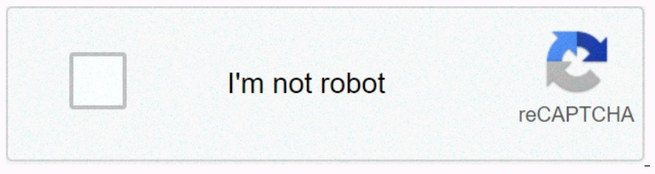

Continue

Angular 4 form validation before submit

One frame. Mobile & desktop. Reading time: 20 minutesShapes are probably the most crucial aspect of your web application. While we often get to events by clicking on links or moving the mouse, these are forms where we get most of our rich data entries from users. On the surface, forms seem simple: make an input tag, fill it out, and send the hits. How hard could it be? It turns out that forms can be very complex. Here are some reasons why: Form entries are designed to change data, How on the side of both i on the serveru Changes should be reflected i elsewhere on the side Users have a lot of leeway in what they enter, are you need to validate the values UI need to state clearly expectations and errors, any if Dependent fields can have complex logic we want to be able to test our forms, without anglying on DOM selectorsly Thankfully, Angular has tools to help with all of these things. FormControls encapsulate inputs in our shapes and give us objects to work with them Validators give us the ability to confirm inputs, however we wish observers should look at our format for change and respond accordingly In this section we will walk through the building forms, step by step. We're going to start with a few simple shapes and build a more complex logic. FormControls and FormGroups Two basic objects in Kot formats are FormControl and FormGroup. FormControl A FormControl represents one entry field - is the smallest unit of an angular shape. FormControls includes the value of a field and a country, such as valid, dirty (modified), or error-free. For example, here's how to use FormControl in TypeScript: // create a new FormControl with the value Nate let nameControl = new FormControl(Nate); years name = nameControl.value; -> Nate // now you can query this control for specific values: nameControl.errors // -> StringMap<string, any=>error nameControl.dirty // -> wrong nameControl.valid // -> true // etc. To build forms, we create FormControls (and FormControls groups) and then attach metadata and logic to them. Like many things in Kotni, we have a class (FormControl, in this case) that we attach to the DOM with an attribute (FormControl, in this case). For example, we may have the following in our form: <!-- part of some bigger form --><input type=text[formcontrol]=name> This will create a new FormControl object within our form. We will talk more about how this works below. FormGroup Most forms have more than one field, so we need a way to manage multiple formControls. If we wanted to validate our form, it is difficult to repeat above the FormControls field and check each FormControl for validity. FormGroups can solve this problem by providing a wrapper interface around formcontrols. Here's how to create FormGroup: let personInfo = new FormGroup({ firstName: new FormControl(Nate), last name: new FormControl(Murray), novo </string,> </string,> }) FormGroup and FormControl have common ancestors (AbstractControl). This means that we can check the status or value of a personInfo just as easily as one FormControl: personInfo.value; -> { // firstName: Nate, // lastName: Murray, // zip: 90210 // } // we can now query this control group for specific values that have sensibility // values u depend on the boy's FormControl value: personInfo.errors // -> StringMap of errors<string, any=> personInfo.dirty // -> false personInfo.valid // -> true // etc. When we tried to get a value from the FormGroup group, we received an object with a pair of key values. This is a really handy way to get a whole set of values from our format without the need to itere across each FormControl individually. Our first form There are many moving pieces to create a form, and several important ones that we haven't touched. Let's get in on the full case and explain every piece as we move on. Most of the code as a case can be found in the row on this post, but if you want to download a version of full-source cases enter the email below: Here is a screenshot of the first form we will build: Demo form with Sku: A simple version In our imaginary application we create an e-commerce site where we are listed products for sale. In this application we need to save the SKU product, so we create a simple form that takes the SKU as the only input field. SKU stands for storage unit. This is a term for a unique id for a product that will be tracked in inventory. When we talk about SKU, we are talking about a man-readable