Strictly Typed Forms in Angular

Leave a Comment / 10 minutes of reading / April 14, 2024

← Angular Tutorial Angular Forms →

Strictly Typed Forms in Angular provide a simple yet effective way to create type safe forms. Angular Forms prior to version 14 were given the type any. Which meant that the accessing properties which did not exist or assigning invalid value to a Form field, etc never resulted in a compile time error. The Strictly Typed forms now can track these errors at compile time and also has the benefit of auto completion, intellisense etc. In this guide we will explore the Typed Forms in Angular in detail.

Table of Contents

Strictly Typed Angular Forms

In the Reactive Forms, we build a Form Model using FormGroup , FormRecord, FormControl and FormArray . A FormControl encapsulates the state of a ***single form element*** (for example a input field), while FormGroup, FormRecord & FormArray help us to create a nested Form Model.

Prior to version 14, Angular assigned the type any to the Form Model & its controls. This allowed us to write a invalid code like assigning a string to a number or accessing a control which did not exists. Compiler never caught these errors.

Consider the following form in any version of Angular prior to 14.

```
1
2  export class AppComponent {
3   form: FormGroup;
4
5   constructor(private fb: FormBuilder) {}
6
7   ngOnInit() {
8     this.form = this.fb.group({
9       name: [''],
10      address: [''],
11      salary: [0],
12    });
13 );
14
15
```

Source Code

The code below always compiles without any issues, although we did not have `state` `FormControl` in our Angular Form. But it results in a run time error.

```
this.form.controls.state.value;

this.form.controls['state'].value;

```

Also, we could set string value to a numeric field like `salary`. Again compiler never warns us.

```
    this.form.controls.salary.setValue('Thousand Dollars');
    console.log(this.form.controls.salary.value);

```

By using the strictly typed forms, we can catch these errors at the time of compilation saving us the headache later.

Typed FormControl

There is no change in how you create forms under the typed forms in Angular. But before diving into creating the Typed Forms, let us learn how to create a Typed `FormControl`.

`FormControl` sets and tracks the individual HTML form element. We create it with the following code.

```
firstName = new FormControl('');

```

The `firstName` control automatically inferred as `FormControl<string | null>`. (In older versions inferred as `FormControl<any>`). We can now assign a **string or null** to it. Any other assignments will result in error.

Although we have initialized the control with a string value, **the inferred type also includes null**. This is because of the `reset` method which can set value of the control to `null`. Hence `null` is included in the type.

```
this.firstName.setValue('');
this.firstName.setValue(null);

this.firstName.setValue('Bill');
console.log(this.firstName.value); //Bill

this.firstName.reset(); //Defaults to null
console.log(this.firstName.value); //null
```

Source Code

Since `firstName` is `string | null`, assigning a number results in error.

```
//Results in compiler error
this.firstName.setValue(0);
```

To create the non nullable control, we must include the `nonNullable` flag and set it to `true`. We also need to provide a **non null initial value**. The code below creates a non nullable `FormControl` `lastName` with the initial value `Bill`.

```
lastName = new FormControl('Bill', { nonNullable: true });
```

Now, assigning `null` will result in error.

```
    //Error
    //this.lastName.setValue(null);
    //this.lastName.setValue(0);
```

Resetting the control will sets its value to **initial value (“bill”)** and not to `null`.

```
//Ok
this.lastName.setValue('');
this.lastName.setValue('Tom');
console.log(this.lastName.value); //Tom

this.lastName.reset(); //Defaults to Initial Value
console.log(this.lastName.value); //Bill
```

Declaring the control, without any initial value, will set its type to `FormControl<any>`. This is essentially means that you are opting out of type checking and can do anything with that control.

```
middleName = new FormControl();
```

Another interesting thing to note that, when you set the initial value to `null`, angular infers the type as `FormControl<null>`. You can now only assign `null` to it. Assigning any other value will result in compile error.

```
address= new FormControl(null);          //address  FormControl<null>

//ok
```

```
this.address.setValue(null);

//error
//this.address.setValue("");
//this.address.setValue(0);

```

You want `null` as the initial value, then only way by which you can do is manually assign the type. The code below creates Form Control with type `FormControl<string | null>`

```

city = new FormControl<string | null>(null);

```

We assign the `FormControl<number>` type to `price` FormControl below. But Angular still assigns `FormControl<number | null>` type to it. Hence control accepts `null` as a valid value and calling `reset` on it would set its value to `null`

```

price = new FormControl<number>(0);

```

You can prevent it by setting the `nonNullable` flag to true.

```

rate = new FormControl<number>(0, { nonNullable: true });

```

Creating Typed Forms in Angular

The best and easiest way is to let Typescript automatically Infer type for you. We can do that just by initializing the form at the time of declaration with initial value for each `FormControl` .

```
export class Demo1Component implements OnInit {

  profileForm = this.fb.group({
    name: [''],
    salary: [0],
    address: this.fb.group({
      city: [''],
      state: [''],
    }),
  });

 constructor(private fb: FormBuilder) {
}

```

source code

Hovering over the `profileForm`, will show you that Angular has assigned type for each from control based on the initial value.

