Contents

[**part 01 – Environment Setup** 2](#_Toc141340386)

[**part 02 – Form Control Validators** 5](#_Toc141340387)

[**part 03 – Form Arrays** 7](#_Toc141340388)

[**part 04 – Form Groups** 9](#_Toc141340389)

[**part 05 – Custom Validation** 11](#_Toc141340390)

[**part 06 – FormBuilder (Component)** 15](#_Toc141340391)

[**part 07 – FormBuilder (Template)** 17](#_Toc141340392)

[**part 08 – Async Validation** 18](#_Toc141340393)

[**Bonus – Form Status** 22](#_Toc141340394)

Day02 Angular Reactive Forms

# part 01 – Environment Setup

We will continue to use the same code from yesterday, however you could just start a new Angular project. I have a starter package for you in the code for today’s work. Just unzip, rename to *skills* and run npm install. It is the same project from Day01 but with the form stripped of all its template driven code. Also the component code was stripped. We will therefore start with the app.module.ts file.

The most important aspect of reactive forms in Angular is how the form itself is coded. The entire form has both an HTML and TypeScript representation. This means that both representations must be synchronized. The form itself is built using objects, so the form object and the form control objects are defined in the component. Once the component code is ready, it is synchronized in the template by accessing the various objects.

1. For this part, go to app.module.ts and either comment or delete the two statements for template driven forms and include the ones for reactive forms:

|  |
| --- |
| **import { AppComponent } from './app.component';**  **~~import { FormsModule } from "@angular/forms";~~**  **import { ReactiveFormsModule } from "@angular/forms";**  **@NgModule({**  **declarations: [**  **AppComponent**  **],**  **imports: [**  **BrowserModule,**  **~~FormsModule,~~**  **ReactiveFormsModule**  **],** |

1. In the class itself, begin the form creation process by first declaring a FormGroup:

|  |
| --- |
| **export class AppComponent {**  **userDetails! : FormGroup;**  **constructor() {** |

This FormGroup is now the most important part of the application. Use the IDE to import the FormGroup class from @angular/forms.

1. For this example, we will create and formalize the FormGroup in the ngOnInit() life-cycle phase:

|  |
| --- |
| **export class AppComponent implements OnInit{**  **userDetails! : FormGroup;**  **}** |

Once you add the highlighted code, the IDE should import the OnInit class for you, if not do it manually.

1. Add the basic structure to satisfy the OnInit interface:

|  |
| --- |
| **export class AppComponent implements OnInit{**  **userDetails! : FormGroup;**  **constructor(){}**  **ngOnInit(): void {}** |

You should also add a constructor

1. We will now initialize the FormGroup inside of this ngOnInit() method:

|  |
| --- |
| **ngOnInit(): void {**  **this.userDetails = new FormGroup({**  **firstName : new FormControl(),**  **lastName : new FormControl()**  **});** |

Once again, hover over the red squiggly line to have the IDE assist you in implementing the FormControl class.

1. Turn to the template and add the userDetails form to the <form> tag:

|  |
| --- |
| **<span id="header">Angular 3 Day 02 - Reactive Forms</span>**  **<div class="container">**  **<form [formGroup]="userDetails">**  **<div>**  **<div class="mb-3">** |

1. Still in the template connect up all the FormControls on the component to each HTML element according to what form control each tag refers to:

|  |
| --- |
| **<label for="fName" class="form-label">First name:</label>**  **<input type="text" name="fName" class="form-control" formControlName="firstName">**  **</div>**  **<div class="mb-3">**  **<label for="lName" class="form-label">Last name:</label>**  **<input type="text" name="lName" class="form-control" formControlName="lastName">**  **</div>**  **</div>** |

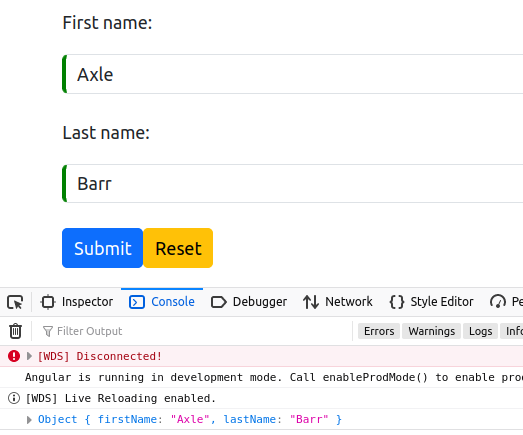
1. We will add an onSubmit() method to the component soon, but first use Angular’s binding method to hook into the form submit process:

|  |
| --- |
| **<div class="container">**  **<form [formGroup]="userDetails" (ngSubmit)="onSubmit()">**  **<div>**  **<div class="mb-3">** |

