

Artigo

[Danusa Calixto](#) · Set. 7, 2022 8min de leitura

## Novidades no Angular 14

Olá! Meu nome é Sergei Sarkisian e crio o front-end do Angular há mais de 7 anos trabalhando na InterSystems. Como o Angular é um framework bastante popular, ele é geralmente escolhido pelos nossos desenvolvedores, clientes e parceiros como parte da pilha para seus aplicativos.

Quero começar uma série de artigos sobre diferentes aspectos do Angular: conceitos, instruções, práticas recomendadas, tópicos avançados e muito mais. Essa série será destinada às pessoas que já estão familiarizadas com o Angular e não abordará conceitos básicos. Como ainda estou no processo de planejamento dos artigos, queria começar destacando alguns recursos importantes da versão mais recente do Angular.

### Formulários com tipos estritos

Provavelmente, esse é um dos recursos mais desejados do Angular nos últimos anos. Com o Angular 14, os desenvolvedores agora podem usar toda a funcionalidade de verificação de tipos estritos do TypeScript com os formulários reativos do Angular.

A classe FormControl é agora genérica e assume o tipo do valor que detém.

```
/* Antes do Angular 14 */
const untypedControl = new FormControl(true);
untypedControl.setValue(100); // o valor está definido, sem erros

// Agora
const strictlyTypedControl = new FormControl<boolean>(true);
strictlyTypedControl.setValue(100); // você receberá a mensagem de erro de verificação
o de tipo aqui

// Também no Angular 14
const strictlyTypedControl = new FormControl(true);
strictlyTypedControl.setValue(100); // você receberá a mensagem de erro de verificação
o de tipo aqui
```

Como você pode ver, o primeiro e o último exemplos são quase iguais, mas têm resultados diferentes. Isso ocorre porque, no Angular 14, a nova classe FormControl deduz tipos do valor inicial informado pelo desenvolvedor. Portanto, se o valor true foi fornecido, o Angular define o tipo boolean | null para FormControl. O valor anulável é necessário para o método .reset(), que anula os valores se nenhum for fornecido.

Uma classe FormControl antiga e sem tipo foi convertida para UntypedFormControl (isso também se aplica para UntypedFormGroup, UntypedFormArray e UntypedFormBuilder), que é basicamente um codinome para FormControl<any>. Se você estiver fazendo upgrade de uma versão anterior do Angular, todas as menções à classe FormControl serão substituídas pela classe UntypedFormControl pela CLI do Angular.

As classes sem tipo\* são usadas com metas específicas:

1. Fazer seu app funcionar da mesma maneira como era antes da transição da versão anterior (lembre-se de que o novo FormControl deduzirá o tipo a partir do valor inicial).

2. Verificar se todos os usos de `FormControl<any>` são desejados. Portanto, você precisará mudar qualquer `UntypedFormControl` para `FormControl<any>` por conta própria.
3. Dar aos desenvolvedores mais flexibilidade (abordaremos isso abaixo).

Lembra-se de que, se o valor inicial for `null`, você precisará especificar explicitamente o tipo `FormControl`. Além disso, o TypeScript tem um bug que exige que você faça o mesmo se o valor inicial for `false`.

Para o grupo do formulário, você também pode definir a interface e transmitir essa interface como um tipo para `FormGroup`. Nesse caso, TypeScript deduzirá todos os tipos dentro de `FormGroup`.

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

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

O método `.group()` do `FormBuilder` agora tem um atributo genérico que pode aceitar sua interface predefinida, como no exemplo acima, em que criamos manualmente o `FormGroup`:

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

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

Como nossa interface só tem tipos primitivos não anuláveis, ela pode ser simplificada com a nova propriedade `nullable` do `FormBuilder` (que contém a instância da classe `NullableFormBuilder`, também criada diretamente):

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

Se você usar o `FormBuilder nullable` ou definir a opção `nullable` no `FormControl`, quando chamar o método `.reset()`, ele usará o valor inicial do `FormControl` como um valor de redefinição.

Além disso, também é muito importante observar que todas as propriedades em `this.form.value` serão marcadas como opcionais. Desta forma:

```
const fb = new FormBuilder();
const login = fb.nullable.group({
```

```
    email: '',
    password: '',
  });

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

Isso ocorre porque, quando você desativa qualquer FormControl dentro do FormGroup, o valor desse FormControl será excluído do form.value

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

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

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

Para obter todo o objeto do formulário, você precisa usar o método `.getRawValue()`:

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

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

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

Vantagens de formulários com tipos estritos:

1. Qualquer propriedade e método que retorna valores do FormControl / FormGroup é agora estritamente tipado. Por exemplo: `value`, `getRawValue()`, `valueChanges`.
2. Qualquer método de mudança do valor do FormControl é agora seguro para os tipos: `setValue()`, `patchValue()`, `updateValue()`
3. Os FormControls têm agora tipos estritos. Isso também se aplica ao método `.get()` do FormGroup. Isso evitará que você acesse FormControls inexistentes no momento da compilação.

## Nova classe FormRecord

A desvantagem da nova classe `FormGroup` é que ela perdeu sua natureza dinâmica. Após a definição, você não poderá adicionar ou remover `FormControls` dela rapidamente.

Para resolver esse problema, o Angular apresenta uma nova classe — `FormRecord`. `FormRecord` é praticamente igual ao `FormGroup`, mas é dinâmico e todos os seus `FormControls` devem ter o mesmo tipo.

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

// Adicione o novo FormControl ao grupo
this.foldersForm.get('folders').addControl('videos', new FormControl(false, { nonNull
able: true }));

// Isso gerará um erro de compilação, já que o controle tem um tipo diferente
this.foldersForm.get('folders').addControl('books', new FormControl('Some string', {
nonNullable: true }));
```

