Angular Forms Tutorial: Complete Guide

15 Comments / 8 minutes of reading / April 15, 2024

← ngDocheck Angular Tutorial Template Driven Forms →

The Angular forms handle user interactions. We use them to collect data from the user. For Example, users login information, updating his profile or placing an order, etc. In this Angular Forms tutorial and in the subsequent tutorials that follow, we learn about Angular Forms, their building blocks, how to create Reactive & Template driven forms, etc. We also going to learn how to create typed & untyped forms, validate forms, listen to user input changes etc with example applications.

Table of Contents

Angular Forms Tutorial

The data entry forms can be very simple to very complex. It can contain large no of input fields, Spanning multiple tabs. Forms may also contain complex validation logic interdependent on multiple fields.

Some things forms are expected to do

- Initialize the forms fields and present it to the user
- Capture the data from the user
- Track changes made to the fields
- Validate the inputs
- Display helpful errors to the user

List of all articles on this Angular Forms Tutorial

Angular Forms Module

The Angular Forms module provides all the above services out of the box. It binds the form field to the Angular component class. It tracks changes made to the form fields so that we can respond accordingly. It also provides the built-in validators to validate the inputs. You can create your custom validator. It presents the validation errors to the user. Finally, it encapsulates all the input fields into an object structure when the user submits the form.

Angular takes two approaches to build the forms. One is Template-driven forms approach and another one is Reactive forms or model-driven forms approach

Template-driven forms approach

In Template-driven approach is the easiest way to build the Angular forms. The logic of the form is placed in the template. It allows us to create sophisticated looking forms easily without writing any JavaScript code.

Model-driven forms approach (Reactive Forms)

In Reactive Forms or Model-driven approach, the logic of the form is defined in the component as an object. The Model-driven approach has more benefits as it makes the testing of the component easier.

In this approach, the representation of the form is created in the component class. where Form fields are created as properties of our component class. This form model is then bound to the HTML elements using the special markups. This makes it easier to test.

Strictly Typed Reactive Forms

Before Angular 14, the Angular Forms did not provide any mechanism for type checking. i.e. you could easily assign a number to string or to a boolean FormControl and get away with that. Accessing non existing property on FormGroup did not throw any compiler error. All of these either resulted in run time error either while testing or at production use.

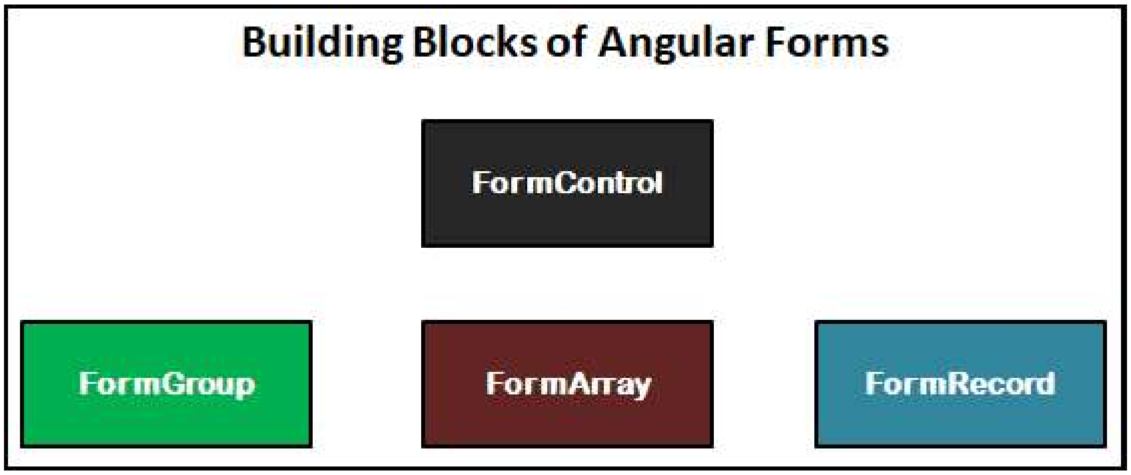
Starting from Angular 14, you can create the typed Angular Forms, which can detect type errors at compile time.

You can create a typed form, just by declaring and initializing the form variable together, which should suffice in most of the situations. Angular infers the type of the form, from the initialization code. You also have the option to create a custom type and assign it to Form Variable.

There is no other change required to use the Typed Angular Forms.

Building Blocks of Angular Forms

The Angular Forms module consists of **four Building blocks**, irrespective of whether you are using Template-driven (**Except FormRecord**) or Reactive forms approach.



FormControl

A FormControl represents a single input field in an Angular form.

Consider a simple Text input box

```

First Name : <input type="text" name="firstname" />

```

As a developer, you would like to know the current value in the Text box. You would also be like to know if the value is valid or not. If the user has changed the value(dirty) or is it unchanged. You would like to be notified when the user changes value.

The FormControl is an object that encapsulates all this information related to the single input element. It Tracks the value and validation status of each of these control.

The FormControl is just a class. A FormControl is created for each form field. We can refer them in our component class and inspect its properties and methods.

