Introduction

In most of the frontend applications we use **HTML forms** for grouping HTML elements and we provide validations using **HTML5 attributes** like **required, minlength, maxlength, pattern** etc.

Every form does such similar tasks across all applications:

- Maintain the form state
- Tracking which part of form is valid and which part is invalid
- Displaying error message to how to make form valid
- Form submitting and resetting

Similarly, angular framework provides us alternative strategies to handle the forms they are:

- **Template Driven Forms (similar to Angular1 forms)**
- **Model Driven / Reactive Forms**

We will build this chapter in the following sequence of small steps

1. Create the Employee model class
2. Create the component that controls the form
3. Create a template with the initial form layout
4. Bind data properties to each form input control with the ngModel two-way data binding syntax
5. Add the name attribute to each form input control
6. Add custom CSS to provide visual feedback
7. Show and hide validation error messages
8. Handle form submission with **ngSubmit**
9. Disable the form's submit button until the form is valid
10. Resetting the form.

1. Create an Employee model class (employee.ts):

```
export class Employee
{
  id: number;
  name: string;
  salary: number;
  department: string;
  constructor(id: number, name: string, sal: number, dept: string)
  {
    this.id = id;
```

```
      this.name = name;
      this.salary = sal;
      this.department = dept;
   }
}
```

2. Create a Form Component (employee-form.component.ts)

```
import { Component } from '@angular/core';
import { Employee } from './employee';

@Component({
   moduleId: module.id,
   selector: 'employee-form',
   templateUrl: 'employee.component.html'
})
export class EmployeeFormComponent {
   departments = ['Sales', 'Marketing', 'HR', 'Accounts'];
   model = new Employee(1, "E1", 10000, this.departments[0]);
   submitted = false;
   onSubmit() { this.submitted = true; }
   // TODO: Remove this when we're done
   get diagnostic() { return JSON.stringify(this.model); }
}
```

3. Edit app.module.ts

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

import { AppComponent }  from './app.component';
import { EmployeeFormComponent } from './employee-form.component'

@NgModule({
  imports: [ BrowserModule, FormsModule ],
  declarations: [AppComponent, EmployeeFormComponent ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

4. Edit app.component.ts

```
import { Component } from '@angular/core';
@Component({
   selector: 'my-app',
   template: `
      <employee-form></employee-form>
`})
export class AppComponent {
}
```

5. Create an initial HTML Form Template (employee-form.component.html)

```
<div class="container">
   <h1>Employee Form</h1>
   <form>
      <div class="form-group">
         <label for="name">Id</label>
         <input type="text" class="form-control" id="id" [value]="model.id" name="id" required>
      </div>
      <div class="form-group">
         <label for="name">Name</label>
         <input type="text" class="form-control" id="name" [value]="model.name" name="name" required>
      </div>
      <div class="form-group">
         <label for="salary">Salary</label>
         <input type="text" class="form-control" id="salary" [value]="model.salary" name="salary" required>
      </div>
      <div class="form-group">
         <label for="name">Department</label>
         <select class="form-control" id="department" [value]="model.department"  name="department" required>
            <option *ngFor="let dept of departments" [value]="dept">{{dept}}</option>
         </select>
```

```
    </div>
    <button type="submit" class="btn btn-default">Submit</button>
  </form>
  <hr />
  Employee: {{diagnostic}}
</div>
```

6. Execute and test the application to note that it's currently performing only one-way binding. Changes made to textboxes are not reflected in the Employee object.

Two-way data binding with ngModel

7. Replace all occurrence of [value] with [(ngModel)] so that Model object and form elements are two-way binding.

```
<div class="container">
  <h1>Employee Form</h1>
  <form>
    <div class="form-group">
      <label for="name">Id</label>
      <input type="text" class="form-control" id="id" [(ngModel)]="model.id" name="id" required>
    </div>
    <div class="form-group">
      <label for="name">Name</label>
      <input type="text" class="form-control" id="name" [(ngModel)]="model.name" name="name" required>
    </div>
    <div class="form-group">
      <label for="salary">Salary</label>
      <input type="text" class="form-control" id="salary" [(ngModel)]="model.salary" name="salary" required>
    </div>
    <div class="form-group">
      <label for="name">Department</label>
      <select class="form-control" id="department" [(ngModel)]="model.department"  name="department"
required>
        <option *ngFor="let dept of departments" [value]="dept">{{dept}}</option>
      </select>
    </div>
    <button type="submit" class="btn btn-default">Submit</button>
  </form>
  <hr />
  Employee: {{diagnostic}}
</div>
```

8. Run the application and see that when values in form elements are changed, employee object is also changing accordingly.
9. Track change-state and validity with *ngModel*

State	Class if true	Class if false
Control has been visited	ng-touched	ng-untouched
Control's value has changed	ng-dirty	ng-pristine
Control's value is valid	ng-valid	ng-invalid

10. Edit employee-form.component.html as below

```
<div>
  <label for="name">Name</label>
  <input type="text" #empName id="name" [(ngModel)]="model.name" name="name" required pattern="[a-zA-Z0-9 ]+">
  {{empName.className}}
</div>
```

11. Run and see how styles are changing when the value name element is visited and changed.

Add Custom CSS for Visual Feedback.

