valid, dirty, touched, etc.

There are two ways in which you can get the reference to the FormControl.

One way is to use the contactForm variable. We can use the contactForm.controls.firstname.valid to find out if the firstname is valid.

The other way is to create a new local variable for each FormControl For Example, the following firstname="ngModel" creates the firstname variable with the FormControl instance.

|  |  |
| --- | --- |
| 1  2  3  4 | <input type="text" id="firstname" name="firstname" required minlength="10"              #firstname="ngModel" [(ngModel)]="contact.firstname"> |

Now, we have a reference to the firstname FormControl instance, we can check its status. We use the valid property to check if the firstname has any errors.

valid: returns either invalid status or null which means a valid status

|  |  |
| --- | --- |
| 1  2  3  4  5 | <div \*ngIf="!firstname?.valid && (firstname?.dirty || firstname?.touched)">     Invalid First Name  </div> |

### Why check dirty and touched?

We do not want the application to display the error when the form is displayed for the first time. We want to display errors only after the user has attempted to change the value. The dirty & touched properties help us do that.

dirty: A control is dirty if the user has changed the value in the UI.  
touched: A control is touched if the user has triggered a blur event on it.

### Error message

The error message ” “Invalid First Name” ” is not helpful. The firstname has two validators. required and minlength

Any errors generated by the failing validation is updated in the errors object. The errors object returns the error object or null if there are no errors.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <div \*ngIf="!firstname?.valid && (firstname?.dirty || firstname?.touched)">    Invalid First Name    <div \*ngIf="firstname.errors.required">       First Name is required    </div>    <div \*ngIf="firstname.errors.minlength">      First Name Minimum Length is {{firstname.errors.minlength?.requiredLength}}    </div>  </div> |

Note that the minlength validators return the {{firstname.errors.minlength?.requiredLength}}, which we use the display the error message.

## Final Template

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81 | <form #contactForm="ngForm" (ngSubmit)="onSubmit(contactForm)" novalidate>      <p>      <label for="firstname">First Name </label>      <input type="text" id="firstname" name="firstname" required minlength="10" #firstname="ngModel"        [(ngModel)]="contact.firstname">    </p>    <div \*ngIf="!firstname?.valid && (firstname?.dirty || firstname?.touched)" class="error">      <div \*ngIf="firstname.errors.required">        First Name is required      </div>      <div \*ngIf="firstname.errors.minlength">        First Name Minimum Length is {{firstname.errors.minlength?.requiredLength}}      </div>    </div>      <p>      <label for="lastname">Last Name </label>      <input type="text" id="lastname" name="lastname" required maxlength="15" #lastname="ngModel"              pattern="^[a-zA-Z]+$"  [(ngModel)]="contact.lastname">    </p>    <div \*ngIf="!lastname?.valid && (lastname?.dirty || lastname?.touched)" class="error">      <div \*ngIf="lastname.errors.required">        Last Name is required      </div>      <div \*ngIf="lastname.errors.maxlength">        Last Name Minimum Length is {{lastname.errors.maxlength?.requiredLength}}      </div>      <div \*ngIf="lastname.errors.pattern">        Only characters are allowed      </div>    </div>          <p>      <label for="email">email </label>      <input type="text" id="email" name="email" required email #email="ngModel" [(ngModel)]="contact.email">    </p>    <div \*ngIf="!email?.valid && (email?.dirty || email?.touched)" class="error">      <div \*ngIf="email.errors.required">        Email is required      </div>      <div \*ngIf="email.errors.email">        Invalid Email Address      </div>    </div>      <p>      <label for="gender">Geneder </label>      <input type="radio" value="male" id="gender" name="gender" #gender="ngModel" required [(ngModel)]="contact.gender">      Male      <input type="radio" value="female" id="gender" name="gender" #gender="ngModel" required        [(ngModel)]="contact.gender"> Female    </p>    <div \*ngIf="!gender?.valid && (gender?.dirty || gender?.touched)" class="error">      <div \*ngIf="gender.errors.required">        Gender is required      </div>    </div>        <p>      <label for="isToc">Accept TOC</label>      <input type="checkbox" id="isToc" name="isToc" required #isToc="ngModel" [(ngModel)]="contact.isToc">    </p>    <div \*ngIf="!isToc?.valid && (isToc?.dirty || isToc?.touched)" class="error">      <div \*ngIf="isToc.errors.required">        Please accept the TOC      </div>    </div>      <p>      <button type="submit" [disabled]="!contactForm.valid">Submit</button>    </p>      <p>{{contactForm.valid}} </p>    </form> |

## Summary

Angular template-driven form validation uses the directives known as validators. The validators handle form validations and display validation messages. The Angular comes up with several built-in validators for this purpose. They are minlength, maxlength, email, pattern, required, etc.