iD item. Our design is super simple: we have a single input for the sku (with tag) and a broadcast button. Let's turn this form into a component. If you remember, there are three parts to define a component: Configure the decorator @Component() Create a template Implement custom functionality in the component definition class Let's take these inline: Load the formModule To use a new form library, we must first make sure that we import the form library into NgModule. There are two ways of using forms in Kot and we will talk about them in this section: using FormsModule or using ReactiveFormsModule. Since we're both going to use it, we're going to import them into the module. To do this, we do the following in our application.ts, where we run the application: // app.module.ts imports { FormsModule, ReactiveFormsModule } from @angular/forms; further down... @NgModule({ statements: [AppComponent, DemoFormSkuComponent, // ... our declaration here], import: [BrowserModule, FormsModule, // <---add= this= reactiveformsmodule=><--> <-- and this], bootstrap: [AppComponent] }) class AppModule {} ensures This we're able to use the form directives in our views. At the risk of jumping ahead, the FormsModule gives us a driven template template directives such as: Whereas ReactiveFormsModule gives us reactive driven directives like formControl and and= this=]= })= class= appmodule= {}= this= ensures= that= we're= able= to= use= the= form= directives= in= our= views.= at= the= risk= of= jumping= ahead,= the= formsmodule= gives= us= template= driven= directives= such= as:= whereas= reactiveformsmodule= gives= us= reactive= driven= directives= like= formcontrol= and=></-- and this], bootstrap: [AppComponent] }) class AppModule {} This ensures that we're able to use the form directives in our views. At the risk of jumping ahead, the FormsModule gives us template driven directives such as: Whereas ReactiveFormsModule gives us reactive driven directives like formControl and > </string,> </string,> ... and more. We have not talked about how to apply these directives or what they are doing, but we will be doing it shortly. For now, know that importing FormsModule and ReactiveFormsModule into our NgModule means that we can apply any of the directives in this list in our view template or inject any of their providers into our components. Reactive-vs. Template-Driven Forms Angle allows you to define forms in two different ways: reactive or template drive. You can see a comparison of two ways here. Instead of describing how different they are, we'll show you examples of different ways in which you can build forms - then you can decide which one is right for your application. Simple Form SKU: @Component decorator first, let's start creating what's called a template drive form. Start with our component: import { Component, OnInit } from @angular/kernel; @Component({ selector: 'app-demo-form-sku', templateUrl: './demo-form-sku.component.html', Here we define the selection app-demo-form-sku. In this case, we can use this component by labeling the app-demo-form-sku as this: Simple form SKU: template Let's <app-demo-form-sku></app-demo-form-sku> our template: <div class=ui raised segment><h2 class=ui header>Demo form: Sku</h2><form #f=ngForm (ngsubmit)=onSubmit(f.value) class=ui form><div class=field><for label=skuInput>SKU</label><input type=text id=skuInput placeholder=SKU name=sku ngmodel=></div><type=class submit=ui button>Submit</button></form><div/> form & NgForm Now things get interesting: since we imported FormsModule, that NgForm is available for our view. Remember that whenever we make directives available to our position, they will relate to any element appropriate to their selector. NgForm does something handy, but not obvious: it includes a form tag in its selector (instead of requiring you to explicitly add ngForm as an attribute). This means that if you import FormsModule, NgForm will automatically connect to any <form> labels you have in the view. This is really useful, but potentially confusing because it's happening behind the scenes. There are two important parts of the functionality that NgForm gives us: FormGroup named ngForm A (ngSubmit) output You see, that we both use in <form> tag in our opinion: <form #f=ngForm (ngsubmit)=onSubmit(f.value) first= we=have= #f=ngForm .= the= #v=thing syn=says that= that=what=clicking it= a==a=local==forth==here=here=here=creating =an=or=ngform,= for= this=this view==the =change= #f.