Angular form validation formarray

I don't think it's entirely possible on the template. This is because to access the state of the FormArray control in the template, you must access this.formGroup.get('features').controls[i].invalid. But because get returns an instance of type AbstractControl, you will not have access to the array controls on it. To do this, you need to create a broadcast number regardless of what is returned from this.formGroup.get('features') to FormArray. I really don't think it would be possible on the template. You must create a method in the class that will return the validity of the control based on the index. So if you still refer to stackblitz eg, here's how: &lt;form [formgroup]=formGroup&gt;&lt;div formarrayname=features&gt;&lt;div class=row no-gutters form-group *ngfor=let feature of features.controls; let index = index; let last = last&gt;&lt;input type=text class=form-control px-2 col [formcontrolname]=index title=feature required=&gt; Invalid IS ID - {{ getValidity(index) }} &lt;div class=col-3 col-md-2 row no-gutters&gt;&lt;button class=col btn btn-outline-danger (click)=removeFeature(index)&gt; - &lt;/button&gt; &lt;button class=col btn btn-success *ngif=last (click)=addFeature()&gt; + &lt;/button&gt;&lt;/div&gt;&lt;/div&gt;&lt;/div&gt;&lt;/form&gt; A in its class: import { Component } from '@angular/core'; import { FormArray, FormBuilder, FormControl, Validators } from '@angular/forms'; @Component({ selector: 'my-app', templateUrl: './app.component.html', styleUrls: ['./app.component.css'] }) export class AppComponent { constructor(private fb: FormBuilder,) {} formGroup = this.fb.group({ features: this.fb.array([this.fb.control. Validators.required])}); get features(): FormArray { return this.formGroup.get('features') as FormArray; } addFeature(): void { this.features.push(this.fb.control('', Validators.required)); } getValidity(i) { return (&lt;FormArray&gt;this.formGroup.get('features')).controls[i].invalid; } removeFeature(index): void { this.features.removeAt(index); console.log(this.features); } UPDATE A few months ago, I realized that calling a method in one of the data binding syntax (i.e. Interpolating strings - {{ ... }}, binding a property - [propertyName]=methodName(), or binding an attribute - [class.class-name]=methodName() OR [style.styleName]=methodName()) is extremely expensive in terms of performance. Therefore, you should do this by using{ {{ formGroup.controls['features'].controls[index].invalid }} Instead of: {{ getValidity(index) }} Here's an updated working stackblitz for ref. If you want to learn more about this topic, I highly recommend that you check out this thread: Angular Performance: DOM Event causes unnecessary calls to Hope to help :) On the Web, some of the earliest elements of user input were a button, check box, text input, and radio buttons. To this day, these elements are still used in modern web applications, even though the HTML standard a long way from the early definition and now allows all kinds of fantasies &lt;/FormArray&gt; &lt;/FormArray&gt; Validating user input is an essential part of any reliable web application. Forms in angular applications can aggregate the state of all inputs that are in this form and provide a general state, such as the validation status of the full form. This can be really useful to decide whether user input will be accepted or rejected without checking each input separately. In this article, you'll learn how to work with forms and easily validate a form in an angular application. Angular 4 offers two different types of forms to work with: template-based and reactive forms. We'll go through each form type using the same example to see how the same things can be implemented in different ways. Below, the next part of the article We will look at an innovative approach to setting up and working with nested forms. Angular 4 Forms In Angular 4, the following four states are commonly used by forms: valid – severity status of all form controls, true if all controls are invalid – inverse of validity; true, if some control is incorrect virgin - gives status about the purity of the form; true, if no control has been modified dirty - the inverse virgin; true if some control has been modified Note a look at the basic form example: &lt;form&gt;&lt;div&gt;&lt;label&gt;Name&lt;/label&gt;&lt;input type=text name=name&gt;&lt;/div&gt;&lt;div&gt;&lt;label&gt;Year of birth&lt;/label&gt;&lt;input type=text name=birthYear&gt;&lt;/div&gt;&lt;div&gt;&lt;h3&gt;Location&lt;/h3&gt;&lt;div&gt;&lt;label&gt;Country&lt;/label&gt;&lt;input type=text name=country&gt;&lt;/div&gt;&lt;div&gt;&lt;label&gt;City&lt;/label&gt;&lt;input type=text name=city&gt;&lt;/div&gt;&lt;/div&gt;&lt;div&gt;&lt;h3&gt;Phone numbers&lt;/h &gt;&lt;div&gt;&lt;label&gt;Phone number 1&lt;/label&gt;&lt;input type=text name=phoneNumber[1]&gt;&lt;button type=button&gt;delete&lt;/button&gt;&lt;/div&gt;&lt;button type=button&gt;Add phone number&lt;/button&gt;&lt;/div&gt;&lt;button type=submit&gt;Register&lt;/button&gt; &lt;button type=button&gt;print out on console&lt;/button&gt;&lt;/form&gt; The specification for this example is as follows: the name - is required and unique among all registered birth usersYear - should be a valid number and the user must be at least 18 and less than 85 years old country - is mandatory, and only to make things a little complicated, we need validation, that if the country is France, then the city must be Paris (let's say our service is offered only in Paris) phoneNumber - each phone number must follow a certain pattern, there must be at least one phone number, and the user can add a new or remove an existing phone number. Button is enabled only if all input is correct and submits the form when clicked. Print to console when clicked prints the value of all inputs on the console. The ultimate goal is to fully implement the defined specification. Template-based forms based on templates are very similar to forms in angularjs (or angular 1, as some refer to it). Thus, someone who has worked with forms at AngularJS will be very familiar with this approach to working with forms. With the introduction of the in Angular 4, it is enforced that each specific form type is in a separate module, and we must explicitly determine which type we will use by importing the appropriate module. This module for template-based forms is FormsModule. With this in mind, you can activate template-based forms as follows: import {FormsModule} from @angular {NgModule} imported @angular {BrowserModule} from {AppComponent @angular import from 'src/app.component'; @NgModule({ imports: [BrowserModule, FormsModule], declarations: [AppComponent], bootstrap: [AppComponent] }) export class AppModule {} As shown in this snippet, we first need to import the browser module because it provides the services necessary to start and run the browser application. (from Angular 4). We then import the required FormsModules to activate template-based forms. And the last is the declaration of the root component, AppComponent, where in the next steps we will implement the form. Note that in this example and the following examples, you must ensure that the application is properly boottrapped by using the BrownerDynamic method. import {platformBrowserDynamic} from '@angular/platform-browser-dynamic'; import {AppModule} from './app.module'; PlatformBrowserDynamic().bootstrapModule(AppModule); We can assume that our AppComponent (app.component.ts) looks something like this: import {Component} from @angular @Component/core'@Component({ selector: 'my-app', templateUrl: 'src/app.component.tpl.html' }) export class AppComponent { } Where the template of this component is in app.component.tpl.html and we can copy the initial template to this file. Note that each input must have a name attribute that you want to correctly identify in the form. Although this seems like a simple form of HTML, we've defined an already supported Angular 4 form (you may not see it yet). When you import FormsModule, Angular 4 automatically detects the form's HTML element and attaches the NgForm component to that element (via the NgForm component selector). This is the case in our example. Although this Angular form 4 is declared, at this point it does not know of any angular 4 supported inputs. Angular 4 is not intrusive to register each input HTML element to the nearest parent element of the form. The key that allows the input element to be noticed as an Angular element 4 and registered in the NgForm component is the NgModel directive. So we can extend the template app.component.tpl.html as follows: &lt;form&gt; .. &lt;input type=text name=name ngmodel=&gt; .. &lt;input type=text name=birthYear ngmodel=&gt; .. &lt;input type=text name=country ngmodel/=&gt; .. &lt;input type=text name=city ngmodel/=&gt; .. &lt;input type=text name=phoneNumber[1] ngmodel/=&gt;&lt;/form&gt; Adding the NgModel directive, all input is recorded in the component Thanks to this, we have defined the fully functioning form of Angular 4 and so far, so good, but we still do not have access The NgForm component and the features it offers. The two main functions offered by NgForm are: Getting the values of all registered input controls Get the general state of all controls To expose NgForm, we can add the following elements to &lt;form&gt; Element: &lt;form #myform=ngForm&gt; .. &lt;/form&gt; This is possible thanks to exportpower of the decorator component. After doing this, we can access the values of all input controls and extend the template to: &lt;form #myform=ngForm&gt; .. &lt;pre&gt;{{myForm.value | json}}&lt;/pre&gt;&lt;/form&gt; With myForm.value, we access JSON data containing the values of all registered inputs, and with {{myForm.value | json}}, we are a nice JSON print with values. What if we want to have a subgroup of inputs from a specific context wrapped in a container and a separate object in JSON values, such as a location that contains a country and city, or phone numbers? Don't stress – template-based forms in angular 4 also take this into account. To achieve this, use the ngModelGroup directive. &lt;form #myform=ngForm&gt; .. &lt;div ngmodelgroup=location&gt; .. &lt;/div&gt;.. &lt;div&gt; .. &lt;/div&gt;&lt;/form&gt; What we're missing now is a way to add multiple phone numbers. The best way to do this would be to use an array as the best iterable representation of a multi-object container, but at the time of writing, this feature is not implemented for template-based forms. So we need to use a workaround to make it work. The phone numbers section must be updated as follows: &lt;div ngmodelgroup=phoneNumbers&gt;&lt;h3&gt;Phone Numbers&lt;/h3&gt;&lt;div *ngfor=let phoneId of phoneNumberIds; let i=index;&gt;&lt;label&gt;Phone Number {{i + 1}}&lt;/label&gt;&lt;input type=text name=phoneNumber[{{phoneId}}] #phonenumber=ngModel ngmodel/=&gt;&lt;button type=button (click)=remove(i); myForm.control.markAsTouched()&gt;delete&lt;/button&gt;&lt;/div&gt;&lt;button type=button (click)=add(); myForm.control.markAsTouched()&gt;Add phone number&lt;/button&gt;&lt;/div&gt; MyForm.control.markAsTouched() is used to form touched so we can display errors at this point. Buttons do not activate this property when clicked, only input. To make the next examples clearer, I won't add this line on the click handler for add() and remove(). Imagine being there. (He is present in Plunkers.) We also need to update AppComponent to include the following code: private number:number =1; phoneNumberIds:number[] = [1]; remove(i:number) { this.phoneNumberIds.splice(i, 1); } add() { this.phoneNumberIds.push(++this.count); } add() { this.phoneNumberIds.push(++this.count); } add() { this.phoneNumberIds.push(++this.count); } add() { this.phoneNumberIds.push(++this.count); } add() { } We need to store a unique id for each new phone number added, a w*ngFor, track phone number control by their ID (I admit it's not very nice, but as long as the Angular 4 team implements this feature, I'm afraid it's the best we can do) Okay, what we've got so far, we've added an Angular 4 supported form with inputs, added a specific grouping of inputs (location and &lt;/form&gt; &lt;/form&gt; numbers) and unveiled the form in the template. But what if we would like to access the NgForm object in some method in the component? We'll look at two ways to act. In the first case, NgForm, labeled myForm in the current example, can be passed as an argument to a function that will serve as a handler for the onSubmit event form. For better integration, the onSubmit event is wrapped by Angular 4, a NgForm-specific event called ngSubmit, and this is the right way to go if we want to perform some actions on the submit. So, the example will now look like this: &lt;form #myform=ngForm (ngsubmit)=register(myForm)&gt; ... &lt;/form&gt; We must have an appropriate method registry implemented in AppComponent. Something like: sign up (myForm: NgForm) { console.log('Successful registration'); console.log(myForm); } This way, using the onSubmit event, we only have access to the NgForm component when the upload is performed. The second way is to use a view query by adding a decorator @ViewChild the properties of