We realize we can mark required fields and invalid data at the same time with a colored bar on the left of the input box:

12. Create a new file (content\forms.css)

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

13. To Index.html, add <link> for forms.css

```
<link rel="stylesheet" href="/content/forms.css">
```

Show and Hide Validation Error messages

We can leverage the ng-invalid class to reveal a helpful message.

For shorter validation expression instead of **f.form.controls.name?** We can get access of instance of ngModel by using local reference template variables

Ex: #empName="ngModel"

14. Edit employee.component.html as below (sample provided for name only)

```
<div class="form-group">
  <label for="name">Name</label>
  <input type="text" #empName="ngModel" class="form-control" id="name" name="name"
                                          [(ngModel)]="model.name" required pattern="[a-zA-Z0-9 ]+">
  <div [hidden]="empName.valid || empName.pristine|| !empName.errors.required" class="alert
alert-danger">
     Name is required
  </div>
  <div *ngIf="(!empName.valid || empName.pristine) && empName.errors.pattern">
     Name cannot contain special charactes !, @, #, $ etc..
  </div>
  {{empName.className}}
</div>
```

Note that the value of template reference variable is provided as ngModel

15. Run and note how "Name is required" will be displayed when value is not provided.

Resetting the form.

16. Set template reference variable for <form>

```
<form #empForm novalidate>
```

17. Add a new button (employee-form.component.html)

```
<button type="button" (click)="newEmployee();" class="btn btn-default">New Employee</button>
```

18. Add a new method to component (employee-form.component.ts)

```
newEmployee() {
  alert("Now adding")
  this.model = new Employee(0, '', 0,'');
}
```

19. Run the application again, click the *New Employee* button, and the form clears. The *required* bars to the left of the input box are red, indicating invalid name property. That's understandable as these are required fields. The error messages are hidden because the form is pristine; we haven't changed anything yet.

Resetting form state:

1. Enter a name and click *New Employee* again. The app displays a ***Name is required*** error message!
2. We don't want error messages when we create a new (empty) employee. Why are we getting one now?
3. Inspecting the element in the browser tools reveals that the *name* input box is ***no longer pristine***. The form remembers that we entered a name before clicking *New Employee*. Replacing the employee object *did not restore the pristine state* of the form controls.

20. We have to clear all of the flags imperatively which we can do by calling the form's reset() method after calling the newEmployee() method.

```
<button type="button" (click)="newEmployee(); empForm.reset() " class="btn btn-default">New
Employee</button>
```

Submit the form with *ngSubmit*

A "form submit" is useless at the moment. To make it useful, we'll update the <form> tag with another Angular directive, NgSubmit, and bind it to the EmployeeFormComponent.onSubmit() method with an event binding.

21. Edit <form> tag as below

```
<form #empForm="ngForm" (ngSubmit)="onSubmit(empForm);" novalidate>
```

22. Replace <input type="button" …> with the following.

    Bind the button's disabled property to the form's over-all validity via the empForm variable.

```
<button type="submit" class="btn btn-default" [disabled]="!empForm.valid">Submit</button>
```

23. Add the below method to EmployeeFormComponent

```
onSubmit(form: any)
{
  alert(form.value.id + " " + form.value.name)
  //alert(this.model.id + " " + this.model.name)
  this.model = new Employee(0, '', 0, '');
  alert("Form data is saved...");
}
```

24. Re-run the application. The form opens in a valid state and the button is enabled.

    Now delete the *Name*. We violate the "name required" rule which is duly noted in our error message as before. **And now the Submit button is also disabled.**

Resetting Form:

In template driven we don't have access to the form model in the component. So, to achieve this we can use **@ViewChild** decorator.

@ViewChild: This decorator gives the reference of anything from the template in the component.

Create a component variable with **@ViewChild** decorator applied to it.

Now we can call the **reset()** in **onSubmit()** like we have used in model driven forms.

```
<form #empForm="ngForm" (ngSubmit)="onSubmit();">


export class EmployeeFormComponent{
  @ViewChild('empForm') empForm: any;
  onSubmit() {
    //Use AJAX here…
   alert(this.empForm.value.id + " " + this.empForm.value.name)
    this.empForm.reset();
  }
}
```