= where=ng=ng=ng=ng=from=the=the=first==it=it=came=from=the=the=the=ng=ng=%,[1=-what=type= of the object= is= ng=ngform=it= it=a= a=the=the can=can=use f=a=a=change=and=our view.= and= exactly= what= we= do= in= the= (ngsubmit)= output.= astute= readers= might= notice= that= i= just= said= above= that= ngform= is= automatically= attached= to=></form></form></form> to=></form><form></form></form> (due to the default NgForm selector), which means that we do not need to add the ngForm attribute to use NgForm. But here we are putting ngForm in the attribute (value) tag. Is that a mistake? No, that's not a mistake. If ngForm was the attribute key, then we would tell Angular that we want to use NgForm on this attribute. In this case, when assigning a reference, we use ngForm as an attribute. This means that the value of the evaluated expression ngForm should be assigned to the local variable of the f. ngForm template already on this item, and you can think of it as exporting this Form Group so that it can be referenced elsewhere in our view. Using syntax, we are bound to ngSubmit: (ngSubmit)=onSubmit(f.value) for ngSubmit. (ngSubmit) - comes from NgForm onSubmit() - will be implemented in our component definition class (below) f.value - f is formgroup that we have set out above. And .value will return the key/value pairs of this FormGroup Total, and this line says when I submit the form, call my component instance and specify the form value as an argument. input & NgModel Our entry tag has some things to touch, before we talk about NgModel: <form #f=ngForm (ngsubmit)=onSubmit(f.value) class=ui form><div class=field><label for=skuInput>SKU</label><input type=text id=skuInput placeholder=SKU name=sku ngmodel=></div> class=ui form in class=field - these classes are completely optional. They come from the framework of the CSS Semantic UI. I've added them in some of our cases just to give them a nice CSS coat, but they are not part of Angular. The attribute code and the input id attribute match, as according to W3C standard We have defined an SKU placeholder that is just a hint to the user of what this entry should say when the empty NgModel directive specifies the selector ngModel. This means that it can be attached to the input tag by adding the following attribute type: ngModel=whatever. In this case, we determine ngModel without attribute value. There are a few different ways to define ngModel in templates and this is the first. When we use ngModel without the attribute value we specify: We want to create a FormControl on this form with the name sku (because of the name attribute on the entry tag) NgModel creates a new FormControl that is automatically added to the FormGroup parent group (in this case on the form) and then links the DOM element to this new FormControl. This means that it specifies the link between the input tag in our view and FormControl and the association matches the name, in this case the sku. NgModel vs. ngModel: what is the difference? In general, when we use PascalCase, such as NgModel, we define the class and refer to the object as defined in the code. The example below (CamelCase), as in ngModel, comes from the optional element of the directive and is used only in the DOM/ template. This is </form>NgModel and FormControl should be considered separate objects. NgModel is a directive that you use in your view, and formControl is an object used to present data and verify in your form. Sometimes we want to make a two-bedroom binding with ngModel as we used to in Corner 1. We're going to look at how to do it towards the end of this chapter. Simple SKU Form: Component Definition Class Now let's look at our class definition: export class DemoFormSkuComponent implements OnInit { constructor() { } ngOnInit() { } onSubmit(form: any): void { console.log('you submitted value:', form); } } Here our class provides one function: onSubmit. This is a function that is named when the form is submitted. For now we will only .log the value that is transferred to. If we combine it all, our code list looks like: import { Component, OnInit } from @angular/kernel; @Component({ selector: 'app-demo-form-sku', templateUrl: './demo-form-sku.component.html', styles: [] }) export class DemoFormSkuComponent implements OnInit { constructor(submitted) { } ngOnInit() { } onSubmit(form: any): void { console.log('you value:', form); } } and the template: <div class=ui raised segment><h2 class=ui header>Demo form: Sku</h2><form #f=ngForm(ngsubmit)=onSubmit(f.value) class=ui form><div class=field><label;label