the component. @ViewChild(myForm) private myForm: NgForm; With this approach, we have access to the form regardless of whether the onSubmit event was fired or not. Now we have a fully functioning Angular 4 form with access to the form in the component. But do you notice something is missing? What if a user enters something like this-is-not-a-year in years entry? Yes, you have it, we are missing validation of the input and we will discuss it in the following section. Validation is really important for any application. We always want to verify your input (we can't trust you) to prevent incorrect data from being sent/saved, and we need to show you some meaningful error message to properly guide you to enter the correct data. In order for some validation rules to be enforced on some input, a validator must be associated with this input. Angular 4 already offers a set of common validators such as: required, maxLength, minLength ... So how can we associate the validator with the input? Well, quite simple; just add the validator directive to the control: &lt;input name=name ngmodel= required/=&gt; In this example, the name input is mandatory. Let's add some validation to all the input in our example. &lt;form #myform=ngForm (ngsubmit)=actionOnSubmit(myForm) novalidate=&gt;&lt;p&gt;Is myForm valid? {{myformat.valid}}&lt;/p&gt; .. &lt;input type=text name=name ngmodel= required/=&gt; .. &lt;input type=text name=birthYear ngmodel= required= pattern=\\d{4.4}&gt; .. &lt;div ngmodelgroup=location&gt; .. &lt;input type=text name=country ngmodel= required/=&gt; .. &lt;input type=text name=city ngmodel/=&gt;&lt;/div&gt;&lt;div ngmodelgroup=phoneNumbers&gt; .. &lt;input type=text ngmodel= required/=&gt; .. &lt;/div&gt; .. &lt;/form&gt; Note: novalidate is used to disable verification of the native browser form. We made the name required, the summer field is required and must consist only of numbers, country input is required, as well as phone phone is required. In addition, we can print the validity status of the form by using {{myForm.valid}}. The improvement in this example may also show what is wrong with user input (not just show the overall state). Before we continue adding additional verification, I would like to implement a secondary component that will allow us to print all errors for the provided control. show-errors.component.ts import { Component, Input } from '@angular/core'; import { AbstractControlDirective, AbstractControl } from '@angular/forms'; @Component({ selector: 'show-errors', template: ' &lt;ul *ngif=shouldShowErrors()&gt;&lt;li style=color: red *ngfor=let error of listOfErrors()&gt;{{error}}&lt;/li&gt;&lt;/ul&gt;', }) export class ShowErrorsComponent { private static readonly errorMessages = { 'required': () =&gt; 'This field is required', 'minlength': (params) =&gt; 'Minimum number of characters is ' + params.requiredLength, 'maxlength': (params) =&gt; 'Maximum character allowed is ' + params.requiredLength, 'pattern': (params) =&gt; 'Required formula is: ' + params.requiredPattern, 'years': (params) =&gt; params.message, 'countryCity': (params) =&gt; params.message, 'uniqueName': (params) =&gt; params.message, 'phoneNumbers': (params) =&gt; params.message, 'phoneNumber': (params) =&gt; params.message }; @Input() private control: Abstract paraControlIve | AbstractControl; shouldShowErrors(): boolean { return this.control &amp;this.control.errors &amp;amp; (this.control.dirty || this.control.touched); } listOfErrors(): string[] { return Object.keys(this.control.errors) .map(field =&gt; this.getMessage(field, this.control.errors[field])); } private getMessage(type: string, params: any) { return ShowErrorsComponent.errorMessages[type](params); } } The error list is displayed only if there are some existing errors and the input is touched or dirty. The message for each error is searched on the map of predefined errorMessages (I added all the messages in advance). This component can be used as follows:&lt;div&gt;&lt;label&gt;Year of Birth&lt;/label&gt;&lt;input type=text name=birthYear #birthyear=ngModel ngmodel= required= pattern=\\d {4,4}&gt;&lt;show-errors [control]=birthYear&gt;&lt;/show-errors&gt;&lt;/div&gt; We need to make NgModel available for each input and pass it to the component, that renders all errors. You may notice that in this example, we used a pattern to check if the data is a number; what if a user enters 0000? This would be invalid input. In addition, we lack validators of a unique name, a strange country restriction (if the country =France, this city must be Paris), a formula for a valid phone number and confirmation that there is at least one phone number. This is the right time to take a look at custom validators. Angular 4 offers an interface that every custom validator must implement, Validator (what a surprise!). The validator interface essentially looks like this: export interface validator { validate(c: AbstractControl): ValidationErrors | null; null; () =&gt; void): void; } W przypadku gdy każda konkretna implementacja MUSI wdrożyć metodę walidacji. Ta metoda sprawdzania poprawności jest naprawdę interesujące, co można odbierać jako dane wejściowe i co powinno być zwracane jako dane wyjściowe. Dane wejściowe jest AbstractControl, co oznacza, że argument może być dowolny typ, który rozszerza AbstractControl (FormGroup, FormControl i FormArray). Dane wyjściowe metody sprawdzania poprawności powinny być null lub undefined (bez danych wyjściowych), jeśli dane wejściowe użytkownika jest prawidłowy lub zwraca validationerrors obiektu, jeśli dane wejściowe użytkownika jest nieprawidłowy. Z tą wiedzą, teraz wdrożymy niestandardowy birthYear validator. import { dyrektywy } z @angular/rdzeń; importu { NG_VALIDATORS, FormControl, Walidator, ValidationErrors } z '@angular/forms'; @Directive({ selector: '[birthYear]', dostawcy: [{provide: NG_VALIDATORS, useExisting: BirthYearValidatorDirective, multi: true}] }) klasa eksportu BirthYearValidatorDirective implementuje Walidator { validate(c: FormControl): ValidationErrors { const numValue = Number(c.value); const currentYear = new Date().getFullYear(); const minYear = currentYear - 85; const maxYear = currentYear - 18; const isValid = !isNaN(numValue) &amp;&amp; numValue &gt;= minYear &amp;amp; numValue &lt;= maxyear;= const= message={ 'years':= {= 'message':= 'the= year= must= be= a= valid= number= between= '= += minyear= += '= and= '= += maxyear= }= };= return= isvalid= null= := message;= }= }= there= are= a= few= things= to= explain= here.= first= you= may= notice= that= we= implemented= the= validator= interface.