The inferred type of `profileForm` is as shown below.

```
FormGroup<{
   name: FormControl<string | null>;
   salary: FormControl<number | null>;
   address: FormGroup<{
```

```
    city: FormControl<string | null>;
    state: FormControl<string | null>;
  }>;

```

The `name` is given the type `FormControl<string | null>`, while `salary` is of `FormControl<number | null>`.

Now, we can assign a number to the `salary` field. But you cannot assign a string to `salary` or number to `name`. The code below result in compiler error.

```

  //No problem here
   this.profileForm.controls.salary.setValue(1000);
   this.profileForm.get('salary')!.setValue(1000);

   //error
   //Argument of type 'number' is not assignable to parameter of type 'string
   this.profileForm.controls.name.setValue(1000);

   this.profileForm.get('name')!.setValue(1000);

   this.profileForm.controls.salary.setValue('Thousand');

   this.profileForm.get('salary')!.setValue('thousand');

```

Trying to add a new control will also result in compile time error, because the shape of the `profileForm` is now fixed.

```

this.profileForm.addControl("designation",new FormControl(0))

```

If you wish to add controls dynamically, you can either use FormRecord or FormArray.

Return value of the Typed Form

The return value of the form is of Type `Partial`. The `Partial` is utility type that creates a new type from an existing type, by marking all the existing properties as optional. This is required as the value property **does not the return the value of the disabled controls**. For Example if you disable the `salary` control, then `value` property will not include it in the result.

```
1
2  let result = this.profileForm.value;
3
```

You can see the type by hovering the mouse over the result variable.

The `getRawValue` method returns all the values (including those disabled). The returned type does not include `Partial` type.

```
1
2  let resultRaw = this.profileForm.getRawValue();
3
```

Intellisense

Another important benefit of Typed Forms in the Intellisense and auto complete.

NonNullableFormBuilder

We have seen that to prevent assigning null value to individual controls, you need to use the `nonNullable: true` flag. But that is difficult if you have lot of controls.

To help in such situations, Angular have a introduced the `NonNullableFormBuilder`, which we can use it in place of FormBuilder API. This API will automatically sets the non nullable flag of all FormControl's.

```
1
2  export class Demo4Component {
3    //inline declaration. Let Angular infer type
4    profileForm = this.fb.group({
5      name: [''],
6      salary: [0],
7      address: this.fb.group({
8        city: [''],
9        state: [''],
10     }),
11   });
12
```

```
  constructor(private fb: NonNullableFormBuilder) {}

}

```

Source Code

The above code infers the type as follow. You can see that the `null` is not included in the individual types.

```

(property) Demo4Component.profileForm: FormGroup<{
    name: FormControl<string>;
    salary: FormControl<number>;
    address: FormGroup<{
        city: FormControl<string>;
        state: FormControl<string>;
    }>;
}>

```

```

  ngOnInit() {
    //Ok
    this.profileForm.controls.name.setValue('Bill');
    this.profileForm.controls.name.setValue('');

    //Compile Error.  You cannot assign null
    //this.profileForm.controls.name.setValue(null);
  }

```

Typed FormArray

You can create a typed FormArray as follow.

```

nameArray = new FormArray([new FormControl('Bill')]);

```

Source Code

The `nameArray` is now an array of `FormControl` with string value.

```

this.nameArray.push(new FormControl('Tom'));

```

Hence pushing a number `FormControl` will result in an error.

```

this.nameArray.push(new FormControl(0));

```

The code below creates a `FormArray` with a `FormGroup` consisting of `Address`, `City` & `State` Controls.

```

addressArray = new FormArray([
  new FormGroup({
    address: new FormControl(''),
    city: new FormControl(''),
    state: new FormControl(''),
  }),
]);

```

You can push these values.

```

  this.addressArray.push(
    new FormGroup({
      address: new FormControl('30563 Marisa Field Apt. 758'),
      city: new FormControl('West Yessenia'),
      state: new FormControl('Virginia'),
    })
  );

```

But adding controls without state will result in an error

```
//ERROR
this.addressArray.push(
  new FormGroup({
    address: new FormControl('30563 Marisa Field Apt. 758'),
    city: new FormControl('West Yessenia'),
  })
);
```

If you do not know the controls ahead of time, then declare the type as `any`.

```
untypedArray = new FormArray<any>([new FormControl('Bill')]);
```

Now, you can push any values to it.

```
this.untypedArray.push(new FormControl('Tom'));
this.untypedArray.push(new FormControl(0));
this.untypedArray.push(
  new FormGroup({
    address: new FormControl('30563 Marisa Field Apt. 758'),
    city: new FormControl('West Yessenia'),
    state: new FormControl('Virginia'),
  })
);
this.untypedArray.push(
  new FormGroup({
    address: new FormControl('30563 Marisa Field Apt. 758'),
    city: new FormControl('West Yessenia'),
  })
);
}
```

Typed FormRecord

Using the untyped `FormGroup`, we were able to add or remove a control at run time. But with the typed `FormGroup` this is not possible.

