Article
Sergei Sarkisian · Jun 30, 2022 8m read

# What's new in Angular 14

Hi! My name is Sergei Sarkisian and I ’ m creating Angular frontend for more than 7 years working in InterSystems. As the Angular is very popular framework, our developers, customers and partners often choose it as part of the stack for their applications.

I would like to start series of articles which will cover different aspects of Angular: concepts, how-to, best practices, advanced topics and more. This series will target people who already familiar with Angular and wouldn ’ t cover basic concepts. As I ’ m in the process of building articles roadmap, I would like to begin with highlighting some important features in most recent Angular release.

## Strictly typed forms

This is, probably, the most wanted feature of Angular in recent couple of years. With Angular 14 developers now able to use all the strict types checking functionality of TypeScript with Angular Reactive Forms.

FormControl class is now generic and takes the type of the value it holds.

```
/* Before Angular 14 */
const untypedControl = new FormControl(true);
untypedControl.setValue(100); // value is set, no errors

// Now
const strictlyTypedControl = new FormControl<boolean>(true);
strictlyTypedControl.setValue(100); // you will receive the type checking error messa
ge here

// Also in Angular 14
const strictlyTypedControl = new FormControl(true);
strictlyTypedControl.setValue(100); // you will receive the type checking error messa
ge here
```

As you see, the first and the last examples are almost the same but have different results. This happens because in Angular 14 new FormControl class infer types from initial value developer provided. So if value true was provided, Angular sets type boolean ¦ null for this FormControl. Nullable value needed for .reset() method which nulls the values if no value provided.

An old, untyped FormControl class was converted to UntypedFormControl (the same is for UntypedFormGroup, UntypedFormArray and UntypedFormBuilder) which is practically an alias for FormControl<any>. If you are upgrading from previous version of Angular, all of your FormControl class mentions will be replaced with UntypedFormControl class by Angular CLI.

Untyped* classes are used with specific goals:

1. Make your app work absolutely the same as it was before the transition from previous version (remember that new FormControl will infer the type from initial value).
2. Make sure that all of FormControl<any> uses are intended. So you will need to change any

UntypedFormControl to FormControl<any> by yourself.
3. To provide developers more flexibility (we will cover this below)

Remember that if your initial value is null then you will need explicitly specify the FormControl type. Also, there is a bug in TypeScript which requires of doing the same if your initial value is false.

For the form Group you can also define the interface and just pass this interface as type for the FormGroup. In this case TypeScript will infer all the types inside the FormGroup.

```
interface LoginForm {
    email: FormControl<string>;
    password?: FormControl<string>;
}

const login = new FormGroup<LoginForm>({
    email: new FormControl('', {nonNullable: true}),
    password: new FormControl('', {nonNullable: true}),
});
```

FormBuilder ' s method .group() now has generic attribute which can accept your predefined interface as in example above where we manually created FormGroup:

```
interface LoginForm {
    email: FormControl<string>;
    password?: FormControl<string>;
}

const fb = new FormBuilder();
const login = fb.group<LoginForm>({
    email: '',
    password: '',
});
```

As our interface has only primitive nonnullable types, it can be simplified with new nonNullable FormBuilder property (which contains NonNullableFormBuilder class instance which can also be created directly):

```
const fb = new FormBuilder();
const login = fb.nonNullable.group({
    email: '',
    password: '',
});
```

Note that if you use nonNullable FormBuilder or if you set nonNullable option in FormControl, then when you call .reset() method it will use initial FormControl value as a reset value.

Also, it ' s very important to note that all the properties in this.form.value will be marked as optional. Like this:

```
const fb = new FormBuilder();
const login = fb.nonNullable.group({
    email: '',
    password: '',
});
```

```
// login.value
// {
//   email?: string;
//   password?: string;
// }
```

This happens because when you disable any FormControl inside the FormGroup, the value of this FormControl will be deleted from form.value

```
const fb = new FormBuilder();
const login = fb.nonNullable.group({
    email: '',
    password: '',
});

login.get('email').disable();
console.log(login.value);

// {
//   password: ''
// }
```

To get the whole form object you should use .getRawValue() method:

```
const fb = new FormBuilder();
const login = fb.nonNullable.group({
    email: '',
    password: '',
});

login.get('email').disable();
console.log(login.getRawValue());

// {
//   email: '',
//   password: ''
// }
```

Advantages of strictly typed forms:

1. Any property and method returning the FormControl / FormGroup values is now strictly typed. E.g. value, getRawValue(), valueChanges.
2. Any method of changing FormControl value is now type safe: setValue(), patchValue(), updateValue()
3. FormControls are now strictly typed. It also applies to .get() method of FormGroup. This will also prevent you from accessing the FormControls that don ’ t exist at compile time.