1. We had an onSubmit() method from Day01, we need to change that method a bit. Here we already have access to the form and its controls, so just print it:

|  |
| --- |
| **onSubmit(){**  **console.log(this.userDetails.value);**  **}** |

Run ng serve and do a simple test.



# part 02 – Form Control Validators

Although there are various ways to add validation, this part shows one of the easiest ways to do it. The Form Control object takes multiple arguments in its constructor, value and validation are typical. The third argument is for asynchronous validation. The Validators class must be imported from @angular/forms.

1. The configuration of validators occurs in the component, not the template. The template is used for messaging. For example, if we add these parameters, VS Code will import the Validators class and make the firstName field a required field:

|  |
| --- |
| **ngOnInit(): void {**  **this.userDetails = new FormGroup({**  **firstName : new FormControl(null, Validators.required),**  **lastName : new FormControl(),**  **email : new FormControl(),**  **department : new FormControl(),** |

Note, null represents a value to be passed to the control. In this case we pass nothing.

1. Continue with the other form controls:

|  |
| --- |
| **this.userDetails = new FormGroup({**  **firstName : new FormControl(null, Validators.required),**  **lastName : new FormControl(null, Validators.required),**  **});** |

Although Validators.required is a function, we do not execute that function only pass a reference to the function. Angular will decide when to execute the function.

1. It is possible to add multiple validation checks. Here the first name must be between 2 and 36 characters long:

|  |
| --- |
| **ngOnInit(): void {**  **this.userDetails = new FormGroup({**  **firstName : new FormControl(**  **null,**  **[Validators.required, Validators.minLength(2))]**  **),**  **lastName : new FormControl(null, Validators.required),** |

Note: when using multiple validation checks on the same control, use an array. You may repeat the same parameters for lastName, but not necessary for this tutorial.

1. Follow up by exposing the formControl with a getter method on the class:

|  |
| --- |
| **onSubmit(){**  **console.log(this.userDetails);**  **}**  **get firstName() { return this.userDetails.get('firstName')!; }** |

1. Up to this point, we are logging the **value** of our userDetails form (Part01 #8-9). In order to see the amount of detail in that form, we should change what is being logged back to the form itself and not the value:

|  |
| --- |
| **onSubmit(){**  **console.log(this.userDetails.~~value~~);  }** |

So, remove the .value part of the statement.

1. This change will show lots of information about the form such as *status* of each form control as well as the form itself. It also shows which form controls were *touched*.

We could use this information to add CSS classes based on the purity of the form and each individual form control. More importantly, however, is the ability to inform the user not only of the error but also what type of error is preventing the submission of the form itself.

1. Based on what is being shown in the console window, you will see each form control has a status. We can address this status in our template and decide whether to show a message or not:

|  |
| --- |
| **class="form-control"**  **formControlName="firstName"**  **>**  **<div \*ngIf="firstName.invalid && (firstName.dirty || firstName.touched)">**    **</div>**  **</div>** |

If the user did not comply with the firstName requirements, they will end up in this <div> tag where we can investigate the problem further.

1. Here we check if there is a value:

|  |
| --- |
| **formControlName="firstName"**  **>**  **<div \*ngIf="firstName.invalid && (firstName.dirty || firstName.touched)">**  **<div \*ngIf="firstName.errors.required">**  **Name is required.**  **</div>**  **</div>**  **</div>**  **<div class="mb-3">** |

1. Here we add an additional check based on length of the name entered:

|  |
| --- |
| **<div \*ngIf="firstName.invalid && (firstName.dirty || firstName.touched)">**  **<div \*ngIf="firstName.errors.required">**  **Name is required.**  **</div>**  **<div \*ngIf="firstName.errors.minlength">**  **Name is too short.**  **</div>**  **</div>** |

You may test the form with the two validation requirements and add more. Of course, you can transfer this knowledge to any other form control you add.

# part 03 – Form Arrays

In order to make this form more like the one in the template-driven example, add a few more fields.