= the= validate= method= checks= if= the= user= is= between= 18= and= 85= years= old= by= the= birth= year= entered.= if= the= input= is= valid,= then= null= is= returned,= or= else= an= object= containing= the= validation= message= is= returned.= and= the= last= and= most= important= part= is= declaring= this= directive= as= a= validator.= that= is= done= in= the= "providers"= parameter= of= the= @directive= decorator.= this= validator= is= provided= as= one= value= of= the= multi-provider= ng_validators.= also,= don't= forget= to= declare= this= directive= in= the= ngmodule.= and= now= we= can= use= this= validator= as= following:=&gt;&lt;/=&gt;&lt;input type=text name=birthYear #year=ngModel ngmodel= required= birthyear/=&gt; Tak, tak proste! W przypadku numeru telefonu możemy zweryfikować format numeru telefonu w ten sposób: import { dyrektywa } z @angular/rdzeń; import { NG_VALIDATORS, Walidator, FormControl, ValidationErrors } z '@angular/forms'; @Directive({ selector: '[phoneNumber]', dostawcy: [{provide: NG_VALIDATORS, useExisting: TelephoneNumberFormatValidatorDirective, multi: true}] }) export class TelephoneNumberFormatValidatorDirective implementuje Walidator { validate(c: FormControl): ValidationErrors { const isValidPhoneNumber = /^\d{3,3}-\d{3,3}-\d{3,3}$/.test(c.value); const message = { 'phoneNumber': { 'message': 'The phone must be valid (XXX-XXX-XXX, where X is a digit)' } }; returns isValidPhoneNumber ? null : message; } } Now come two validations, for country and number of phone numbers. Notice something in common for both of them? Both require more than one to carry out an appropriate verification. Well, remember the Validator interface, and what did we say about it? The argument to the validation method is AbstractControl, which can be user input or the form itself. This provides the ability to implement a validator that uses multiple controls to determine a specific validation state. import { directives } from @angular/core; import { NG_VALIDATORS, Validator, Form Group, ValidationErrors } from '@angular/forms'; @Directive({ selector: '[countryCity]', providers: [{provide: NG_VALIDATORS, useExisting: CountryCityValidatorDirective, multi: true}] }) CountryCityValidatorDirective export class implements Validator { validate(form: FormGroup): ValidationErrors { const countryControl = form.get('location.country'); const cityControl = form.get('location.city'); if (countryControl != null &amp;amp; cityControl != null) { const country = countryControl.value; const city = cityControl.value; let error = null; if (country === 'France' &amp;amp; city !== 'Paris') { error = 'If the country is France, the city must be Paris'; } const message = { 'countryCity': { 'message': error } }; return error ? message : null; } } } } We have implemented a new validator, country-city validator. You notice that now as an argument the validation method receives FormGroup and from this FormGroup we can retrieve the input required for validation. The other things are very similar to a single input validator. The phone number validator will look like this: import { directive } from @angular/core; import { NG_VALIDATORS, Validator, Form Group, ValidationErrors, FormControl } from '@angular/forms'; @Directive({ selector: '[phoneNumbers]', provider: [{provide: NG_VALIDATORS, useExisting: TelephoneNumbersValidatorDirective, multi: true}] }) export class TelephoneNumbersValidatorDirective implements Validator { validate(form: FormGroup): ValidationErrors { const message = { 'phoneNumbers': { 'message': 'At least one phone number must be entered' } }; &lt;FormGroup&gt; form.get('phoneNumbers'); const hasPhoneNumbers = phoneNumbers &amp;amp; Object.keys(phoneNumbers.controls).length &gt; 0; back hasPhoneNumbers? null : message; } } We can use them this way: &lt;form #myform=ngForm countrycity= telephonenumbers=&gt; .. &lt;/form&gt; Same as input validators, right? Just now applied to the form. Do you remember the ShowErrors component? We implemented it to work with AbstractControlDirective, which means that we can reuse it to display all errors associated directly with this form. Note that at this point, the only directly associated validation rules with the form are city-country and phone numbers (other validation modules are associated with specific form controls). To print all form errors, do the following: #myform=ngForm countrycity= telephonenumbers=&gt;&lt;show-errors [control]=myForm&gt;&lt;/show-errors&gt; .. &lt;/form&gt; The last thing that remains is to validate the unique name. This is &lt;/FormGroup&gt; &lt;/FormGroup&gt; miscellaneous; to verify that the name is unique, it is most likely a back-end connection that is needed to check all existing names. This classifies as an asynchronous operation. To do this, we can reuse the previous technique for custom validators, just perform validation to return an object that will be resolved sometime in the future (promise or observable). In our case, we will use the promise: import { directive } from @angular/core; import { NG_ASYNC_VALIDATORS, Validator, FormControl, ValidationErrors } from '@angular/forms'; @Directive({ selector: '[uniqueName]', provider: [{provide: NG_ASYNC_VALIDATORS, useExisting: UniqueNameValidatorDirective, multi: true}] }) export class UniqueNameValidatorDirective implements Validator { validate(c: FormControl): ValidationErrors { const message = { 'uniqueName': { 'message': 'Message': 'Name is not unique' }; }; returns a new promise(resolve =&gt; { setTimeout(() =&gt; { resolve(c.value === 'Existing' ? message : null); }, 1000); }} } We wait for 1 second and then return the result. As with synchronization validation, if a promise is resolved with null, validation has been passed; If the promise is resolved with anything else, validation fails. Also note that now this validator is registered with another other provider that NG_ASYNC_VALIDATORS. One of the useful form properties for asynchronous validators is the pending property. It can be used as follows: &lt;button [disabled]=myForm.pending&gt;Register&lt;/button&gt; disables the button until the asynchronous checkers are resolved. Here's plunker containing the full AppComponent, ShowErrors component and all validators. In these examples, we've covered most of the cases of working with template-based forms. We've shown that template-based forms are very similar to forms in AngularJS (AngularJS developers will be able to migrate). In this type of form, it is quite easy to integrate Angular 4 forms with minimal programming, mainly with manipulations in the HTML template. Reactive forms reactive forms were also