# FormGroup in Angular

2 Comments / 15 minutes of reading / March 20, 2024

⟵ FormControl Angular Tutorial Form Array Example ⟶

The FormGroup is a collection of Form controls It Tracks the value and validity state of a group of Form control instances. The FormGroup is one of the building blocks of the angular forms. The other two are FormControl and FormArray. In this article, we will learn what is FormGroup is and learn some of its important properties & methods

Table of Contents

## What is FormGroup

Consider a simple HTML form

```

<form>
  First Name : <input type="text" name="firstname" />
  Last Name  : <input type="text" name="lastname" />
  Email      : <input type="text" name="email" />
</form>

```

All of the above input fields are represented as the separate FormControl instance. We can use the FormControl instance to check the value & validity of each field. But if we wanted to check the validity of our entire form, we have to check the validity of each and every `FormControl` for validity. Imagine a form having a large no of fields. It is cumbersome to loop over large no of FormControl and check for validity.

The `FormGroup` solve this issue by providing a wrapper around a collection of FormControl's It encapsulates all the information related to a group of form elements. It Tracks the value and validation status of each of these control. We can use it to check the validity of the elements. set its values & listen for change events, add and run validations on the group, etc

The `FormGroup` is just a class. We create a `FormGroup` to organize and manage the related elements. For Example form elements like address, city, state, pin code, etc can be grouped together as a single FormGroup. It makes it easier to manage them. A

FormGroup aggregates the values of each child FormControl into one object, with each control name as the key. It calculates its status by reducing the status values of its children. For example, if one of the controls in a group is invalid, the entire group becomes invalid.

## Using FormGroup

**Every Angular Form must have at least one top-level FormGroup. We use It to track the value & validity of the entire form.**

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, we create the form model in the component class. First, we need to import the FormGroup, FormControl, & Validators.

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

When instantiating a FormGroup, pass in a collection of child controls as the first argument. The key for each child registers the name for the control.

The following form model has two Form Groups. One is the top-level Form group, which we have named as `reactiveForm`. The other one is nested Form Group, which we have named it as `address`.

```

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

```

And in the Template, we use `formGroup`, `formControlName` and `formGroupName` directive to bind the Form to the template.

```

<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>

  <div formGroupName="address">

    <p>
      <label for="address">Address</label>
      <input type="text" class="form-control" name="address" formControlName="addres
    </p>

    <p>
      <label for="city">City</label>
      <input type="text" class="form-control" name="city" formControlName="city">
    </p>

    <p>
      <label for="state">State</label>
      <input type="text" class="form-control" name="state" formControlName="state">
    </p>

  </div>

  <button>Submit</button>

</form>

```

## Template Driven forms

In Template-driven forms. the model is built in the template first. The top-level form is bound to ngForm directive, which we have named as `templateForm`. We add `ngModel` directive to each form element to create Form Controls. The name attribute will become the name of the Form Control. The ngModelGroup directive is used to create the nested Form Group.

```

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

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

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


  <div ngModelGroup="address">

    <p>
      <label for="address">Address</label>
      <input type="text" class="form-control" name="address" ngModel>
    </p>

    <p>
      <label for="city">City</label>
      <input type="text" class="form-control" name="city" ngModel>
    </p>

    <p>
      <label for="state">State</label>
      <input type="text" class="form-control" name="state" ngModel>
    </p>
  </div>

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

 </form>
```

```

```

We can get the reference to the top-level form group in component class using the `ViewChild` as shown below.

```

import { NgForm, Validators } from '@angular/forms';

export class TemplateComponent implements OnInit {
  @ViewChild('templateForm', null) templateForm: NgForm;

  ....
}

```

## Setting Value

We use setValue or `patchValue` method of the `FormGroup` to set a new value for the entire FormGroup.

