FormControl in Angular

2 Comments / 10 minutes of reading / March 9, 2023

`FormControl` sets and tracks the individual HTML form element. it is one of the building blocks of the angular forms. The other two are FormGroup and FormArray. In this tutorial, we will learn what is FormControl is and learn some of the properties & methods of it.

Table of Contents

What is FormControl

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 its value.

The `FormControl` is an object that encapsulates all the 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 to them in our component class and inspect its properties and methods

We 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. You can add validation rules to it.

Using FormControl

The Angular has two approaches to building the Angular Forms. One is Template-driven and the other one is Reactive Forms.

To use the Angular forms, First, we need to import the `FormsModule` (for template-driven forms) & `ReactiveFormsModule` (for Reactive Forms) from the `@angular/forms` in your route module.

```

import { FormsModule, ReactiveFormsModule } from '@angular/forms';

```

Also, add it to the `imports` metadata.

```

imports: [
   BrowserModule,
   AppRoutingModule,
   FormsModule,
   ReactiveFormsModule
],

```

Reactive Forms

In Reactive Forms approach, It is our responsibility to build the Model using `FormGroup`, `FormControl` and `FormArray`.

To use `FormControl`, first, we need to import the `FormControl` from the `@angular/forms`

```

import { FormGroup, FormControl, Validators } from '@angular/forms'

```

Then create the top-level FormGroup. The first argument to `FormGroup` is the collection of `FormControl`. They are added using the `FormControl` method as shown below.

```

reactiveForm = new FormGroup({
  firstname: new FormControl('',[Validators.required]),
  lastname: new FormControl(),
  email: new FormControl(),
})
```

```

```

Or you can make use of the FormBuilder API

```

this.reactiveForm = this.formBuilder.group({
  firstname: ['',[Validators.required]],
  lastname: [''],
  email: [''],
});

```

Bind the form element with the template using the **formControlName** directive as shown below

```

<form [formGroup]="reactiveForm" (ngSubmit)="onSubmit()" novalidate>

  <p>
    <label for="firstname">First Name </label>
    <input type="text" id="firstname" name="firstname" formControlName="firstname">
  </p>

  <p>
    <label for="lastname">Last Name </label>
    <input type="text" id="lastname" name="lastname" formControlName="lastname">
  </p>

  <p>
    <label for="email">Email </label>
    <input type="text" id="email" name="email" formControlName="email">
  </p>

  <p>
    <button type="submit">Submit</button>
  </p>

</form>

```

Template-driven forms

In template-driven forms, the `FormControl` is defined in the Template. The `<Form>` directive creates the top-level `FormGroup`. We use the `ngModel` directive on each Form element, which automatically creates the `FormControl` instance.

```

<form #templateForm="ngForm" (ngSubmit)="onSubmit(templateForm)" novalidate>

  <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>
    <button type="submit">Submit</button>
  </p>

</form>

```

Use the `viewChild` to get the reference to the `FormModel` in the Component class. The `control` property of the `NgForm` returns the top-level `formgroup`

```

@ViewChild('templateForm',null) templateForm: NgForm;

```

Setting the value

setValue()

abstract setValue(value: any, options?: Object): void

We use setValue or patchValue method of the FormControl to set a new value for the form control. There is no difference between setValue and patchValue at the FormControl level.

```

setEmail() {
  this.reactiveForm.get("email").setValue("sachin.tendulakar@gmail.com");
};

```

```

setEmail() {
  this.templateForm.control.get("email").setValue("sachin.tendulkar@gmail.com");
};

```

patchValue()

abstract patchValue(value: any, options?: Object): void

```

setEmail() {
  this.reactiveForm.get("email").setValue("sachin.tendulakar@gmail.com");
};

```

```

setEmail() {
  this.templateForm.control.get("email").setValue("sachin.tendulkar@gmail.com");
};

```

Must Read: setValue & patchValue in Angular forms

Two-way binding

The two-way data binding is the preferred way to to keep the component model in sync with the `FormModel` in Template-driven forms.

```

  <p>
    <label for="firstname">First Name </label>
    <input type="text" id="firstname" name="firstname" [(ngModel)]="contact.firstname">
  </p>

```

Using two-way data binding in Reactive forms is deprecated since the Angular 7

Finding the Value

value

`value: any`

The value returns the current value of `FormControl` It is Readonly. To set the value of the control either use the `setValue` or `patchValue` method

```

//reactive forms
this.reactiveForm.get("firstname").value

//template driven forms
```

```
this.templateForm.control.get("firstname").value

```

valueChanges

valueChanges: Observable<any>

The angular emits the valueChanges event whenever the value of the control changes. The value may change when the user updates the element in the UI or programmatically through the setValue / patchValue method. We can subscribe to it as shown below

```

//reactive Forms

this.fNameChange = this.reactiveForm.get("firstname").valueChanges.subscribe(x => {
   console.log(x);
})

```

Similarly in template-driven forms.

```

setTimeout(() => {
   this.fNameChange = this.templateForm.control.get("firstname").valueChanges.subscribe
     console.log(x);
   })
});

```

Must Read: ValueChanges in Angular

Control Status

The `FormControl` tracks the validation *status* of the HTML Element to which it is bound. The following is the list of status-related properties

status

```
status: string
```

The Angular runs validation checks, whenever the value of a *form control* changes. Based on the result of the validation, the control can have four possible states.