1. Add an email and department key (name) to the FormGroup object. Also to avoid problems moving forward remove the validation from firstName for now:

|  |
| --- |
| **firstName : new FormControl(  ~~null,   [Validators.required,   Validators.minLength(3)]~~  ),**  **lastName : new FormControl(),**  **email : new FormControl(),**  **department : new FormControl(),**  **});**  **} firstName : new FormControl(** |

1. Now, for prizes, we can add a FormArray. To do this just use the new operator and inside of an array structure, add FormControls, again with the new operator:

|  |
| --- |
| **firstName : new FormControl(null, [Validators.required, Validators.minLength(3)]),**  **lastName : new FormControl(),**  **email : new FormControl(),**  **department : new FormControl(),**  **prizes : new FormArray([**  **new FormControl(null),**  **new FormControl(null),**  **new FormControl(null),**  **])**  **});**  **}** |

Use the IDE’s *quick fix* to import the FormArray class.

1. We can wrap up this section with getter methods for all the form controls we just added. Lets start with the email and department:

|  |
| --- |
| **onSubmit(){**  **console.log(this.userDetails);**  **}**  **//get firstName() { return this.userDetails.get('firstName')!; }**  **get email() { return this.userDetails.get('email')!; }**  **get department() { return this.userDetails.get('department')!; }**  **}** |

You can remove the firstName getter, we will be using a inner FormGroup for that.

1. If you remove the validation code then you must remove the corresponding HTML on the template. You can of course use a block comment to remove the code from being part of the app:

|  |
| --- |
| **>**    **<!-- ~~<div \*ngIf="firstName.invalid && (firstName.dirty || firstName.touched)">~~**  **~~<div \*ngIf="firstName.errors.required">~~**  **~~Name is required.~~**  **~~</div>~~**  **~~<div \*ngIf="firstName.errors.minlength">~~**  **~~Name is too short.~~**  **~~</div>~~**    **~~</div>~~ -->** |

1. When returning the prizes control, remember it is now a FormArray, so it is handled a bit differently:

|  |
| --- |
| **get department() { return this.userDetails.get('department')!; };**  **get allPrizes(){**  **return this.userDetails.get('prizes') as FormArray;**  **}** |

# part 04 – Form Groups

It is possible to have nested form groups. In this section we will add both the first and last names into a nested form group called fullName. In Part05 we will add a specific validation for this group.

1. For form groups we can start with the template or the class. Here we will start with the class. Simply create a new FormGroup object and insert the FormControls you wish to be a part of that group:

|  |
| --- |
| **ngOnInit(): void {**  **this.userDetails = new FormGroup({**  **fgFullName: new FormGroup({**  **}),**  **firstName : new FormControl(** |

Notice that it already has the object to pass name/value pairs, the set of curly braces.

1. Now just cut the firstName and lastName controls from where they are and paste them into the new FormGroup:

|  |
| --- |
| **ngOnInit(): void {**  **this.userDetails = new FormGroup({**  **fgFullName : new FormGroup({**  **firstName : new FormControl(),**  **lastName : new FormControl(),**  **}),**  **email : new FormControl(),**  **department : new FormControl(),**  **prizes : new FormArray([**  **new FormControl(null),**  **new FormControl(null),**  **new FormControl(null),**  **])**  **});**  **}** |

Now we have a new FormGroup and its name is fgFullName

1. Back on the template, we just need to wrap our two fields, first and last names into a pair of <div> tags so that we can apply our new FormGroup to that pair:

|  |
| --- |
| **<form [formGroup]="userDetails" (ngSubmit)="onSubmit()">**  **<div formGroupName = 'fgFullName' class="mb-3">**  **<div class="mb-3">**  **<label for="fName" class="form-label">First name:</label>**  **<input**  **type="text"**  **name="fName"**  **class="form-control"**  **formControlName="firstName"**  **>**  **</div>**  **<div class="mb-3">**  **<label for="lName" class="form-label">Last name:</label>**  **<input**  **type="text"**  **name="lName"**  **class="form-control"**  **formControlName="lastName"**  **>**  **</div>**  **</div>** |

I also removed all validation code so that there is better understanding of the more important pieces of the form. Notice the formGroupName.