### SetValue

Sets the value of the `FormGroup`. It accepts an object that matches the structure of the group, with control names as keys. The structure must match exactly, otherwise, it will result in an error.

setValue(value: { [key: string]: any; }, options: { onlySelf?: boolean; emitEvent?: boolean; } = {}): void

```

setValue() {

  this.reactiveForm.setValue({
    firstname: "Sachin",
    lastname: "Tendulakr",
    email: "sachin@gmail.com",
    address: {
      address: "19-A, Perry Cross Road, Bandra (West)",
      city: "Mumbai",
      state: "Maharatsra",
    }
  })
}

```

You can also update the nested FormGroup separately,

```

setAddress() {
  this.reactiveForm.get("address").setValue({
    address: "19-A, Perry Cross Road, Bandra (West)",
    city: "Mumbai",
    state: "Maharatsra",
  })
}

```

## patchValue

Patches the value of the FormGroup. It accepts an object with control names as keys and does its best to match the values to the correct controls in the group.

patchValue(value: { [key: string]: any; }, options: { onlySelf?: boolean; emitEvent?: boolean; } = {}): void

```

patchValue() {

  this.reactiveForm.patchValue({
    email: "sachin@gmail.com",
    address: {
      state: "Maharatsra",
    }
  })
}

```

We can Both setValue & patchValue

- onlySelf : When true, each change only affects this control and not its parent. The default is true.
- 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 value is updated. When false, no events are emitted. The configuration options are passed to the updateValueAndValidity method.

## Finding the Value

### value

The value returns the object with a key-value pair for each member of the Form Group. It is Readonly. To Set Value either setValue or patchValue

value: any

```

onSubmit() {
  console.log(this.reactiveForm.value);
}

```

### valueChanges

valueChanges: Observable<any>

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

In the example below, the first valuesChanges are fired, when any of the control is changed. While the second valuesChanges event is raised only when the controls under the address form group are changed

```

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

```

## Adding Controls Dynamically to Form Group

We usually add controls, while initializing the FormGroup.

```

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

}

```

The Forms API also allows add controls dynamically

## addControl()

Adds a control to the FormGroup and also updates validity & validation status. If the control already exists, then ignores it

addControl(name: string, control: AbstractControl): void

```

  addControl() {
    this.middleName = new FormControl('', [Validators.required]);
    this.reactiveForm.addControl("middleName",this.middleName);
  }

```

## registerControl()

Adds control to this FormGroup but does not update the validity & validation status. If the control already exists, then ignores it

registerControl(name: string, control: AbstractControl): AbstractControl

```

registerControl() {
  this.middleName = new FormControl('', [Validators.required]);
  this.reactiveForm.addControl("middleName",this.middleName);
}

```

## removeControl()

This method will remove the control with the provided name from the FormGroup.

removeControl(name: string): void

```

```

```
remodeControl() {
  this.reactiveForm.removeControl("middleName");
}

```

### setControl()

Replaces the control with the provided name with the new control.

setControl(name: string, control: AbstractControl): void

```

setControl() {
  this.middleName = new FormControl('test', [Validators.required]);
  this.reactiveForm.setControl("middleName",this.middleName);
}

```

### contains()

Check whether the control with the provided name exists or not..

contains(controlName: string): boolean

```

containsControl() {
  console.log(this.reactiveForm.contains("middleName"));
}

```

## Control Status

The FormGroup tracks the validation *status* of all the FormControls, which is part of the FormGroup. That also includes the status of nested FormGroup or FormArray. If any of the control becomes invalid, then the entire FormGroup becomes invalid.

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 FormGroup can have four possible states.

**VALID:** All the controls of the FormGroup has passed all validation checks.
**INVALID:** At least one of the control has failed at least one validation check.
**PENDING:** This Group is in the midst of conducting a validation check.
**DISABLED:** This FormGroup is exempt from validation checks

```

//reactive forms
this.reactiveForm.status

```

### valid

valid: boolean

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

```

this.reactiveForm.valid

```

### invalid

invalid: boolean

A FormGroup is invalid when one of its controls has failed a validation check or the entire FormGroup is disabled.

```

this.reactiveForm.invalid

```

### pending

pending: boolean

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

```

this.reactiveForm.pending

```

### disabled

disabled: boolean

A FormGroup is disabled when all of its controls are disabled.

```

```

```
this.reactiveForm.disabled

```

### enabled

enabled: boolean

A FormGroup is enabled as long one of its control is enabled.

```

this.reactiveForm.disabled

```

### pristine

pristine: boolean

A FormGroup is pristine if the user has not yet changed the value in the UI in any of the controls

```

this.reactiveForm.pristine