VALID: The FormControl has passed all validation checks.
INVALID: This control has failed at least one validation check.
PENDING: This control is in the midst of conducting a validation check.
DISABLED: This control is exempt from validation checks

```

//reactive forms
this.reactiveForm.get("firstname").status

//template driven forms
this.templateForm.control.get("firstname").status

```

valid

```
valid: boolean
```

A control is valid when it has passed all the validation checks and is not disabled.

```

this.reactiveForm.get("firstname").valid

```

invalid

invalid: boolean

A control is invalid when it has failed one of the validation checks and is not disabled

```

this.reactiveForm.get("firstname").invalid

```

pending

pending: boolean

A control is pending when it is in the midst of conducting a validation check.

```

this.reactiveForm.get("firstname").pending

```

disabled

disabled: boolean

Control is disabled when its status is DISABLED.

```

this.reactiveForm.get("firstname").disabled

```

enabled

enabled: boolean

Control is enabled as long as the status is not DISABLED.

```

this.reactiveForm.get("firstname").disabled

```

pristine

pristine: boolean

Control is pristine if the user has not yet changed the value in the UI.

```

this.reactiveForm.get("firstname").pristine

```

dirty

dirty: boolean

Control is dirty if the user has changed the value in the UI.

```

```

```
this.reactiveForm.get("firstname").dirty

```

touched

touched: boolean

True if the control is marked as touched. A control is marked touched once the user has triggered a blur event on it.

```

this.reactiveForm.get("firstname").touched

```

untouched

untouched: boolean

True if the control has not been marked as touched. A control is untouched if the user has not yet triggered a blur event on it.

```

this.reactiveForm.get("firstname").untouched

```

Changing the Status

We can also change the status of the control programmatically by using the following methods.

When we change the status of a control programmatically or via UI, the validity & value of the parent control is also calculated and updated. There may arise circumstances when you do not want that to happen. In such circumstances, you can make use of the `onlySelf:true` to ensure that the parent control is not checked.

markAsTouched

This method will mark the control as `touched`.

```
markAsTouched(opts: { onlySelf?: boolean; } = {}): void
```

- `onlySelf` if true then only this control is marked. If false it will also mark all its direct ancestors also as touched. The default is false.

```

this.reactiveForm.get("firstname").markAsTouched()
this.reactiveForm.get("firstname").markAsTouched({ onlySelf:true; })

```

markAllAsTouched

```
markAllAsTouched(): void
```

Marks the control and all its descendant controls as touched.

markAsUntouched

```
markAsUntouched(opts: { onlySelf?: boolean; } = {}): void
```

Marks the control as untouched.

- `onlySelf` if true only this control is marked as untouched. When false or not supplied, mark all direct ancestors as untouched. The default is false.

markAsDirty

```
markAsDirty(opts: { onlySelf?: boolean; } = {}): void
```

Marks the control as dirty. A control becomes dirty when the control's value is changed through the UI.

- `onlySelf` if true, only this control is marked as dirty else all the direct ancestors are marked as dirty. The default is false.

markAsPristine

```
markAsPristine(opts: { onlySelf?: boolean; } = {}): void
```

Marks the control as pristine.

- `onlySelf` if true, only this control is marked as pristine else all the direct ancestors are marked as pristine. The default is false.

markAsPending

```
markAsPending(opts: { onlySelf?: boolean; emitEvent?: boolean; } = {}): void
```

Marks the control as pending. We mark it as pending when the control is in the midst of conducting a validation check.

- `onlySelf` : When true, mark only this control. When false or not supplied, mark all direct ancestors. The default is false.
- `emitEvent` : When true or not supplied (the default), the `statusChanges` observable emits an event with the latest status the control is marked pending. When false, no events are emitted.

disable

```
disable(opts: { onlySelf?: boolean; emitEvent?: boolean; } = {}): void
```

Disables the control. This means the control is exempt from validation checks and excluded from the aggregate value of any parent. Its status is DISABLED.

- `onlySelf` : When true, mark only this control. When false or not supplied, mark all direct ancestors. Default is false..
- `emitEvent` : When true or not supplied (the default), both the `statusChanges` and `valueChanges` observables emit events with the latest status and value when the control is disabled. When false, no events are emitted.

enable

```
enable(opts: { onlySelf?: boolean; emitEvent?: boolean; } = {}): void
```

Enables control. This means the control is included in validation checks and the aggregate value of its parent. Its status recalculates based on its value and its validators.