1. Remember from Part02 #4 and Part03 #3 we added *getter* methods for the various controls. Here you can return the entire fgFullname FormGroup or the individual parts:

|  |
| --- |
| **}**  **get firstName() {**  **return this.userDetails.get('fgFullName').get('firstName');**  **};** |

1. Follow that logic for the last name:

|  |
| --- |
| **get firstName() {**  **return this.userDetails.get('fgFullName').get('firstName');**  **};**  **get lastName() {**  **return this.userDetails.get('fgFullName').get('lastName');**  **};** |

So, we get the individual group members via the group itself

1. Now we can re-introduce validation for the form controls in the .ts file. You can refer to Part02 #2 for the previous component code:

|  |
| --- |
| **ngOnInit(): void {**  **this.userDetails = new FormGroup({**  **fgFullName : new FormGroup({**  **firstName : new FormControl(null, Validators.required),**  **lastName : new FormControl(null, Validators.required),**  **}),**  **email : new FormControl(),** |

1. In Part03 #4 we removed the code on the template for validation, we can now replace that code, but with the FormGroup taken into consideration for first and last names:

|  |
| --- |
| **class="form-control"**  **formControlName="firstName"**  **>**  **<div \*ngIf="firstName.invalid && (firstName.dirty || firstName.touched)">**  **<div \*ngIf="firstName.errors.required">**  **Name is required.**  **</div>**  **<!-- <div \*ngIf="firstName.errors.minlength">**  **Name is too short.**  **</div> -->**  **</div>** |

Note, I wrapped the minLength validation in comments as we have not yet added minLength on the component. We can do it later. You may of course follow this coding for the last name field. Also, I did not have to use the group name on the template side because I exported the firstName object via the getter method on the class.

# part 05 – Custom Validation

Although there is a pattern validator, this part is **not** an example of that type of validator. With this validator we are using the normal regular expressions in a test format to check the presence of a pattern. Our example will simply check that the first name field contains only alphabetic characters.

1. First, in the app.component.ts file, import the AbstractControl class from @angular/forms:

|  |
| --- |
| **import { Component, OnInit } from '@angular/core';**  **import { FormControl, FormGroup, NgForm, Validators } from '@angular/forms';**  **import { FormArray, AbstractControl } from '@angular/forms';** |

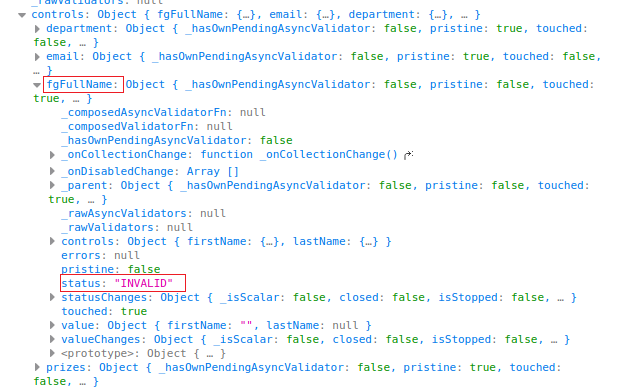
1. Begin the function that will perform the custom validation:

|  |
| --- |
| **get allPrizes(){**  **return this.userDetails.get('prizes') as FormArray;**  **}**  **alphaCheck(control: AbstractControl): void {**  **}**  **}** |

AbstractControl is a super class of the Control class.

1. Before continuing, submit the current form **with a known error** in it. For example submit the form without entering any values in the first name field. Of course this violates the *required* check. In the console window, find this control and look at the pattern of its error value:  
     
   

Notice the *name:value* pattern. Also notice that there is now an fgFullName pointer. That name points to a value made up of first and last names.

1. Continuing on, if we drill down further into fgFullName we see that the entire group has a status of invalid.  
   

Notice the Controls part, we can drill down even further.

1. Once we get to firstName itself, we can see the errors object:



1. If we enter a value for the first name field and submit, we see a different message in the errors part for the first name object:  
   

In this case notice that pristine is *false*, status is *valid* and the value is *Axle* as expected. However, notice that the errors value is *null*, NOT false, but NULL. This is important for comparison code.

1. Continuing with our new validation, let’s begin with what we know. We also know that one option to return is the *null* value. The only other possible value to be returned is some kind of *name: value* pair. Add in both possibilities now:

|  |
| --- |
| **alphaCheck(control: AbstractControl): { [key: string]: boolean } | null {**  **return null;**  **}** |

Note: I added in the :void so that the error goes away. The above highlighted code replaces the :void.