```

### dirty

dirty: boolean

A FormGroup is dirty if the user has changed the value in the UI in any one of the control.

```

this.reactiveForm.dirty

```

### touched

touched: boolean

True if the FomGroup is marked as touched. A FormGroup is marked as touched once the user has triggered a blur event on any one of the controls

```

this.reactiveForm.touched

```

### untouched

untouched: boolean

True if the FormGroup has not been marked as touched. A FormGroup is untouched if the user has not yet triggered a blur event on any of its child controls

```

this.reactiveForm.untouched

```

## Changing the Status

We can also change the status of the FormGroup by using the following method.

### markAsTouched

The FormGroup is marked as touched if anyone of its control is marked as touched. The control is marked as touched once the user has triggered a blur event on it.

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.

In the following example, the City is marked as touched. It will also mark both the address & reactiveFormGroup as touched.

```

markCityAsTouched() {
  this.reactiveForm.get("address").get("city").markAsTouched();
}

```

By Passing the onlySelf:true argument, you can ensure that only the city is marked as touched , while address & reactiveForm are not affected.

```

markCityAsTouched() {
  this.reactiveForm.get("address").get("city").markAsTouched({onlySelf:true});
}

```

The following code marks the address FormGroup as touched. while the child controls are not marked as touched . The parent FormGroup is marked as touched.

```

markAddressAsTouched() {
  this.reactiveForm.get("address").markAsTouched();
}

```

While onlySelf:true marks only the address group as touched, leaving the top-level FormGroup

```

```

```
markAddressAsTouched() {
  this.reactiveForm.get("address").markAsTouched({onlySelf:true});
}

```

## markAllAsTouched

Marks the control and all its descendant controls as touched.

markAllAsTouched(): void

The following example marks the address and all its controls i.e `city` , `state` & `address` as `touched` . The parent `FormGroup` stays as it is.

```

markAllAddressTouched() {
  this.reactiveForm.get("address").markAllAsTouched();
}

```

## markAsUntouched

Marks the control as untouched.

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

- `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.

The following code will mark the city as untouched. It will recalculate the untouched & touched status of the parent Group. If all the other controls are untouched then the parent FormGroup `address` is also marked as untouched.

```

markCityAsUnTouched() {
  this.reactiveForm.get("address").get("city").markAsUntouched();
}

```

By using the onlySelf:true you can ensure that only the city is marked as untouched, leaving the parent FormGroup as it is.

```

markCityAsUnTouched() {
  this.reactiveForm.get("address").get("city").markAsUntouched({onlySelf:true});
}

```

Similarly, you can mark the entire FormGroup as untouched. While this does not have any effect on the child controls, but it does recalculate the untouched status of the parent FormGroup. You can use the onlySelf:true ensure that it does not happen.

```

markAddressAsUnTouched() {
  this.reactiveForm.get("address").markAsUntouched();
}

```

## markAsDirty

The FormGroup becomes dirty when any one of its control is marked as dirty. A control becomes dirty when the control's value is changed through the UI. We can use the markAsDirty method to manipulate the dirty status.

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

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

The Following code marks the entire form as dirty. It does not change the status of any of the child controls.

```

markFormAsDirty() {
  this.reactiveForm.markAsDirty();
}

```

The following code marks the City as dirty. It will also change the Dirty status of Parent FormGroup.

```

markCityAsDirty() {
  this.reactiveForm.get("address").get("city").markAsDirty();
}

```

You can use the `onlySelf:false` to ensure that the parent FormGroup is not affected by our change.

```

  markCityAsDirty() {
    this.reactiveForm.get("address").get("city").markAsDirty({onlySelf:false});
  }
```

You can also make the entire FormGroup as dirty. It does not affect the child controls, but parent FormGroup is also marked as dirty. Unless you pass the {onlySelf:true} argument

```

markAddressAsDirty() {
  this.reactiveForm.get("address").markAsDirty({onlySelf:false});
}

```

## markAsPristine

The FormGroup becomes pristine when none of its controls values are changed via UI. The pristine is the opposite of dirty. We can use the markAsPrisitine method to manipulate the pristine status.