for=skuInput>SKU</label><input type=text id=skuInput placeholder=SKU name=sku ngmodel=></div><button type=submit class=ui button>Broadcast</button></form></div; If we test this in our browser, here's how it looks: Demo Form with Sku: Simple version submitted using FormBuilder Building our FormControls and FormGroups implicitly using ngForm is convenient , but it does not give us much adjustment options. A more flexible and more common way to configure forms is to use formBuilder. FormBuilder is a hard-called assistant class that helps us build forms. As you remember, forms made from FormControls and FormGroups and FormBuilder help us to keep them (you can think of it as a factory facility). Add formBuilder to our previous example. Let's see: how to use FormBuilder in component definition class, how to use our custom formGroup on a form in the Jet Forms view with FormBuilder For this component, we will use formGroup and formControl directives, which means that we need to import the appropriate classes. We start by importing them by importing { Component, OnInit } from @angular/kernel; import { FormBuilder, FormGroup } @angular/forms; By using FormBuilder We inject FormBuilder by creating an argument in the constructor of our component class: importing { Component, OnInit } from @angular/kernel; import { FormBuilder, FormGroup } @angular/forms; @Component({ selector: 'app-demo-form-sku-with-builder', templateUrl: './demo-form-sku-with-builder.component.html', styles: [] }) Export Class DemoFormSkuWithBuilderCompent implements OnInit { Form group; constructor(fb: FormBuilder) { this.myForm = fb.group({ 'sku': ['ABC123'] }); } ngOnInit() { } onSubmit(value: string: void { console.log('you submitted value: ', value); } During injection, a FormBuilder is created and assigned to the variable fb (in the constructor). There are two main functions that we will use on FormBuilder: Control – create a new FormControl group – create a new FormGroup Notification that we have set a new instance variable called myForm on this class. (We could just as easily call it a form, but I want to distinguish between our FormGroup and the form we had before.) myForm is typed to formgroup. FormGroup is created by calling fb.group(). .group takes the subject of key value pairs that define FormControls in this group. In this case, we are setting one control panel and the value is [ABC123] - this is the default value of this control ABC123. (You will notice that this is a field. This is because we will later add more configuration options.) Now that we have my Form, we have to use it in a view (that is, we need to fit it with our form element). By using my notification in the view We want to change <form> to use my List. If you remember, we said in the last section that ngForm is automatically used for us when we use FormsModule. We also mentioned that ngForm creates its own FormGroup. Well, in this case, we do not want to use outside FormGroup. Instead, we want to use our myForm instance variable, which we created with our FormBuilder. How can we do that? Kot provides the second directive we use, when we have an existing formGroup: it's called formGroup and we use it like this: <h2 class=ui header>Demo form: Sku z builder</h2><form [formgroup]=myForm here= we're= telling= angular= that we= want= it=use=my=myform as=the=the format=for=this=type=type=how=earlier said that= when= using =ng=ng=ngform will be=automatically applied to=a=></format><form> element? There is an exception: NgForm will not be used for <form>which has a Group format. If you are interested, the NgForm selector is: {lang=javascript,line-numbers=off} form:not([ngNoForm]):not(formGroup),ngForm,[ngForm] This means that you can have a form that will not use NgForm using the ngNoForm attribute. We also need to change toSubmit to use myForm instead of f, because it is now myForm that has our configuration and values. There is one last thing we need to do to make this work: plug our FormControl to the input tag. When we want to connect an existing FormControl to a input, we use formControl: <label for=skuput>SKU</label><input type=text=skuInput placeholder=SKU [formcontrol]=myForm.controls['sku']> Here we are instructing the Control Directive form to view myForm.controls and use the existing FormControl file for this entry. Attempts! So videti vse skupaj: uvoz { Component, OnInit } iz @angular/jedro; uvoz { FormBuilder, FormGroup } iz @angular/obrazcev; </form></form></form> </form></form></form> picker: app-demo-form-sku-with-builder, templateUrl: ./demo-form-sku-with-builder.component.html, styles: [] }) export class DemoFormSkuWithBuilderComponent implements OnInit { myForm: FormGroup; Constructor(fb: FormBuilder) { this.myForm = fb.group({ 'sku': ['ABC123'] }); } ngOnInit() { } onSubmit(value: string): void { console.log('submitted you value: ', value); } and the template: <div class=ui raised