Como você pode ver, há uma outra limitação — todos os `FormControls` precisam ter o mesmo tipo. Se você realmente precisar de um `FormGroup` dinâmico e heterogêneo, deverá usar a classe `UntypedFormGroup` para definir seu formulário.

## Componentes sem módulos (individuais)

Apesar de ainda ser considerado experimental, esse é um recurso interessante. Ele permite definir componentes, diretivas e pipes sem incluí-los em qualquer módulo.

O conceito ainda não está totalmente pronto, mas já conseguimos desenvolver um aplicativo sem `ngModules`.

Para definir um componente individual, você precisa usar a nova propriedade `standalone` no decorator `Component/Pipe/Directive`:

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

Nesse caso, o componente não pode ser declarado em qualquer `NgModule`. No entanto, ele pode ser importado em `NgModules` e outros componentes individuais.

Cada componente/pipe/diretiva individual agora tem um mecanismo para importar as dependências diretamente no decorator:

```
@Component({
  standalone: true,
  selector: 'photo-gallery',
  // um módulo existente é importado diretamente em um componente individual
  // CommonModule é importado diretamente para usar diretivas padrão do Angular como
  *ngIf
```

```
// o componente individual declarado acima também é importado diretamente
imports: [CommonModule, MatButtonModule, MatTableComponent],
template: `
  ...
  <button mat-button>Next Page</button>
  <app-table *ngIf="expression"></app-table>
`
})
export class PhotoGalleryComponent {
}
```

Como mencionei acima, você pode importar componentes individuais em qualquer NgModule existente. Não é mais necessário importar todo o SharedModule. Podemos importar somente o que é realmente necessário. Essa também é uma boa estratégia para começar a usar novos componentes individuais:

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

Você pode criar um componente individual com a CLI do Angular ao digitar:

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

### Aplicativo Bootstrap sem módulos

Se você quiser se livrar de todos os NgModules do seu aplicativo, você precisa usar o bootstrap de maneira diferente. O Angular tem uma nova função para isso que você precisa chamar no arquivo main.ts:

```
bootstrapApplication(AppComponent);
```

O segundo parâmetro dessa função permitirá definir os fornecedores necessários em todo o app. Como a maioria dos fornecedores geralmente existe em módulos, o Angular (por enquanto) exige o uso de uma nova função de extração importProvidersFrom para eles:

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

### Rota de componente individual de lazy load:

O Angular tem uma nova função loadComponent de rota de lazy loading, que serve exatamente para o carregamento de componentes individuais:

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

```
}
```

Agora, `loadChildren` não só permite o lazy load de `ngModule`, mas também carrega rotas filhas diretamente do arquivo de rotas:

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

### Algumas observações no momento da redação do artigo

- O recurso de componentes individuais ainda está em fase experimental. Ela receberá melhorias no futuro com a migração para Vite builder em vez de Webpack, ferramentas otimizadas, tempos de desenvolvimento mais rápidos, arquitetura de app mais robusta, testes mais fáceis e muito mais. Por enquanto, várias dessas coisas estão faltando, então não recebemos o pacote completo, mas pelo menos podemos começar a desenvolver nossos apps com o novo paradigma do Angular em mente.
- Os IDEs e ferramentas do Angular ainda não estão totalmente prontos para analisar estatisticamente novas entidades individuais. Já que é necessário importar todas as dependências em cada entidade individual, se você deixar algo passar, o compilador pode também não perceber e falhar no tempo de execução. Isso melhorará com o tempo, mas agora as importações exigem maior atenção dos desenvolvedores.
- Não há importações globais no Angular no momento (como em Vue, por exemplo), então você precisa importar cada uma das dependências em todas as entidades individuais. Espero que isso seja solucionado em uma versão futura, já que o principal objetivo desse recurso a meu ver seja reduzir o boilerplate e facilitar as coisas.

#

Isso é tudo por hoje. Até mais!

[#Angular](#) [#Angular2](#) [#Desenvolvimento de IU \(Interface do Usuário\)](#) [#Frontend](#) [#Outro](#)

---

URL de origem: <https://pt.community.intersystems.com/post/novidades-no-angular-14>