1. We also know that we want to match a certain pattern, so indicate this with a const value.

|  |
| --- |
| **alphaCheck(control: AbstractControl): { [key: string]: boolean } | null {**  **const regExp : RegExp = /^[A-Za-z]+$/;**    **return null;**  **}** |

This is a very simple pattern, it just means no numbers are allowed.

1. Next we grab the value the user gave us, from our control:

|  |
| --- |
| **alphaCheck(control: AbstractControl): { [key: string]: boolean } | null {**  **const regExp : RegExp = /^[A-Za-z]+$/;**  **const cValue = control.value;**  **return null;**  **}** |

1. Finally we can do the check and return the pattern that our form control expects, in the case of a violation:

|  |
| --- |
| **alphaCheck(control: AbstractControl): { [key: string]: boolean } | null {**  **const regExp : RegExp = /^[A-Za-z]+$/;**  **const cValue = control.value;  if (!regExp.test(cValue)) {**  **return { alphaCheck: true };**  **}**  **return null;**  **}** |

We have now satisfied all the conditions of creating a custom validation. If the control is valid, we return null and if it is not valid we return an object where the name is alphaCheck and the value is true.

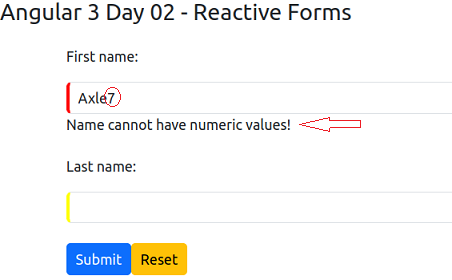
1. All we need to do now is add this function to our Validators array. If you do not already have an array, create one. Also add in the minLength() check if you want:

|  |
| --- |
| **this.userDetails = new FormGroup({**  **fgFullName : new FormGroup({**  **firstName : new FormControl(**  **null,**  **[Validators.required, Validators.minLength(3), this.alphaCheck]**  **),**  **lastName : new FormControl(null, Validators.required),**  **}),**  **email : new FormControl(),** |

1. Then add the necessary HTML to complete the task:

|  |
| --- |
| **<div class="mb-3">**  **<label for="fName" class="form-label">First name:</label>**  **<input**  **type="text"**  **name="fName"**  **class="form-control"**  **formControlName="firstName"**  **>**  **<div \*ngIf="firstName.invalid && (firstName.dirty || firstName.touched)">**  **<div \*ngIf="firstName.errors.required">**  **Name is required.**  **</div>**  **<div \*ngIf="firstName.errors.minlength">**  **Name is too short.**  **</div>**  **<div \*ngIf="firstName.errors.alphaCheck">**  **Name cannot have numeric values!**  **</div>**  **</div>** |

Here, I also added in the minLength check



# part 06 – FormBuilder (Component)

The FormBuilder class acts like an API which makes the development of reactive forms a bit easier. It is not necessary, but it does shorten the code and may even save time. It is especially useful if you do a lot of dynamic form creation operations. Since the Class itself is *injected*, using the FB class negates the use of the new keyword.

Changes were made in the template. For this part, I removed the messaging <div> and <span> tags from below the firstName field. Only the first and last name fields remain.

In the component, I remove all the functions except the onSubmit() method.

There is a zipped file called app-d2-part06-starter.zip that contains these changes.

This is only the src folder, everything else remails the same.

1. First import the FormBuilder class from @angular/forms:

|  |
| --- |
| **import { Component, OnInit } from '@angular/core';**  **import { FormControl, FormGroup, FormBuilder } from '@angular/forms';** |

1. Inject the service via the constructor:

|  |
| --- |
| **constructor(private fb: FormBuilder) {**  **};**  **onSubmit(userDetails : FormGroup){** |

This is typical of using Services in Angular.

1. The FormBuilder class has a group() method in which we simply pass the structure of the form we want to build, using an object of course:

|  |
| --- |
| **export class AppComponent {**  **userDetails = this.fb.group({**  **firstName: [''],**  **lastName: [''],**  **email: [''],**  **department:['']**  **});**  **constructor(private fb: FormBuilder) {** |

Notice that the fb object is used before the constructor.