known as model-based forms, but I like to call them programmable forms, and soon you'll see why. Reactive forms are a new approach to handling angular 4 forms, so unlike template-based, AngularJS developers will not be familiar with this type. Can we start now, remember how template-based forms had a special module? Well, reactive forms also have their own module, called ReactiveFormsModule, and must be imported to activate these types of forms. import {ReactiveFormsModule} from import @angular/forms {NgModule} from import @angular/core {BrowserModule} from @angular/platform-browser import {AppComponent} from @NgModule({ imports: [BrowserModule, ReactiveFormsModule], declarations: [AppComponent], bootstrap: [AppComponent] }) export class AppModule {} In addition, do not forget to start the application. We can start with the same template, as in the previous section. At this point, if FormsModule is not imported (and make sure it isn't), we only have a plain HTML form element with several forms of form, no angular magic here. We come to the point where you can see why I like to call this approach programmable. To enable Angular 4 forms, you must manually declare the FormGroup object and fill it with controls like this: import { FormGroup, FormControl, FormArray, NgForm } from @angular/forms; import { Component, OnInit } from @angular/core; @Component({ selector: 'my-app', templateUrl: 'src/app.component.html' }) export class AppComponent implements OnInit { private myForm: FormGroup; constructor() { } ngOnInit() { this.myForm = new FormGroup({ 'name': new FormControl(), 'birthYear': new FormControl(), 'location': new FormGroup({ 'country': new FormControl(), 'city': new FormControl() }), 'phoneNu member count'new FormArray([new Form Control('')]) }); } printMyForm() { console.log(this.myForm); } register(myForm: NgForm) { console.log('Registration succeeded.'); console.log(myForm.value); } } } The PrintForm and register methods are the same from the previous examples and will be used in the next steps. The key types used here are FormGroup, FormControl, and FormArray. These three types are all that we need to create a valid FormGroup. FormGroup is easy; it is a simple container of controls. FormControl is also easy; this is any control (e.g. input). And finally, FormArray is part of a puzzle that we lacked in the template approach. FormArray allows you to maintain a control group without specifying a specific key for each control, basically a control board (seems like the perfect thing for phone numbers, right?). When constructing any of these three types, be aware of this 3rd rule. The constructor for each type receives three arguments — a value, a validator, or a list of validators and an asynchronizer, or a list of asynchronous validators defined in the code: constructor(value: arbitrary, validator?: ValidatorFn | ValidatorFn[], asyncValidator?: AsyncValidatorFn | AsyncValidatorFn[]); For FormGroup, a value is an object in which each key represents the name of the control, and the value is the control itself. In the case of FormArray, the value is an array of controls. For FormControl, the value is the initial value or initial state (the object containing the value and the disabled property) of the control. We created the FormGroup Object, but the template is still not aware of the object. Connecting between FormGroup in a component and a template is done using four directives: formGroup, formControlName, formGroupName and formArrayName, used in this way: &lt;form [formgroup]=myForm (ngsubmit)=register(myForm)&gt;&lt;div&gt;&lt;label&gt;Name&lt;/label&gt;&lt;input type=text=name formcontrolname=name&gt;&lt;/div&gt;&lt;div&gt;&lt;label&gt;Year of birth&lt;/label&gt;&lt;input type=text formcontrolname=birthYear&gt;&lt;/div&gt;&lt;div &gt;&lt;/div &gt;&lt;/form&gt; &gt;&lt;/form&gt; &lt;h3&gt;Locations&lt;/h3&gt;&lt;div&gt;&lt;label&gt;Country&lt;/label&gt;&lt;input type=text name=country formcontrolname=country&gt;&lt;/div&gt;&lt;div&gt;&lt;label&gt;City&lt;/label&gt;&gt;&lt;lt;input type=text name=city formcontrolname=city&gt;&lt;/div&gt;&lt;div formarrayname=phoneNumbers&gt;&lt;h3&gt;Phone Numbers&lt;/h3&gt;&lt;div *ngfor=let phoneNumberControl of myForm.controls.phoneNumbers.controls; let i=index;&gt;&lt;label&gt;Phone number {{i + 1}}&lt;/label&gt;&lt;input type=text name=phoneNumber[{{phoneId}}] [formcontrolname]=i&gt;&lt;button type=button=button click)=add()&gt;Add phone number&lt;/button&gt;&lt;/div&gt;&lt;pre&gt;{{myForm.value | json}}&lt;/pre&gt;&lt;button type=submit&gt;Register print&lt;/button&gt; &lt;button type=button (click)=printMyForm()&gt;to console&lt;/button&gt; Now, when we have FormArray, you can see that we can use this structure to render all phone numbers. And now to add support for adding and removing phone numbers (in component): remove(i: number) { (&lt;FormArray&gt;this.myForm.get('phoneNumbers')).removeAt(i); } add() { (&lt;FormArray&gt;this.myForm.get('phoneNumbers')).push(new FormControl('')); } Now we have a fully functioning reactive angular form 4. Notice the difference from template-based forms in which formgroup was created in the template (scanning the template structure) and passed to the component, in reactive forms the opposite is true, the full form group is created in the component, and then passed to the template and linked to the corresponding controls. But again, we have the same validation problem, a problem that will be solved in the next section. Validation When it comes to validation, reactive forms are much more flexible than template-based forms. Without any additional changes, we can reuse the same validation modules that were implemented earlier (for a template based on a template). Thus, by adding the validator directive, we can activate the same validation: &lt;form [formgroup]=myForm (ngsubmit)=register(myForm) countrycity= telephonenumbers= novalidate=&gt;&lt;input type=text name=name formcontrolname=name required= uniquename=&gt;&lt;show-errors [control]=myForm.controls.name&gt;&lt;/show-errors&gt; .. &lt;input type=text name=birthYear formcontrolname=birthYear required= birthyear=&gt;&lt;show-errors [control]=myForm.controls.birthYear&gt;&lt;/show-errors&gt; .. &lt;div formgroupname=location&gt; .. &lt;input type=text name=country formcontrolname=country required=&gt;&lt;show-errors [control]=myForm.controls.location.controls.country&gt;&lt;/show-errors&gt; .. &lt;input type=text name=city formcontrolname=city&gt; .. &lt;/div&gt;&lt;div formarrayname=phoneNumbers&gt;&lt;h3&gt;Phone numbers&lt;/h3&gt; .. &lt;input type=text