You can use FormControl to set the value of the Form field, find the status of form field like (valid/invalid, pristine/dirty, touched/untouched) etc & add validation rules to it.

The above input field is created using the FormControl as shown below.

```

let firstname= new FormControl('');        //Creating a FormControl in a Reactive forms

```

Then, you can retrieve the current value in the input field using the value property

```

firstname.value   //Returns the value of the first name field

```

You can check the validation status of the First Name element as shown below

```

firstname.errors      // returns the list of errors
firstname.dirty      // true if the value has changed (dirty)
firstname.touched     // true if input field is touched
firstname.valid      // true if the input value has passed all the validation

```

FormGroup

FormGroup is a collection of FormControls . Each FormControl is a property in a FormGroup with the control name as the key.

Often forms have more than one field. It is helpful to have a simple way to manage the Form controls together.

Consider the following Form. we have three input fields street, city & Pincode.

```

city : <input type="text" name="city" >
Street : <input type="text" name="street" >
PinCode : <input type="text" name="pincode" >

```

All of the above input fields are represented as the separate FormControl. If we wanted to check the validity of our form, we have to check the validity of each and every FormControl for validity. Imagine Form having large no of fields. It is cumbersome to loop over large no of FormControls and check for validity.

FormGroup solve's this issue by providing a wrapper interface around a collection of FormControls A FormGroup tracks the status of each child FormControl and aggregates the values into one object. with each control name as the key

We can group these input fields under the group address as shown below

```

let address= new FormGroup({
    street : new FormControl(""),
    city : new FormControl(""),
    pinCode : new FormControl("")
})

```

In the above example, the address is our `FormGroup`, consisting of 3 Form Controls `city`, `street`, and `pinCode`. Now we can check the validity of the entire group together. For example, if the state is invalid, then the address FormGroup returns the invalid state.

You can read the value of an address using the value method, which returns the JSON object as shown below

```

address.value

```

The Return value

```

address {
    street :"",
    city:"",
    Pincode:""
}

```

You can access child control as

```

```

```
address.get("street")

```

Check the Validation status as follows

```

address.errors     // returns the list of errors
address.dirty      // true if the value of one of the child control has changed (dirty)
address.touched    // true if one of the child control is touched
address.valid      // true if all the child controls passed the validation

```

A typical Angular Form can have more than one FormGroup. A FormGroup can also contain another FormGroup.

The Angular form is itself a FormGroup

FormArray

FormArray is an array of form controls. It is similar to FormGroup except for one difference. In FormGroup each FormControl is a property with the control name as the key. In FormArray is an array of form controls.

We define the FormArray as shown below

```

contactForm = new FormGroup( {
    name: new FormControl(''),
    cities:new FormArray([
      new FormControl('Mumbai'),
      new FormControl('Delhi')
    ])
  });

```

You can get the reference to the `cities` from the `contactForm.get` method

```

 cities() :FormArray {
    return this.contactForm.get("cities") as FormArray
  }

```

Check the Validation status as follows

```

cities.errors      // returns the list of errors
cities.dirty       // true if the value of one of the child control has changed (dirty)
cities.touched     // true if one of the child control is touched
cities.valid       // true if all the child controls passed the validation

```

FormRecord

The `FormRecord` is also a collection of FormControl. It is very similar to FormGroup, but allows us to add FormControl's dynamically at run time.

```

cities!: FormRecord<FormControl<string | null>>;

```

Now, you can add new FormControl to cities FormRecord . In the example below a new FormControl with key mumbai and delhi added to the cities FormRecord.

```

this.cities.addControl('mumbai', new FormControl(''));
this.cities.addControl('delhi', new FormControl(''));

```

Both FormRecord and FormArray allows us to add or remove FormControl ‘s at runtime. The difference is how they are structured. In FormRecord controls becomes a property of the FormRecord . Each control is represented as key-value pair, while in FormArray , the controls become part of an array.

Click to learn more about FormRecord.

Summary

In this tutorial, we learned what is Angular Forms all about. We looked at the basic building blocks of Angular Forms i.e. FormGroup , FormControl , FormArray & FormRecord . The Angular allows us to build Forms using two different approaches. One is Template Driven & another one is Reactive Forms or Model-driven. In the next few tutorials, we look at how to build Angular Forms using both of these options.

References

Forms Overview

Read More

1. Angular Forms Tutorial: Fundamental & Concepts (This article)
2. Template Driven Forms in Angular
3. Set Value in Template Driven forms in Angular
4. Reactive Forms in Angular
5. FormBuilder in Reactive Forms
6. SetValue & PatchValue in Angular
7. StatusChanges in Angular Forms
8. ValueChanges in Angular Forms
9. FormControl
10. FormGroup
11. FormArray Example
12. Build Dynamic or Nested Forms using FormArray
13. Validations in Reactive Forms in Angular
14. Custom Validator in Reactive Forms
15. Passing Parameter to Custom Validator in Reactive Forms
16. Inject Service into Custom Validator
17. Validation in Template Driven Forms
18. Custom Validator in Template Driven Forms

← ngDocheck Angular Tutorial Template Driven Forms →

Related Posts