Chapter-3

**Custom Validator in Template Driven Forms**

[Create a new Angular Project](https://www.tektutorialshub.com/angular/angular-create-first-application/). Copy the following code toapp.component.ts

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import { Component } from '@angular/core';    @Component({    selector: 'app-root',    templateUrl: './app.component.html',  })  export class AppComponent {      constructor() {    }    } |

Copy the following code app.component.html

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | <h1>Custom Validator in Template driven forms</h1>    <h2>Template Form</h2>    <form #myForm="ngForm" (ngSubmit)="onSubmit(myForm)" novalidate>      <label for="numVal">Number :</label>    <input type="text" name="numVal" ngModel #numVal="ngModel">      <p>Is Form Valid : {{myForm.valid}} </p>      <p>Form  : {{ myForm.value | json}} </p>    <p>      <button type="submit" [disabled]="!myForm.valid">Submit</button>    </p>    </form> |

It has only one input field numVal. Let us create a validator to ensure that the value of the numVal is greater than 10.

**Built-in Validators**

The [Angular Forms](https://www.tektutorialshub.com/angular/angular-forms-fundamentals/) Module already has a few built-in validators. They are listed below. But we do not have a greater than validator.

1. [Required validator](https://www.tektutorialshub.com/angular/template-driven-form-validation-in-angular/#required-validation)
2. [Min length Validator](https://www.tektutorialshub.com/angular/template-driven-form-validation-in-angular/#minlength-validation)
3. [Max length Validator](https://www.tektutorialshub.com/angular/template-driven-form-validation-in-angular/#maxlength-validation)
4. [Pattern Validator](https://www.tektutorialshub.com/angular/template-driven-form-validation-in-angular/#pattern-validation)
5. [Email Validator](https://www.tektutorialshub.com/angular/template-driven-form-validation-in-angular/#email-validation)

[**BEST ANGULAR BOOKS**](https://www.tektutorialshub.com/angular/angular-best-books/)  
**The Top 8**[**Best Angular Books**](https://www.tektutorialshub.com/angular/angular-best-books/)**, which helps you to get started with Angular**

**How to Build Custom Validator in template-driven form**

Building a Validator in [template-driven forms](https://www.tektutorialshub.com/angular/angular-template-driven-forms/) is similar to building an Angular directive. The directive must implement the Validator interface.

**Validator Interface**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | interface Validator {    validate(control: AbstractControl): ValidationErrors | null    registerOnValidatorChange(fn: () => void)?: void  } |

The directive must implement the validate function. Notice that the validate function has the same signature as the [ValidatorFn](https://angular.io/api/forms/ValidatorFn) Interface. Whenever the Validator directive is invoked angular looks for the validate method and invokes it.

**Validate Function**

A Validator is just a function, which must implement [ValidatorFn](https://angular.io/api/forms/ValidatorFn) Interface.

|  |  |
| --- | --- |
| 1  2  3  4  5 | interface ValidatorFn {    (control: AbstractControl): ValidationErrors | null  } |

The function takes the [AbstractControl](https://angular.io/api/forms/AbstractControl). This is the base class for [FormControl](https://www.tektutorialshub.com/angular/formcontrol-in-angular/), [FormGroup](https://www.tektutorialshub.com/angular/formgroup-in-angular/), and [FormArray](https://www.tektutorialshub.com/angular/angular-formarray-example-in-reactive-forms/). The validator function must return a list of errors i.e [ValidationErrors](https://angular.io/api/forms/ValidationErrors) or null if the validation has passed

**Custom Validator Example**

Create the gte.validator.ts and copy the following code.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | import { Validator, NG\_VALIDATORS, FormControl } from '@angular/forms'  import { Directive, OnInit, forwardRef } from '@angular/core';      @Directive({    selector: '[gteValidator]',    providers: [      { provide: NG\_VALIDATORS, useExisting: gteValidatorDirective, multi: true }    ]  })  export class gteValidatorDirective implements Validator, OnInit {      ngOnInit() {    }      validate(c: FormControl) {        let v: number = +c.value;        if (isNaN(v)) {        return { 'gte': true, 'requiredValue': 10 }      }        if (v <= +10) {        return { 'gte': true, 'requiredValue': 10 }      }        return null;    }  } |

We decorate the [directive](https://www.tektutorialshub.com/angular/angular-directives/) using @Directive decorator.

We use the directive as an attribute in the HTML template. The attribute needs a name or selector. We assign the name as gteValidator in the selector metadata section of the directive decorator.

|  |  |
| --- | --- |
| 1  2  3 | selector: '[gteValidator]', |

The Angular knows nothing about the Validation capabilities of our directive. Hence we need to register it in [Angular Providers](https://www.tektutorialshub.com/angular/angular-providers/) metadata using the special injection token [NG\_VALIDATORS](https://angular.io/api/forms/NG_VALIDATORS). We also set multi:true because there can be more validation directives.

|  |  |
| --- | --- |
| 1  2  3 | { provide: NG\_VALIDATORS, useExisting: gteValidatorDirective, multi: true } |

The directive class must implement the validate method. The validate method must honor the [ValidatorFn](https://angular.io/api/forms/ValidatorFn) Interface.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | validate(c: FormControl) {        let v: number = +c.value;        if (isNaN(v)) {        return { 'gte': true, 'requiredValue': 10 }      }        if (v <= +10) {        return { 'gte': true, 'requiredValue': 10 }      }        return null;    } |

It is a simple function, which checks if the value is a number and is less than 10. It returns null if it passes all checks.