segment><h2 class=ui header>Demo form: Sku z builder</h2><form [formgroup]=myForm (ngsub =onSubmit(myForm.value) class=ui form><div class=field><label for=skuInput>SKU</label><input type=text id=skuInput placeholder=SKU [formcontrol]=myForm.controls['sku']></div><button type=submit class=ui button>Submit</button></form ></div> Remember: To create a new FormGroup and FormControls implicit application: But to connect to existing FormGroup and FormControls use: formGroup and formControl Add validation Our users will not always enter data in exactly the correct format. If someone enters the data in the wrong format, we want to give them feedback and not allow them to submit a form. We use validators for this. Validators are provided with the Validators module and the simplest validator is Validators.required, which simply says that a specific field is required or formcontrol will be considered invalid. To use validators we need to do two things: Assign a validator to the FormControl object Check the status of the validator in the view and take appropriate action To assign the validator formControl to the object we simply pass as another argument for our FormControl constructor: let control = new FormControl('sku', Validators.required); Or in our case, because we use FormBuilder we use the following syntax: constructor(fb: FormBuilder) { this.myForm = fb.group({ 'sku': [', Validators.required] }); this.sku = this.myForm.controls['sku']; } Now we have to use our verification in sight. There are two ways we can access validation values in a view: FormControl can explicitly assign a class instance variable - which is more verbose but gives us easy access to FormControl in the view. In the view, we can find formControl from my Form. This requires less work in the component definition class, but there is a slightly more verbose in the view. To make this difference clearer, let's look at this example in both ways: Specifically setting formcontrol as instance variables Here's a screenshot of how our validation form will look: Demo validation form The most flexible way to handle individual FormControls in your view is to set each FormControl up as an instance of a variable in the component definition class. Here's how you could set the sku in our class: export class DemoFormWithValidationsExplicitComponent { myForm: FormGroup; AbstractControl; konstruktor(fb: FormBuilder) { this.myForm = fb.group({ 'sku': ['', ''], }); this.sku = this.myForm.controls['sku']; } onSubmit(value: string): void { console.log('you submitted value: ', value); } } Notice that: We are setting sku: AbstractControl at the top of the class and we assign it.sku when we create myForm with FormBuilder This is great because it means you can reference the sku anywhere in our view components. This should set the instance variable for each field in our form. For large shapes, this can get quite verbose. Now that we have our coku validated, I want to look at four different ways, which we can use in our view: Validate our entire form and display the message Validation of our individual field and display the message Validation of our individual field and coloring the field red if the validation of our individual field is invalid at a specific request and display Message Form Message We can validate the entire form by viewing myForm.valid: <div *ngif=!myForm.valid remember,= myform= is= a= formgroup= and= a= formgroup= is= valid= if= all= of =the=the=children= formcontrols are= also valid.= field= we=can=also=a= message= for=the=specific=field if= that=field's= format control=is= disabled:= [formcontrol]=sku><div *ngif=!sku.valid class=ui error message>SKU is not valid</div><div *ngif=sku.hasError('required') field= i'm= using= the= semantic= ui= css= framework's= css= class.