Article
Sergei Sarkisian · Juil 3, 2022 9m de lecture

Quoi de neuf dans Angular 14

Bonjour, je m'appelle Sergei Sarkisian et je crée des fronts Angular depuis plus de 7 ans en travaillant chez InterSystems. Comme Angular est un framework très populaire, nos développeurs, clients et partenaires le choisissent souvent comme partie de la pile pour leurs applications.

J'aimerais commencer une série d'articles qui couvriront différents aspects d'Angular : concepts, comment faire, meilleures pratiques, sujets avancés et plus encore. Cette série s'adressera aux personnes qui connaissent déjà Angular et ne couvrira pas les concepts de base. Comme je suis en train d'établir la feuille de route des articles, je voudrais commencer par mettre en évidence certaines fonctionnalités importantes de la dernière version d'Angular.

Formes strictement typées

Il s'agit probablement de la fonctionnalité d'Angular la plus demandée ces dernières années. Avec Angular 14, les développeurs peuvent désormais utiliser toutes les fonctionnalités de vérification stricte des types de TypeScript avec les formulaires réactifs d'Angular.

La classe FormControl est maintenant générique et prend le type de la valeur qu'elle contient.

```
/* Avant Angular 14 */
const untypedControl = new FormControl(true);
untypedControl.setValue(100); // est définie, aucune erreur

// Maintenant
const strictlyTypedControl = new FormControl<boolean>(true);
strictlyTypedControl.setValue(100); // vous recevrez le message d'erreur de vérificat
ion de type ici

// Également dans Angular 14
const strictlyTypedControl = new FormControl(true);
strictlyTypedControl.setValue(100); // vous recevrez le message d'erreur de vérificat
ion de type ici
```

Comme vous le voyez, le premier et le dernier exemple sont presque identiques, mais les résultats sont différents. Cela se produit parce que dans Angular 14 la nouvelle classe FormControl infère les types à partir de la valeur initiale fournie par le développeur. Ainsi, si la valeur true a été fournie, Angular définit le type boolean ¦ null pour ce FormControl. La valeur nullable est nécessaire pour la méthode .reset() qui annule les valeurs si aucune valeur n'est fournie.

Une ancienne classe FormControl non typée a été convertie en UntypedFormControl (il en est de même pour UntypedFormGroup, UntypedFormArray et UntypedFormBuilder) qui est pratiquement un alias pour FormControl<any>. Si vous effectuez une mise à jour depuis une version précédente d'Angular, toutes les mentions de votre classe FormControl seront remplacées par la classe UntypedFormControl par Angular CLI.

Les classes Untyped* sont utilisées avec des objectifs spécifiques :

1. Faire en sorte que votre application fonctionne absolument comme elle l'était avant la transition de la

version précédente (rappelez-vous que le nouveau FormControl déduira le type de la valeur initiale).
2. S'assurer que toutes les utilisations de FormControl<any> sont prévues. Donc vous devrez changer tout UntypedFormControl en FormControl<any> par vous-même.
3. Pour fournir aux développeurs plus de flexibilité (nous couvrirons ceci ci-dessous)

Rappelez-vous que si votre valeur initiale est null alors vous devrez explicitement spécifier le type de FormControl. Aussi, il y a un bug dans TypeScript qui exige de faire la même chose si votre valeur initiale est false.

Pour le groupe de formulaire, vous pouvez aussi définir l'interface et juste passer cette interface comme type pour le FormGroup. Dans ce cas, TypeScript déduira tous les types à l'intérieur du FormGroup.

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

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

La méthode de FormBuilder .group() a maintenant un attribut générique qui peut accepter votre interface prédéfinie comme dans l'exemple ci-dessus où nous avons créé manuellement FormGroup :

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

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

Comme notre interface n'a que des types primitifs non nuls, elle peut être simplifiée avec la nouvelle propriété nonNullable FormBuilder (qui contient l'instance de la classe NonNullableFormBuilder qui peut aussi être créée directement) :

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

Notez que si vous utilisez un FormBuilder non Nullable ou si vous définissez une option nonNullable dans le FormControl, alors lorsque vous appelez la méthode .reset(), elle utilisera la valeur initiale du FormControl comme valeur de réinitialisation.

Aussi, il est très important de noter que toutes les propriétés dans this.form.value seront marquées comme optionnelles. Comme ceci :

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

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

Cela se produit parce que lorsque vous désactivez un FormControl à l'intérieur du FormGroup, la valeur de ce FormControl sera supprimée de form.value.

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

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

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

Pour obtenir l'objet complet du formulaire, vous devez utiliser la méthode .getRawValue() :

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

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

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

Avantages des formulaires strictement dactylographiés :

1. Toute propriété et méthode retournant les valeurs du FormControl / FormGroup est maintenant strictement typée. Par exemple, value, getRawValue(), valueChanges.
2. Toute méthode permettant de modifier la valeur d'un FormControl est maintenant sécurisée par le type : setValue(), patchValue(), updateValue().
3. Les FormControls sont maintenant strictement typés. Cela s'applique également à la méthode .get() de FormGroup. Cela vous empêchera également d'accéder aux FormControls qui n'existent pas au moment de la compilation.