## New FormRecord class

The downside of a new FormGroup class is that it lost it ’ s dynamic nature. Ones defined, you won ’ t be able to add or remove FormControls from it on the fly.

To address this problem Angular presents new class – FormRecord. FormRecord is practically the same as

FormGroup, but it ’ s dynamic and all of it ’ s FormControls should have the same type.

```
folders: new FormRecord({
  home: new FormControl(true, { nonNullable: true }),
  music: new FormControl(false, { nonNullable: true })
});

// Add new FormContol to the group
this.foldersForm.get('folders').addControl('videos', new FormControl(false, { nonNull
able: true }));

// This will throw compilation error as control has different type
this.foldersForm.get('folders').addControl('books', new FormControl('Some string', {
nonNullable: true }));
```

And as you see, this has another limitation – all the FormControls must be of the same type. If you really need both dynamic and heterogenous FormGroup, you should use UntypedFormGroup class to define your form.

## Moduleless (standalone) components

This feature still marked as experimental, but it is interesting one. It allows you to define components, directives and pipes without including them in any module.

The concept isn ’ t fully ready yet, but we are already able to build an application without ngModules.

To define a standalone component you need to use new standalone property in Component/Pipe/Directive decorator:

```
@Component({
  selector: 'app-table',
  standalone: true,
  templateUrl: './table.component.html'
})
export class TableComponent {
}
```

In this case this component can ’ t be declared in any NgModule. But it can be imported in NgModules and in other standalone components.

Each standalone component/pipe/directive now have mechanism to import it ’ s dependencies directly in the decorator:

```
@Component({
  standalone: true,
  selector: 'photo-gallery',
  // an existing module is imported directly into a standalone component
  // CommonModule imported directly to use standard Angular directives like *ngIf
  // the standalone component declared above also imported directly
  imports: [CommonModule, MatButtonModule, TableComponent],
  template: `
    ...
    <button mat-button>Next Page</button>
    <app-table *ngIf="expression"></app-table>
  `,
})
export class PhotoGalleryComponent {
}
```

As I mentioned above, you can import standalone components in any existing ngModule. No more imports of entire sharedModule, we can import only things we really need. This is also a good strategy to start using new standalone components:

```
@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, HttpClientModule, TableComponent], // import our standalon
e TableComponent
  bootstrap: [AppComponent]
})
export class AppModule {}
```

You can create standalone component with Angular CLI by typing:

```
ng g component --standalone user
```

## Bootstrap moduleless application

If you want to get rid of all the ngModules in your application, you will need to bootstrap your app differently. Angular has new function for that which you need to call in main.ts file:

```
bootstrapApplication(AppComponent);
```

The second parameter of this function will allow you to define the providers you need across your app. As most of the providers are usually exist in the modules, Angular (for now) requires to use a new importProvidersFrom extraction function for them:

```
bootstrapApplication(AppComponent, { providers: [importProvidersFrom(HttpClientModule
)] });
```

## Lazy load standalone component route:

Angular has new lazy-loading route function loadComponent which exists exactly for loading standalone components:

```
{
  path: 'home',
  loadComponent: () => import('./home/home.component').then(m => m.HomeComponent)
}
```

loadChildren now doesn ’ t only allow you to lazy load ngModule, but also to load children routes directly from the routes file:

```
{
  path: 'home',
  loadChildren: () => import('./home/home.routes').then(c => c.HomeRoutes)
}
```

## Some notes at the moment of article writing

- The standalone components feature is still in experimental stage. It will get much better in the future with moving to Vite builder instead of Webpack, better tooling, faster build times, more robust app architecture, easier testing and more. But now many of this things are missing, so we didn ’ t receive the whole package, but at least we can start to develop our Apps with new Angular paradigm in mind.
- IDEs and Angular tools are not fully ready yet to statically analyze new standalone entities. As you need to import all the dependencies in each standalone entity, in case if you miss something, the compiler can miss it also and fail you in the runtime. This will improve with time, but now it requires more attention to imports from developers.
- There are no global imports presented in Angular at the moment (as it done in Vue, for example), so you need to import absolutely each dependency in every standalone entity. I hope that it will be solved in future version as the main goal of this feature as I see is to reduce the boilerplate and make things easier.

That ’ s all for today.
See you!

#Angular #Angular2 #Frontend #UI Development #Other

**Source URL:**https://community.intersystems.com/post/whats-new-angular-14