- onlySelf : When true, mark only this control. When false or not supplied, mark all direct ancestors. The default is false.
- emitEvent : When true or not supplied (the default), both the statusChanges and valueChanges observables emit events with the latest status and value when the control is enabled. When false, no events are emitted.

Status Change Event

statusChanges

statusChanges: Observable<any>

We can subscribe to the status changes event by subscribing it to the statusChanges as shown below. The event is fired whenever the validation status of the control is calculated.

```

//Reactive Forms
this.reactiveForm.get("firstname").statusChanges.subscribe(x => {
  console.log(x);
})

//Template Driven Forms
this.templateForm.control.get("firstname").statusChanges.subscribe(x => {
   console.log(x);
})

```

Must Read: StatusChanges in Angular

Validation

The way we add validators depends on whether we use the Template-driven forms or reactive forms.

In Reactive forms, the validators are added while declaring the controls

```

reactiveForm = new FormGroup({
  firstname: new FormControl('',[Validators.required]),
  lastname: new FormControl(),
  email: new FormControl(),
})

```

While in the template-driven forms in the template

```

<p>
  <label for="firstname">First Name </label>
  <input type="text" id="firstname" name="firstname" ngModel required >
</p>

```

updateValueAndValidity()

updateValueAndValidity(opts: { onlySelf?: boolean; emitEvent?: boolean; } = {}): void

The `updateValueAndValidity` forces the form to perform validation. This is useful when you add/remove validators dynamically using `setValidators`, `RemoveValidators` etc

- onlySelf : When true, only update this control. When false or not supplied, update all direct ancestors. Default is false..
- emitEvent : When true or not supplied (the default), both the statusChanges and valueChanges observables emit events with the latest status and value when the control is updated. When false, no events are emitted.

```

//reactive forms
this.reactiveForm.get("firstname").updateValueAndValidity();

//template driven forms
this.templateForm.control.get("firstname").updateValueAndValidity();

```

setValidators() / setAsyncValidators()

Programmatically adds the sync or async validators. This method will remove all the previously added sync or async validators.

setValidators(newValidator: ValidatorFn | ValidatorFn[]): void

setAsyncValidators(newValidator: AsyncValidatorFn | AsyncValidatorFn[]): void

```

//Reactive Form
setValidator() {
  this.reactiveForm.get("firstname").setValidators([Validators.required, Validators.minLengt
  this.reactiveForm.get("firstname").updateValueAndValidity();
}

```

```

//Template driven forms
setValidator() {
  this.templateForm.control.get("firstname").setValidators([Validators.required, Validators.
  this.templateForm.control.get("firstname").updateValueAndValidity();
}

```

clearValidators() / clearAsyncValidators()

clearValidators(): void

clearAsyncValidators(): void

clearValidators & clearAsyncValidators clears all validators.

```

//reactive forms
clearValidation() {
   this.reactiveForm.get("firstname").clearValidators();
   this.reactiveForm.get("firstname").updateValueAndValidity();
}

```

```

//template driven forms
clearValidation() {
  this.templateForm.control.get("firstname").clearValidators();
  this.templateForm.control.get("firstname").updateValueAndValidity();
}

```

errors()

errors: ValidationErrors | null

An object containing any `errors` generated by failing validation, or null if there are no errors.

```

getErrors() {

  const controlErrors: ValidationErrors = this.reactiveForm.get("firstname").errors;
  if (controlErrors) {
    Object.keys(controlErrors).forEach(keyError => {
      console.log("firtname "+ ' '+keyError);
    });
  }
}

```

setErrors()

setErrors(errors: ValidationErrors, opts: { emitEvent?: boolean; } = {}): void

```

setErrors() {
   this.reactiveForm.get("firstname").setErrors( {customerror:'custom error'});
}

```

getError()

getError(errorCode: string, path?: string | (string | number)[]): any

Reports error data for the control with the given path.

```

this.reactiveForm.getError("firstname")

//
this.reactiveForm.getError("address.pincode");
this.reactiveForm.getError(["address","pincode"]);

```

hasError

hasError(errorCode: string, path?: string | (string | number)[]): boolean

Reports whether the control with the given path has the error specified.

```

this.reactiveForm.hasError("firstname")
```

```
//
this.reactiveForm.hasError("address.pincode");
this.reactiveForm.hasError(["address","pincode"]);
```

Reset

abstract reset(value?: any, options?: Object): void

Resets the control. We can also pass the default value.

```
this.reactiveForm.get("firstname").reset('');
this.reactiveForm.get("firstname").reset('test');
```

Summary

In this tutorial, we learned what is FormControl is and looked at the various methods & properties that are available.

List of All tutorials on Angular Forms

1. Angular Forms Tutorial: Fundamental & Concepts
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