For Example, adding the new control designation to the `companyForm` will result in error.

```
companyForm = new FormGroup({
  company: new FormControl(''),
});

//Error
this.companyForm.addControl("designation",new FormControl(0))
```

This is where FormRecord comes handy. We can use it add controls dynamically.

The code below adds `designations` `FormRecord` to `companyForm`. It is of type `FormControl<string | null>`.

```
companyForm = new FormGroup({
  company: new FormControl(''),
  designations: new FormRecord<FormControl<string | null>>({}),
});
```

Source Code

Now you can add any number of controls to the `designations` as long it is of type `FormControl<string | null>`.

```
ngOnInit() {
  this.companyForm.controls.designations.addControl(
    'CEO',
```

```
    new FormControl('')
  );
  this.companyForm.controls.designations.addControl(
    'Admin',
    new FormControl('')
  );
  this.companyForm.controls.designations.addControl(
    'Manager',
    new FormControl('')
  );
 }

```

Visit FormRecord in Angular to learn more about FormRecord.

Various ways to Create Typed Forms in Angular

We have seen how typed forms work, let us see various ways to create them

Let Angular Infer the Type

The simplest way is to initialize the form at the time of declaration. The Angular will infer the type of the from the initialization.

```

export class Demo1Component implements OnInit {

 //inline declaration. Angular will the infer type
  profileForm = this.fb.group({
    name: [''],
    salary: [0],
    address: this.fb.group({
      city: [''],
      state: [''],
    }),
  });

```

Source Code

Create a Custom Type

We can also create a custom type and assign it to our form. The code below creates a custom type `IProfile`.

```
interface IProfile {
  name: FormControl<string | null>;
  salary: FormControl<number | null>;
  address: FormGroup<Iaddress>;
}

interface Iaddress {
  city: FormControl<string | null>;
  state: FormControl<string | null>;
}
```

Source Code

And use it declare the `profileForm`.

```
profileForm!: FormGroup<IProfile>;
```

Now, you can use ngOnInit to initialize the form.

```
  ngOnInit() {
    this.profileForm = new FormGroup({
      name: new FormControl(''),
      salary: new FormControl(0),
      address: new FormGroup({
        city: new FormControl(''),
        state: new FormControl(''),
      }),
    });
```

Initialize it in the Class Constructor

You can also initialize the form in the component constructor. But the better way is to initialize the form at the time of declaration and let angular do the rest.

```
profileForm;

constructor(private fb: FormBuilder) {

  this.profileForm = this.fb.group({
    name: [''],
    salary: [0],
    address: this.fb.group({
      city: [''],
      state: [''],
    }),

  //this.profileForm = new FormGroup({
  //  name: new FormControl(''),
  //  salary: new FormControl(0),
  //  address: new FormGroup({
  //    city: new FormControl(''),
  //    state: new FormControl(''),
  //  }),
  //});

  });

```

Source Code

Do not use ngOnInit to initialize the typed form

We used to declare the form with the type `FormGroup` and initialize it in the `ngOnInit` method. But that will create a untyped form.

For Example, code below create a `profileForm`, but does not initialize it. The Angular will infer the type as `FormGroup<any>`. The initialization code in ngOnInit will not have any effect on the type.

```

```

```
profileForm:FormGroup;

constructor(private fb: FormBuilder) {
 }

 ngOnInit() {
   this.profileForm = this.fb.group({
     name: [''],
     salary: [0],
     address: this.fb.group({
       city: [''],
       state: [''],
     }),
   });
 }

```

UntypedFormGroup, UntypedFormBuilder & UntypedFormControl

When the existing applications are migrated, all the existing `FormGroup` references (types and values) were converted to `UntypedFormGroup`. Similarly `FormBuilder`, `FormControl` & `FormArray` references are converted to `UntypedFormBuilder`, `UntypedFormControl` & `UntypedFormArray`.

The `UntypedFormGroup` is an alias for `FormGroup<any>`. Hence during the migration all forms are migrated to untyped versions.

You can incrementally enable the types by removing the `Untyped` from the declaration and moving the initialization logic to the declaration.

Summary

1. Angular Typed forms now can track the type errors at compile time and also has the benefit of auto completion, intellisense etc.