name=phoneNumber[{{phoneId}}] [formcontrolname]=i telephonenumber=&gt;&lt;show-errors [control]=phoneNumberControl&gt;&lt;/show-errors&gt; .. &lt;/div&gt; .. &lt;/form&gt; Please note that we now do not have the NgModel directive to go to the ShowErrors component, but the complete FormGroup is already constructed and we can pass the correct &lt;/FormArray&gt;&lt;/FormArray&gt; &lt;/FormArray&gt; do pobierania błędów. Oto pełny plunker pracy z tego typu walidacji dla reaktywnych form. Ale nie byłoby fajnie, gdybyśmy po prostu ponownie za pomocą walidatorów, prawda? Przyjrzymy się, jak określić moduły sprawdzania poprawności podczas tworzenia grupy formularzy. Pamiętasz regułę 3s, o którym wspominaliśmy o konstruktorze dla FormGroup, FormControl i FormArray? Tak, powiedzieliśmy, że konstruktor może odbierać funkcje walidatora. Spróbujmy więc tego podejścia. Najpierw musimy wyodrębnić funkcje sprawdzania poprawności wszystkich walidatorów do klasy, która wystawia je jako metody statyczne: import { FormArray, FormControl, FormGroup, ValidationErrors } z @angular/forms; export class CustomValidators { static birthYear(c: FormControl): ValidationErrors { const numValue = Number(c.value); const currentYear = new Date().getFullYear(); const minYear = currentYear - 85; const maxYear = currentYear - 18; const isValid = !isNaN(numValue) &amp;&amp; numValue &gt;= minYear &amp;&amp; numValue &lt;= maxyear;= const= message={ 'years':= {= 'message':= 'the= year= must= be= a= valid= number= between= '= += minyear= += '= and= '= += maxyear= }= };= return= isvalid= null= := message;= }= static= countrycity(form:= formgroup):= validationerrors= {= const= countrycontrol=form.get('location.country'); const= citycontrol=form.get('location.city'); if= (countrycontrol= !=null &amp;&amp;= citycontrol= !=null) {= const= country=countryControl.value; const= city=cityControl.value; let= error=null; if= (country=== 'france'= &amp;&amp;= city= !== 'paris')= {= error='If the country is France, the city must be Paris' ;= }= const= message={ 'countrycity':= {= 'message':= error= }= };= return= error= message= := null;= }= }= static= uniquename(c:= formcontrol):=&gt;&lt;/=&gt; &lt;ValidationErrors&gt;{ const message = { 'uniqueName' : { 'message': 'The name is not unique' } }; return new Promise(resolve =&gt; { setTimeout(() =&gt; { resolve(c.value === 'Existing' ? message : null); }, 1000); }); } static telephoneNumber(c: FormControl): ValidationErrors { const isValidPhoneNumber = /^\d{3,3}-\d{3,3}-\d{3,3}$/.test(c.value); const message = { 'telephoneNumber': { 'message': 'The phone number must be valid (XXX-XXX-XXX, where X is a digit)' } }; return isValidPhoneNumber ? null : message; } static telephoneNumbers(form: FormGroup): ValidationErrors { const message = { 'telephoneNumbers': { 'message': 'At least one telephone number must be entered' } }; const phoneNumbers = &lt;FormArray&gt;form.get('phoneNumbers'); const hasPhoneNumbers = phoneNumbers &amp;amp; Object.keys(phoneNumbers.controls).length &gt; 0; return hasPhoneNumbers ? null : message; } } Teraz możemy zmienić tworzenie 'myForm' na: this.myForm = new FormGroup({ 'name': new FormControl('', Validators.required, CustomValidators.uniqueName), 'birthYear': new FormControl('', [Validators.required, CustomValidators.birthYear]), 'location': new FormGroup({ 'country': new FormControl('', Validators.required), new FormControl() }), 'phoneNumbers': new FormArray([this.buildPhoneNmberCompon()])},&lt;/FormArray&gt; &lt;/ValidationErrors&gt; &lt;/ValidationErrors&gt; CustomValidators.phoneNumbers]);); See? Rule 3s, when defining FormControl, multiple validators can be declared in an array, and if you want to add multiple validators to a Form Group they must be merged using Validators.compose (also Validators.composeAsync is available). And that's all, validation should work completely. This example also includes Plunker. It goes out to everyone that hates the new word. To work with reactive forms, a shortcut is available — a constructor to be more precise. FormBuilder allows you to create a full Formgroup using the constructor pattern. You can do this by changing the FormGroup construct in this way: constructor (private fb: FormBuilder) { } ngOnInit() { this.myForm = this.fb.group({ 'name': ['', Validators.required, CustomValidators.uniqueName], 'birthYear': ['', [Validators.required, CustomValidators.birthYear]], 'location': this.fb.group({ 'country': ['', Validators.required], 'city': '' }), 'phoneNumbers': this.fb.array([this.buildPhoneNumberComponent()]) }, { validator: Validators.compose([CustomValidators.countryCity, CustomValidators.telephoneNumbers]) }); } Not a very big improvement from the occurrence of the new, but it is not. And, don't worry, there's a plunker for that as well. In this second section, we looked at reactive forms in Angular 4. As you can see, this is a completely new approach to adding support for forms. Although it seems complete, this approach gives the developer total control over the underlying structure that enables forms in Angular 4. In addition, because reactive forms are created manually in a component, they are visible and provide an easy way to test and control, while this was not the case with template-based forms. Nesting form nesting forms is in some cases useful and a required feature, mainly when the status (e.g. severity) of a subgroup of control forms must be specified. Think of a tree of ingredients; we may be interested in the importance of a component in the middle of this hierarchy. It would be really difficult to achieve if we had one form in the main ingredient. But, oh boy, this is a sensitive way on several levels. First, nesting real HTML forms, according to html specifications, is not allowed. We can try to nest &lt;form&gt; Elements. In some browsers it may actually work, but we can't be sure it will work on all browsers because it's not in the HTML specification. In AngularJS, the way around this limitation was to use the ngForm directive, which offered AngularJS form functions (only grouping controls, not all form capabilities, such as server posting), but could be placed on any element. In addition, in AngularJS, nesting forms (when I say mold, I mean NgForm) was available out of the box. Simply declaring a tree of several With the ngForm directive, the state of each form has been propagated up to the root element. W &lt;/form&gt; &lt;/form&gt; section, we'll look at a few options on how to nest forms. I would like to stress that we can distinguish between two types of nesting: within the same component and in different components. Nesting in the same