Nouvelle classe FormRecord

L'inconvénient de la nouvelle classe FormGroup est qu'elle a perdu sa nature dynamique. Une fois définie, vous ne serez pas en mesure d'ajouter ou de supprimer des FormControls à la volée.

Pour résoudre ce problème, Angular présente une nouvelle classe - FormRecord. FormRecord est pratiquement le même que FormGroup, mais il est dynamique et tous ses FormControls doivent avoir le même type.

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

// Ajouter un nouveau FormContol au groupe
this.foldersForm.get('folders').addControl('videos', new FormControl(false, { nonNull
able: true }));

// Cela entraînera une erreur de compilation car le contrôle est de type différent.
this.foldersForm.get('folders').addControl('books', new FormControl('Some string', {
nonNullable: true }));
```

Et comme vous le voyez, ceci a une autre limitation - tous les FormControls doivent être du même type. Si vous avez vraiment besoin d'un FormGroup à la fois dynamique et hétérogène, vous devriez utiliser la classe UntypedFormGroup pour définir votre formulaire.

Composants sans module (autonomes)

Cette fonctionnalité est toujours considérée comme expérimentale, mais elle est intéressante. Elle vous permet de définir des composants, des directives et des tuyaux sans les inclure dans un module.

Le concept n'est pas encore totalement au point, mais nous sommes déjà capables de construire une application sans ngModules.

Pour définir un composant autonome, vous devez utiliser la nouvelle propriété standalone dans le décorateur Composant/Pipe/Directive :

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

Dans ce cas, ce composant ne peut être déclaré dans aucun NgModule. Mais il peut être importé dans les NgModules et dans d'autres composants autonomes.

Chaque autonome composant/pipe/directive a maintenant un mécanisme pour importer ses dépendances directement dans le décorateur :

```
@Component({
  standalone: true,
  selector: 'photo-gallery',
```

```
  // un module existant est importé directement dans un composant autonome
    // CommonModule importé directement pour utiliser les directives Angular standard
 comme *ngIf
  // le composant autonome déclaré ci-dessus est également importé directement
  imports: [CommonModule, MatButtonModule, TableComponent],
  template: `
    ...
    <button mat-button>Next Page</button>
        <app-table *ngIf="expression"></app-table>
  `,
})
export class PhotoGalleryComponent {
}
```

Comme je l'ai mentionné ci-dessus, vous pouvez importer des composants autonomes dans n'importe quel ngModule existant. Il n'est plus nécessaire d'importer l'intégralité d'un module partagé, nous ne pouvons importer que les éléments dont nous avons réellement besoin. C'est également une bonne stratégie pour commencer à utiliser de nouveaux composants autonomes:

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

Vous pouvez créer un composant autonome avec Angular CLI en tapant:

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

Application Bootstrap sans module

Si vous voulez vous débarrasser de tous les ngModules dans votre application, vous devrez amorcer votre application différemment. Angular a une nouvelle fonction pour cela que vous devez appeler dans le fichier main.ts:

```
bootstrapApplication(AppComponent);
```

Le second paramètre de cette fonction vous permettra de définir les providers dont vous avez besoin dans votre application. Comme la plupart des fournisseurs existent généralement dans les modules, Angular (pour l'instant) exige d'utiliser une nouvelle fonction d'extraction importProvidersFrom pour eux :

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

Chargement paresseux de la route des composants autonomes:

Angular dispose d'une nouvelle fonction de route de chargement paresseux, loadComponent, qui existe exactement pour charger des composants autonomes:

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

loadChildren ne vous permet plus seulement de charger paresseusement un ngModule, mais aussi de charger les routes des enfants directement à partir du fichier des routes :

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

Quelques notes au moment de la rédaction de l'article

- La fonctionnalité des composants autonomes est encore au stade expérimental. Elle sera bien meilleure à l'avenir avec le passage à Vite builder au lieu de Webpack, un meilleur outillage, des temps de construction plus rapides, une architecture d'application plus robuste, des tests plus faciles et plus encore. Mais pour l'instant, beaucoup de ces éléments sont manquants, nous n'avons donc pas reçu le paquet complet, mais au moins nous pouvons commencer à développer nos applications avec le nouveau paradigme Angular en tête.
- Les IDE et les outils Angular ne sont pas encore tout à fait prêts à analyser statiquement les nouvelles entités autonomes. Comme vous devez importer toutes les dépendances dans chaque entité autonome, si vous manquez quelque chose, le compilateur peut aussi le manquer et vous faire échouer au moment de l'exécution. Cela s'améliorera avec le temps, mais pour l'instant, les développeurs doivent accorder plus d'attention aux importations.
- Il n'y a pas d'importations globales présentées dans Angular pour le moment (comme cela se fait dans Vue, par exemple), donc vous devez importer absolument chaque dépendance dans chaque entité autonome. J'espère que ce problème sera résolu dans une prochaine version, car l'objectif principal de cette fonctionnalité est de réduire le boilerplate et de rendre les choses plus faciles.

C'est tout pour aujourd'hui.
A bientôt !

#Angular #Angular2 #Développement de l'interface utilisateur #FrontEnd #Autre