1. Check the app.module.ts file, if the FormsModule is commented out, remove the comments and make sure the FormsModule is also part of the imports array:

|  |
| --- |
| ***... other code***  **import { AppComponent } from './app.component';**  **import { ReactiveFormsModule } from "@angular/forms";**  ***... other code***  **imports: [**  **BrowserModule,**  **ReactiveFormsModule,**  **],**  ***... other code***  **})**  **export class AppModule { }** |

1. At this point we can test what we have, just enter a few details on the form and submit it. However, it might be better to change the onSubmit() function to print just the values entered and not the entire form:

|  |
| --- |
| **onSubmit(userDetails : FormGroup){**  **console.log(this.userDetails.value);**  **}** |

Note, there may be a formControlName error in the console window, we can ignore this for now. Before you submit, check that the <form> tag has the value of userDetails inside of the onSubmit() function. If you submit the form, you will get results for email and department even though these fields are not yet part of the HTML form, only the FormGroup.

1. Complete the form tag like in previous situations:

|  |
| --- |
| **<div class="container">**  **<form [formGroup]="userDetails" (ngSubmit)="onSubmit(userDetails)">**  **<div class="mb-3">** |

1. We handle validation in the normal way; first import the Validators class then implement the various validators #6:

|  |
| --- |
| **import { Component } from '@angular/core';**  **import { FormGroup, FormBuilder, Validators } from '@angular/forms';**  **@Component({** |

1. After importing the Validators class, implement the various validators like required and minlength:

|  |
| --- |
| **userDetails = this.fb.group({**  **firstName: ['', Validators.required],**  **lastName: ['', Validators.required],**  **email: ['', Validators.required],**  **department:['']**  **});** |

# part 07 – FormBuilder (Template)

The template is almost exactly like it was before.

1. We can start adding details to declare the form, this is the same as Part01 #8:

|  |
| --- |
| **<span id="header">Angular 3 Day 02 - Reactive Forms</span>**  **<div class="container">**  **<form [formGroup]="userDetails" (ngSubmit)="onSubmit(userDetails)">**  **<div class="mb-3">** |

1. Next, we synchronize the template, we already have formControlName attributes in place:

|  |
| --- |
| **class="form-control"**  **formControlName="firstName"**  **>**  **<div \*ngIf = "userDetails?.get('firstName')?.errors?.required">**  **First name is required!**  **</div>**  **</div>**  **<div class="mb-3">** |

1. Just like before, we can have child groups:

|  |
| --- |
| **export class AppComponent{**  **userDetails = this.fb.group({**  **fullName : this.fb.group({**  **firstName: ['', Validators.required],**  **lastName: ['', Validators.required],**  **}),**  **email: ['', Validators.required],**  **department:['', Validators.required]**  **});**  **constructor(private fb: FormBuilder){}** |

1. If we did this, the template will break. Below is how we target a form control that is part of an inner grouping:

|  |
| --- |
| **formControlName="firstName"**  **>**  **<div \*ngIf = "userDetails?.get('fullName')?.get('firstName')?.errors?.required">**  **First name is required!**  **</div>**  **</div>**  **<div class="mb-3">** |

1. The above won’t work as is, we need to wrap the two fields that make up fullName into its own div:

|  |
| --- |
| **<div class="mb-3" formGroupName = 'fullName'>**  **<label for="fName" class="form-label">First name:</label>**  **<input type="text" class="form-control" formControlName="firstName">**  **<div**  **\*ngIf =** |

This is as far as we go with the FormBuilder class. As you can see we develop the code exactly like before. The class simply makes life a bit easier.

# part 08 – Async Validation

Anytime you hear the word async, always think about time and how time may affect your code. Sometimes we may have to access a service that is outside the system we control; this may lead to our code waiting on an answer from the supplier. There is a mechanism to handle situations like this. This mechanism is a protective measure that protects our app from crashing. A crash may happen if the answer is delayed or never comes.

1. First import the AbstractControl class from @angular/forms:

|  |
| --- |
| **import { Component } from '@angular/core';**  **import { FormGroup, FormBuilder, Validators, AbstractControl } from '@angular/forms';** |

Also verify that Validators is also being imported.

1. We now write the custom function to validate an email. The validator function takes a form control but it returns either a Promise or an Observable object:

|  |
| --- |
| **restrictEmail(control : AbstractControl) : Promise<any> | Observable<any>{**  **};** |

In our example we return a Promise not an Observable. You will need to import the Observable class from a third-party library, **rxjs**.