= error. =which= means= if= i= add= the= class= error= to= the=></div><div class=field> will display an input mark with a red border. To do this, we can use the property syntax to set the conditional classes: <div class=field [class.error]=!sku.valid && sku.touched> Here you notice that we have two conditions for setting the class .error: We check for !sku.valid and sku.touched. The idea here is that we want to show the error state only if the user tried to edit the form (touched it) and is now invalid. To test this, type some information in the entry mark and delete the contents of the field. Specific validation The form field can be invalid for several reasons. We often want to show a different message, depending on the reason for the failed validation. To find a specific validation error, we use the hasError method: <div *ngif=sku.hasError('required') class=ui error message>SKU</div is required. This means that you can download another path argument to find a specific field from the formGroup. For example, the previous example could be written as: <div *ngif=myForm.hasError('required', 'sku') class=error>SKU</div is required together Here is a complete list of the code of our form validation form with formControl set as instance variable: { Import } from @angular/kernel; import { FormBuilder, FormGroup, AbstractControl } iz @angular/obrazcev; @Component({ selector: app-demo-form-with-validations-explicit, </div></div></div> </div></div></div> ./demo-form-with-validations-explicit.component.html, slogi: [] }) izvozni razred DemoFormWithValidationsExplicitComponent { myForm: FormGroup; sku: AbstractControl; konstruktor(fb: FormBuilder) { this.myForm = fb.group({ 'sku': ['', Validators.required] }); this.sku = this.myForm.controls['sku']; } onSubmit(value: string): void { console.log('you submitted value: ', value); } } } In predloga: <div class=ui raised segment><h2 class=ui header>Demo obrazec: z validacijami (eksplicitno)</h2><form [formgroup]=myForm (ngsubmit)=onSubmit(myForm.value) class=ui form [class.error]=!myForm.valid && myForm.touched><div class=field [class.error]=!sku.valid && sku.touched><label for=skuInput>SKU</label><input type=text id=skuInput placeholder=SKU [formcontrol]=sku><div *ngif=!sku.valid class=ui error message>SKU ni veljaven</div></div><div *ngif=sku.hasError('required') class=ui error message>Zahtevana je SKU</div></div></div><div *ngif=!myForm.valid class=ui error message>Obrazec ni veljaven</div><button type=submit class=ui button>Oddaj</button></form></div>Odstranjevanje spremenljivke primerka sku V zgornjem primeru smo nastavili sku : AbstractControl kot spremenljivko primerka. Pogosto ne bomo želeli ustvariti spremenljivke primerka za vsak AbstractControl, kako bi se sklicevanje na to FormControl v našem pogledu brez spremenljivke primerka? Namesto tega lahko uporabimo lastnost myForm.controls kot v: <label for=skuInput>SKU</label><input type=text id=skuInput placeholder=SKU [formcontrol]=myForm.controls['sku']></div *ngif=!myForm.controls['sku'].valid class=ui error message>SKU ni veljaven</div></div *ngif=myForm.controls['sku'].hasError('required') in= this= way= we= can= access= the= sku= control= without= being= forced= to= explicitly= add= it= as= an= instance= variable= on= the= component= class.= we= used= bracket-notation,= e.g.= myform.controls['sku'].= we= could= also= use= the= dot-notation,= e.g= myform.controls.sku.= in= general,= be= aware= that= typescript= may= give= a= warning= if= you= use= the= dot-notation= and= the= object= is= not= properly= typed= (but= that= is= not= a= problem= here).= custom= validations= we= often= are= going= to= want= to= write= our= own= custom= validations.= let's= take= a= look= at= how= to= do= that.= to= see= how= validators= are= implemented ,= let's= look= at= validators.required= from= the= angular= core= source:= export= class= validators= {= static= required(c:= formcontrol):=></div><string, boolean=> { return isBlank(c.value) || c.value == ? {zahtevano: true} : null; } Validator: - Vzame FormControl kot svoj vnos in - Vrne StringMap, kjer je ključ koda napake in vrednost je res, če ne uspe Pisanje Validator Recimo, da imamo posebne zahteve za našo<string, boolean=> sku. Recimo, da mora naš sku začeti s 123. Lahko bi napisali kot je to: funkcija skuValidator(control: FormControl): { [s: niz]: boolean } { if (!control.value.match(/^123/)) { return {invalidSku: true}; } } } Ta validator vrne kodo napake neveljavnoSku, če je </string,></string,> </string,></string,> (checkable value) does not start with 123. Assigning the validator to FormControl Now we need to add a validator to our FormControl. However, there is one small problem: we already have a validator on the side. How can we add multiple validators to one field? To do this we use Validators.compose: constructor(fb: FormBuilder) { this.myForm = fb.group({ 'sku': ['', Validators.compose([Validators.required, skuValidator])]] }); The validators.composition are wrapping our two validators and allowing us to assign them to FormControl. FormControl is invalid unless both confirmations are valid. Now we can use our new validator in the view: <div *ngif=sku.hasEror('invalidSku') class=ui error message>SKU must start with 123</div> Note that in this section I use explicit notation about adding an instance variable for each FormControl. This means that the view in this section refers to FormControl. If you run a sample code, you will notice that if you enter something in a field, the required validation will be filled in, but the validation is invalid. This is excellent - it means that we can partially confirm our fields and display the relevant messages. Watching For Changes So far we value from our form extracted only by calling toSubmit when the form is submitted. However, we often want to keep an eye on any changes in the values in the control panel. Both FormGroup and FormControl have EventEmitter that can be used to observe changes. EventEmitter is obsessed, which means that it complies with a certain specification for observing changes. If you're interested in spec that can be viewed, find it here To view changes to the control we have: get access to EventEmitter by calling control.valueChanges. Then add the observer using the .subscribe method Here is an example: constructor(fb: FormBuilder) { this.myForm = fb.group({ sku: ['', Validators.required] }); this.sku = this.myForm.controls['sku']; this.sku.valueChanges.subscribe((value: string) => {console.log('sku changed to:', value); }); this.myForm.valueChanges.subscribe((form: any) => { console.log('formchanged to:', form); }); Here we see two separate events: changes in the field of the sku and changes to the form as a whole. The observation we are making is an object with one key: the following (there are other keys that you can download, but we will not worry about these now). The following is the function that we want to call with a new value whenever the value changes. If we enter 'kj' in the text box we will see in our console: change the juice to: k form changed to: Object {sku: k} changed to: kj form changed to: Object {sku: kj} As you can see each keystrose causes a change of control so that our observation is triggered. When observing a single FormControl, we receive a value (e.g. kj), but when we observe the entire shape, we get an object of key value pairs (e.g. {sku: kj}). ngModel NgModel is a special directive: the model fits the form. two-bedroom data binding. Two-way data binding is almost always more complex and difficult to reason with regard to one-way data binding. The angle is built in such a way that it generally has a single direction of data: top to bottom. However, when it comes to shapes, there are times when it is easier to opt for a two-way landing. Just because you have used the ng-model in the past in corner 1, do not rush immediately using ngModela. There are good reasons to avoid a two-smiley data commitment. Of course, ngModel may be handy, but please note that we do not necessarily rely on two-way data binding as much as in Corner 1. Let's change the form a little bit and say we want to inaute the product name. We will use ngModel to synchronize the component variable with the view. First, here is our component definition class: DemoFormNgModelComponent { productName: string; constructor() { this.productName = ng-book: The Complete Guide to Angular } onSubmit(value: void { console.log('you submitted value: ', value); } } Notice that we simply store the product name: the string as the instance variable. Then we use ngModel on input <label for=productNameInput>tag: Product name</label><input type=text id=productNameInput placeholder=Product Name name=productName[(ngmodel)]=productName> Now you notice something - the syntax for ngModel is funny: we use both brackets and brackets around the ngModel attribute! The idea with which we want to use is to use input brackets and output brackets. It's a sign of two-swords. Finally, let's display the value of our productName in the view: <div class=ui info message> The product name is: {{productName}} </div> Here it looks like: Demo form with ngModel Easy! Wrapping Forms have a lot of moving pieces, but Angular makes it fairly straight. Once you get a handle on how to use FormGroups, FormControls and Validations, it's pretty easy going from there! There!