If Validation fails it returns the ValidationErrors. It is a key-value pair object of type [key: string]: any and it defines the broken rule. The key is the string and should contain the name of the broken rule. The value can be anything, but usually set to true.

We return the following key-value pair when the validation fails

|  |  |
| --- | --- |
| 1  2  3 | return { 'gte': true, 'requiredValue': 10 } |

The 'gte': true: indicates that the validation has failed. 'requiredValue': 10 is used by the template to display that the expected value is greater than 10.

**Using the Custom Validator**

Since this is a template-driven form., we do not have to do anything in the component class. In the HTML template just add the attribute gteValidator as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <label for="numVal">Number :</label>      <input type="text" name="numVal" ngModel #numVal="ngModel" gteValidator>      <div \*ngIf="!numVal.valid && (numVal.dirty ||numVal.touched)">      <div \*ngIf="numVal.errors.gte">        The number should be greater than {{numVal.errors.requiredValue}}      </div>    </div> |

Validators return ValidationErrors. They are added to the control’s errors collection of the control. The valid property of the control is set to false.

Hence we check if the valid property. We also check the dirty and touched property. Because we do not want to display the error message when the form is displayed for the first time.

We check if the gte is true and display the error message. Note that gte is the name of the key we used while creating the validator.

We also make use of requiredValue to show a meaningful message to the user.

|  |  |
| --- | --- |
| 1  2  3  4  5 | <div \*ngIf="numVal.errors.gte">        The number should be greater than {{numVal.errors.requiredValue}}      </div> |

**Passing Parameter to Validator**

We have hardcoded the value of 10 in the above example. This will make our validator difficult to reuse. If we want to resue it, we need to pass the number to be checked as the parameter.

Since they are directives, we can use [Input decorator](https://angular.io/api/core/Input) to pass the parameter to the Validator.

Open the template add the special attribute gteNum="20"

|  |  |
| --- | --- |
| 1  2  3 | <input type="text" name="numVal" ngModel #numVal="ngModel" gteValidator gteNum="20" > |

You can read the gteNum from the template using the Input decorator as shown below

|  |  |
| --- | --- |
| 1  2  3 | @Input("gteNum") gteNum:number |

Now, you can remove the hardcoded value 10 and use the gteNum instead.

The complete validator code is as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | import { Validator, NG\_VALIDATORS, FormControl } from '@angular/forms'  import { Directive,  Input } from '@angular/core';    @Directive({    selector: '[gteValidator]',    providers: [      { provide: NG\_VALIDATORS, useExisting: gteValidatorDirective, multi: true }    ]  })  export class gteValidatorDirective implements Validator {      @Input("gteNum") gteNum:number      validate(c: FormControl) {        let v: number = +c.value;        if (isNaN(v)) {        return { 'gte': true, 'requiredValue': this.gteNum }      }        if (v <= +this.gteNum) {        return { 'gte': true, 'requiredValue': this.gteNum }      }        return null;    }    } |

**Injecting Service into Validator**

The validator may depend on some external service to validate the value. For Example, it may need to fetch data from the back end server.

Let us move the validation logic in the above validator to a separate service. Create a service gte.service.ts and copy the following code

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import { Injectable } from '@angular/core';    @Injectable({    providedIn: 'root',  })  export class gteService {      gte(num:any, requiredValue:Number) : Boolean {        if (isNaN(num)) {        return false;      }        if (num <= +requiredValue) {        return false;      }        return true;    }  } |

In the validation directive, create a constructor method and inject the service. The complete code is as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | import { Validator, NG\_VALIDATORS, FormControl } from '@angular/forms'  import { Directive,  Input } from '@angular/core';  import { gteService } from 'projects/injectService1/src/app/gte.service';    @Directive({    selector: '[gteValidator]',    providers: [      { provide: NG\_VALIDATORS, useExisting: gteValidatorDirective, multi: true }    ]  })  export class gteValidatorDirective implements Validator {      @Input("gteNum") gteNum:number      constructor(private gteService:gteService) {    }      validate(c: FormControl) {        let v: number = +c.value;        if (this.gteService.gte(v,this.gteNum)) {        return { 'gte': true, 'requiredValue': this.gteNum }      }        return null;    }  } |

**Summary**

We learned how to create a custom validator in template-driven forms. We also talked about how to pass parameter and inject service into our directive.