1. Add this function as a validator. You will need to put your existing required validation for email, into an array and add this one:

|  |
| --- |
| **}),**  **email: ['', [Validators.required, this.restrictEmail]],**  **department:['', Validators.required]**  **});** |

Remember to just use a pointer to the restrictEmail() function, do not execute the function.

1. Since we are supposed to return a Promise, let’s start that process:

|  |
| --- |
| **restrictEmail(control : AbstractControl) : Promise<any> |**  **Observable<any>{**  **return new Promise(()=>{**  **});**  **}** |

This is the same as declaring a new object using the new keyword, then using the return keyword to return the object itself. In this code we do **both** processes in one statement. We create and return the object in one step.

1. The Promise’s constructor takes two parameters to handle both cases of a Promise. Either the Promise was successful or not, so one parameter for *success* and one for *errors*:

|  |
| --- |
| **restrictEmail(control : AbstractControl) : Promise<any> |**  **Observable<any>{**  **return new Promise((resolve, reject)=>{**  **});**  **}** |

These parameters are functions themselves.

1. At this point we can just use the resolve() function to return an answer:

|  |
| --- |
| **restrictEmail(control : AbstractControl) : Promise<any> |**  **Observable<any>{**  **return new Promise((resolve, reject)=>{**  **resolve({restrictEmail:false});**  **});**  **}** |

Notice that the message (parameter) is wrapped in an object just like Part05 when we did the alphaCheck() function.

1. The above code does not tell the whole story, what about if the answer is NOT false. In that case we should return null, but based on what? Well, here is where we have to talk to the service (class) to find out if a particular email is allowed or not, so lets use the value of the control and ask the service:

|  |
| --- |
| **restrictEmail(control : AbstractControl) : Promise<any> |**  **Observable<any>{**  **return new Promise((resolve, reject)=>{**  **if(control.value === "axle@skillsoft.com")**  **resolve({restrictEmail:false});**  **else**  **resolve(null);**  **});**  **}** |

So, the email in the if statement is restricted, users cannot enter that email.

1. The above code still does not tell the time story. In other words we have not demonstrated asynchronous behavior. Since we do not have an actual service to connect to, let’s simulate such a service. We know that the service will take about 2.5 seconds to respond, so we can wrap our if statement into a setTimeout() function to **force** it to take 2.5 seconds to respond:

|  |
| --- |
| **restrictEmail(control : AbstractControl) :**  **Promise<any> |**  **Observable<any>{**  **return new Promise((resolve, reject)=>{**  **setTimeout(()=>{**  **if(control.value === "axle@skillsoft.com")**  **resolve({restrictEmail:false});**  **else**  **resolve(null);**  **}, 2500);**  **});**  **}** |

Now the answer we are waiting for will take exactly 2.5 seconds to return. This means that after that amount of time, either we receive a restrictEmail message or a null.

1. At this point you may not have template code for the email or department fields. Lets add at least the email field so that we can enter a value and add validation:

|  |
| --- |
| **<div class="mb-3">**  **<label for="email" class="form-label">Email:</label>**  **<input**  **type="email"**  **name="email"**  **class="form-control"**  **formControlName="email"**  **>**  **</div>**  **<div \*ngIf = "userDetails?.get('email')?.errors?.restrictEmail == false">**  **This email cannot be used!**  **</div>** |

Notice that the validation message is based on the restrictEmail() method which returns false if the user entered that restricted email.

1. In order to test how it works so far, you would need to open the Developer’s Console and look at the Inspector window in Firefox or Elements tab in Chrome. The images below show the changes in the status of the email control as we go from one stage to the next:



# Bonus – Form Status

It is possible to ‘listen’ for changes in any of the form controls or the entire form itself. If you want to implement such a listener, the ngOnInit() method is a good place to do this.

1. First implement the NgOnInit interface by importing the it and adding the method:

|  |
| --- |
| **import { Component, OnInit } from '@angular/core';**  **...**  **export class AppComponent implements OnInit {**  **...**  **ngOnInit(): void {}** |

These lines of code must be present.

1. Inside of the ngOnInit() method, we can start coding the listener:

|  |
| --- |
| **ngOnInit(): void {**  **this.userDetails?.get("fullName.firstName")?.valueChanges;**  **}** |

We have used code like this before, firs use the get() method to return the form control from the form group, then access its properties and methods.