component If you look at an example that we implemented with a template-based approach and a reactive approach, you'll notice that we have two internal control containers, location, and phone numbers. To create this container to store values in a separate property object, we used the NgModelGroup, FormGroupName, and FormArrayName directives. If you have a good look at the definition of each directive, you will notice that each of them extends the ControlContainer class (directly or indirectly). Well, what you know, it turns out that this is enough to provide the functionality we need by wrapping the state of all internal controls and propagating this state to the parent. For a template-based form, we need to make the following changes: &lt;form #myform=ngForm (ngsubmit)=register(myForm) novalidate=&gt; .. &lt;div ngmodelgroup=location #location=ngModelGroup countrycity=&gt;.. &lt;show-errors [control]=location&gt;&lt;/show-errors&gt;&lt;/div&gt;&lt;div ngmodelgroup=phoneNumbers #phonenumbers=ngModelGroup telephonenumbers=&gt;.. &lt;show-errors [control]=phoneNumbers&gt;&lt;/show-errors&gt;&lt;/div&gt;&lt;/form&gt; We have added the ShowErrors component to each group to display errors directly associated only with that group. Since we have moved countryCity and phoneNumbers validators to another level, we also need to update them accordingly: // country-city-validator.directive.ts let countryControl = form.get('country'); let cityControl = form.get('city'); And phone numbers-validator.directive.ts to: let phoneNumbers = form.controls; letphoneNumbers = phoneNumbers &amp;amp; Object.keys(phoneNumbers).length &gt; 0; You can try the full example from template-based forms including Plunker. And for reactive forms, we will need similar changes: &lt;form [formgroup]=myForm (ngsubmit)=register(myForm) novalidate=&gt; .. &lt;div formgroupname=location&gt;.. &lt;show-errors [control]=myForm.controls.location&gt;&lt;/show-errors&gt;&lt;/div&gt;&lt;div formarrayname=phoneNumbers&gt;.. &lt;show-errors [control]=myForm.controls.phoneNumbers&gt;&lt;/show-errors&gt;&lt;/div&gt; .. &lt;/form&gt; The same changes from country-city-validator.directive.ts and phone numbers-validator.directive.ts are required for countryCity and phoneNumbers validators in CustomValidators to properly locate controls. Finally, we need to modify the Form Group structure to: this.myForm = new FormGroup({ 'name': new FormControl('', Validators.required, CustomValidators.uniqueName), 'birthYear': new FormControl('', [Validators.required, CustomValidators.birthYear]), 'location': new FormGroup({ 'country': new FormControl('', Validators.required), 'city': new FormControl() }, CustomValidators.countryCity), new FormArray([this.buildPhoneNumberComponcompent()], FormArray([this.buildPhoneNumberComponcompent()], }); And you have it - we have improved validation for reactive forms as well and as expected, Plunker for this example. Nesting in different components It can be a shock to all AngularJS developers, but in Angular 4 nesting forms in different components does not work out of the box. I will be honest with you; in my opinion, nesting is not supported for some reason (probably not because the Angular 4 team just forgot about it). The main enforced principle of Angular4 is the one-way flow of data, from top to bottom through the component tree. The whole structure was designed in such a way, where vital surgery, detection of changes, is performed in the same way, from top to bottom. If we fully comply with this principle, we should not have any problems, and all changes should be resolved within a single full detection cycle. That's at least the idea. To verify that the one-way data flow is implemented correctly, the sympathetic Angular 4 team have implemented a feature that, after each change detection cycle, while in development mode, an additional change detection round is triggered to verify that no binding has changed as a result of backflow. What this means, let's think of the component tree (C1, C2, C3, C4), as in Figure 1, change detection begins in component C1, continues in component C2, and ends in component C3. If we have any method in C3 with a side effect that changes some bindings in C1, it means that we are moving the data up, but the detection of changes for C1 has already passed. When you work in development mode, the second round starts and notices a change in C1 that occurred as a result of method execution in some child components. Then you're in trouble and you'll probably see the exception expression changed after checking. You can simply disable development mode and there will be no exception, but the problem will not be solved; plus, how would you sleep at night, just sweeping all your problems under the carpet anyway? Once you know this, think about what we do if we aggregate the status of forms. True, the data is moved up the component tree. Even when working with single-level forms, the integration of form forms (ngModel) and the form itself is not so pleasant. Trigger an additional change detection cycle when you record or update the value of a control (this is done with a resolved promise, but keep it secret). Why do I need an extra round? Well, for the same reason, the data flows up, from the control to the form. But sometimes nesting forms in multiple components is a required feature, and we need to think about a solution to support this requirement. From what we know so far, the first idea that comes to mind is to use reactive forms, create a full form tree in some root component, and then pass the mold child components as input. In this way, you have firmly connected the parent to the child and cluttered the business logic of the root component with support for creating all child forms. Come on, we are professionals, I'm sure we can find a way to create completely isolated components with molds and provide a way in which the form simply promotes you to anyone who is a parent. All this is said, here is a directive that allows nesting Angular 4 forms (implemented because it was needed for the project): import { OnInit, OnDestroy, Directive, SkipSelf, Optional, Attribute, Injector, Input } from @angular / core; import { NgForm, FormArray, FormGroup, AbstractControl } from @angular/forms; const resolvedPromise = Promise.resolve(null); @Directive({ selector: '[nestableForm]' }) export class NestableFormDirective implements OnInit, OnDestroy { private static readonly FORM_ARRAY_NAME = 'CHILD_FORMS'; private currentForm: FormGroup; @Input() private formGroup: FormGroup; constructor(@SkipSelf() @Optional() private parentForm: NestableFormDirective, private injector: Injector, @Attribute('rootNestableForm') private isRoot) { } ngOnInit() { if (!this.currentForm) { // NOTE: at this point both NgForm and ReactiveFrom should be available this.executePostponed(() =&gt; &lt;2&gt; this.resolveAndRegister()); } ngOnDestroy() { this.executePostponed(() =&gt; this.parentForm.removeControl(this.currentForm)); } public registerNestedForm(control: AbstractControl): void { // NOTE: prevents circular reference (adding to itself) if (control === this.currentForm) { throw new Error('Trying to add yourself! The nest form can only be added in the parent NgForm or FormGroup.'); } (&lt;FormArray&gt;this.currentForm.get(NestableFormDirective.FORM_ARRAY_NAME)).push(control); } public removeControl(control: AbstractControl): void { const array = (&lt;FormArray&gt;this.currentForm.get(NestableFormDirective.FORM_ARRAY_NAME)); const idx = array.controls.indexOf(control); array.removeAt(idx); } private resolveAndRegister(): void { this.currentForm = this.resolveCurrentForm(); this.currentForm.addControl(NestableFormDirective.FORM_ARRAY_NAME, new FormArray([])); this.registerToParent(); } private resolveCurrentForm(): FormGroup { // NOTE: template-based or model-based =&gt; specified by input formGroup return to.formGroup ? this.formGroup : this.injector.get(NgForm).control; } private registerToParent(): { if (this.parentForm != null &amp;&amp;amp;! this.isRoot) { this.parentForm.registerNestedForm(this.currentForm); } } private executePostponed(callback: () =&gt; void): void { resolvedPromise.then(() =&gt; callback()); } } The following GIF shows one main component containing form 1, and inside this form there is another nested component, component-2. component-2 contains form-2, which has a nested form-2.1, form-2.2, and an ingredient (component-3) that has a reactive tree and an ingredient (component-4) that contains the form, is isolated from all other forms. I know it's pretty messy, but I wanted to create a fairly complex scenario to show the functionality of this&lt;/FormArray&gt; &lt;/FormArray&gt; &lt;/FormArray&gt; An example is implemented in this Plunker. Features that this offers: Allows nesting by adding nestableForm directives to elements: forms, ngForm, [ngForm], [formGroup] Works with template-based and reactive forms Allows you to create a form tree that includes multiple components Isolates a suborder of forms from rootNestableForm=true (does not register with the parent nestableForm) This directive allows the form in the child component to register with the first parent nestableForm, regardless of whether the parent form is declared in the same component or not. We will go into the details of the implementation. First, let's look at the builder. The first argument is: @SkipSelf() @Optional() private parentForm: NestableFormDirective This will search for the first NestableFormDirective parent. @SkipSelf not to match up and @Optional because you might not find a parent for the main form. Now we have a reference to the parent nested form. The second argument is: Private Injector: Injector Injector is used to retrieve the current FormGroup supplier (template or reactive). And the last argument is: @Attribute('rootNestableForm') private isRoot to get a value that specifies whether this form is isolated from the form tree. Then, in ngInit as the postponed action (remember the reversed data flow?), the current form group is recognized, the new formArray control named CHILD_FORMS is registered to that form group (where the child forms will be registered), and as the last action, the current FormGroup group is registered as a child to the parent nesting form. The ngOnDestroy action is performed when the form is destroyed. After destroy, again as a postponed activity, the current form is removed from the parent (unregister). You can further customize the directive for nested forms to meet your specific needs by removing support for reactive forms, registering each child form with a specific name (not the CHILD_FORMS array), and so on. This implementation of the nestableForm directive has met the requirements of the project and is presented here as such. It covers several basic cases, such as adding a new form or dynamically deleting an existing form (*ngIf) and propagating the state of the form to the parent. This basically boils down to operations that can be resolved within a single change detection cycle (with or without deferral). If you want a more advanced scenario, such as adding conditional validation to some inputs (e.g. [required]=someCondition) that require 2 rounds of change detection, it will not work because of the one-detection-cycle-resolution rule imposed by Angular 4. In any case, if you plan to use this directive or implement another solution, you should be especially careful in to the things that have been mentioned related to the detection of changes. At this point, this is how Angular 4 is implemented. That could change in the future — we don't know. The current configuration and enforced constraint in angular 4 mentioned in this article may be a defect or It remains to be seen. Forms Made easy with Angular 4 As you can see, the Angular team has done a really good job in providing many form-related features. I hope this post will serve as a complete guide to working with different types of forms in Angular 4, also giving you insight into some of the more advanced concepts such as mold nesting and change detection. Despite all the different posts related to Angular 4 forms (or another Angular 4 theme on this issue), in my opinion, the best starting point is the official Angular 4 documentation. Also, Angular guys have nice documentation in their code. Many times I have found a solution, looking at their source code and documentation there, without Googling or anything. About nesting forms, discussed in the last section, I believe that any AngularJS programmer who starts learning Angular 4 will come across this problem at some point, which was kind of my inspiration to write this post. As we have seen, there are two types of forms and there is no strict rule that you can not use them together. It's nice to keep the codebase clean and consistent, but sometimes something can be done more easily with form-based templates and, sometimes, the other way around. So, if you don't mind slightly larger bundle sizes, I suggest you use what you consider more appropriate case by case. Just do not mix them in the same ingredient, as this will probably lead to some confusion. Plunkers used in this post similar: Smart .js form validation