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

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

The following code marks the Form as Pristine. It will also mark all the child controls as Pristine

```

markFormAsPristine() {
  this.reactiveForm.markAsPristine();
}

```

The following code marks the city as Pristine. It will also calculate the Pristine status of the Parent FormGroup. If all the other controls are pristine then the parent FormGroup becomes pristine.

```

markCityAsPristine() {
  this.reactiveForm.get("address").get("city").markAsPristine({onlySelf:false});
}

```

You can make use of the `onlySelf:true` to ensure that the pristine status of the parent group is not calculated.

## markAsPending

Marks the control as pending. We usually use this when we running our validation checks. Pending means the status of the control cannot be determined at this time.

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

- `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.

The following code marks the entire form as Pending. It does not change the status of child Controls.

```

this.reactiveForm.markAsPending();

```

The following will mark the address FormGroup as Pending. It will also mark the Parent FormGroup as Pending also, which you can control using the `onlySelf:true` argument

```

markAddressAsPendng() {
  this.reactiveForm.get("address").markAsPending();
}

```

This method also triggers the statusChange Event. You can make use of `emitEvent:false` argument, which will stop the `statusChange` event being triggered.

```

markAddressAsPendng() {
  this.reactiveForm.get("address").markAsPending({emitEvent:false});
}


```

## disable

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.

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

- `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.

The following code disables all the controls in the FormGroup.

```

disableAll() {
  this.reactiveForm.disable();
}

```

If you disable all the controls individually, then the FormGroup is automatically disabled.

```

disableAll() {
  this.reactiveForm.get("firstname").disable();
  this.reactiveForm.get("lastname").disable();
  this.reactiveForm.get("email").disable();
  this.reactiveForm.get("address").disable();
}

```

### enable

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.

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

- 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.

The following command enables all the controls in the Group. Even the controls previously disabled are also enabled.

```

enableAll() {
   this.reactiveForm.enable();
}

```

Enables only address FormGroup.

```

enableAddress() {
  this.reactiveForm.get("address").enable();
}

```

Enable a Single Control

```

enableFirstName() {
  this.reactiveForm.get("firstname").enable();
}

```

## Status Change Event

### statusChanges

```
statusChanges: Observable<any>
```

The statusChanges event is fired whenever the status of the form is calculated. We can subscribe to this event as shown below. We can subscribe it at the FormControl level or at the FormGroup level.

**Note that this event is fired whenever the status is calculated.**

In the example below, the first statusChanges is emitted, when the status of the top-level FormGroup is calculated. The second statusChange event is emitted, when the address FormGroup status is calculated.

```

this.reactiveForm.statusChanges.subscribe(x => {
  console.log(x);
})

this.reactiveForm.get("address").statusChanges.subscribe(x => {
   console.log(x);
})

```

## Validation

The validators can be added to FormControl, FormGroup or to the FormArray.

### updateValueAndValidity()

The updateValueAndValidity forces the form to perform validation. When applied to the FormGroup, it will calculate the validity of all the child controls, including nested form groups & form arrays This is useful when you add/remove validators dynamically using `setValidators`, `RemoveValidators` etc

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

- `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.

```

this.reactiveForm.updateValueAndValidity();
this.reactiveForm.get("address").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

```

setValidator() {
  this.reactiveForm.get("address").setValidators([addressValidator]);
  this.reactiveForm.get("address").updateValueAndValidity();
}


export const addressValidator = (control: AbstractControl): {[key: string]: boolean} => {
  const city = control.get('city').value;
  const state = control.get('state').value;
  console.log(control.value);
  if (city=="" && state=="") {
    return { address:false };
  }
  return null;
};

```

## clearValidators() / clearAsyncValidators()

clearValidators(): void

clearAsyncValidators(): void

clearValidators & clearAsyncValidators clears all validators.

```

//reactive forms
clearValidation() {
   this.reactiveForm.get("address").clearValidators();
   this.reactiveForm.get("address").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.errors;
  if (controlErrors) {
    Object.keys(controlErrors).forEach(keyError => {
      console.log("firtname "+ ' '+keyError);
    });
  }
}

```

## setErrors()

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

```

setErrors() {
  this.reactiveForm.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")

//
```