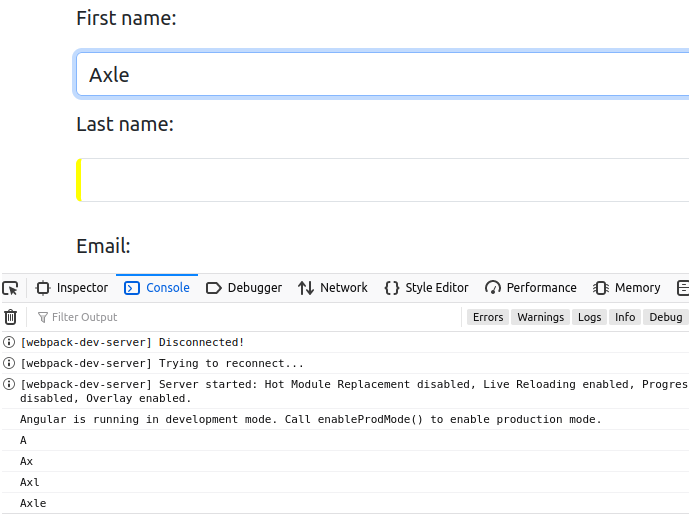
1. In the case of a formControl, there is a valueChanges property that returns an Observable. All we have to do is subscribe to this observable and provide the required function

|  |
| --- |
| **ngOnInit(): void {**  **this.userDetails?.get("fullName.firstName")?.valueChanges.subscribe();**  **}** |

1. We can use that function to handle any change to the control. In this case we simply log the changes, but in the future you may want to perform some other task based on the change

|  |
| --- |
| **ngOnInit(): void {**  **this.userDetails?.get("fullName.firstName")?.valueChanges.subscribe(selectedValue => {**  **console.log(selectedValue);**  **})**  **}** |

Notice that selectedValue is a variable that I created to hold the value being returned from the subscription.

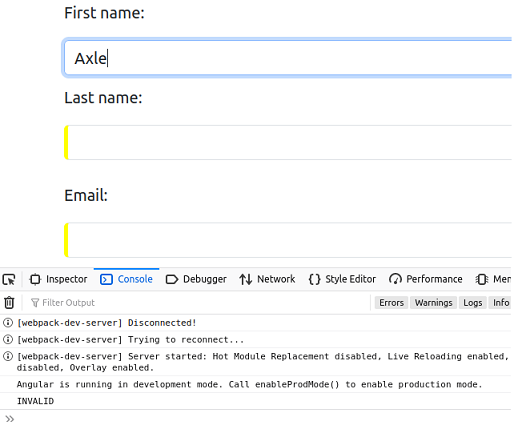


J

In the image, the name Axle has four letters, so for four times, the valueChanges event occurred and was captured by the subscription.

1. We can also listen for a change in status of a control or the entire form in this case. To see the effect of this change in code, comment the previous line of code and insert the following:

|  |
| --- |
| **ngOnInit(): void {**  **/\* this.userDetails?.get("fullName.firstName")?.valueChanges.subscribe(selectedValue => {**  **console.log(selectedValue);**  **}); \*/**  **//**  **this.userDetails?.statusChanges.subscribe(selectedValue => {**  **console.log(selectedValue);**  **});**  **}** |

1. If you test this code using firstName field, you will see it reports on the status of the entire form. Here, although the firstName field is satisfied, the entire form itself is not valid. It becomes valid once ALL the fields are entered correctly.
2. What is interesting about this status change is with the field that has the asynchronous operation on it. If you start typing something into the Email field, you will see the status change from pending to valid or invalid depending on the email you enter:



1. You can also use setValue and patchValue just like we did in the template driven form:

|  |
| --- |
| **this.userDetails?.statusChanges.subscribe(selectedValue => {**  **console.log(selectedValue);**  **});**  **//**  **this.userDetails.patchValue({**  **fullName : {**  **firstName: "your first name here",**  **lastName: "your last name here",**  **}**  **});**  **}** |

1. For a final thought, it is a good idea to reset your form. There is a reset() method associated with the form itself, here is an example of where to use it:

|  |
| --- |
| **onSubmit(userDetails : FormGroup){**  **console.log(this.userDetails);**  **this.userDetails.reset();**  **}** |