----------------------------------------------------------------Example 1------------------------------------------------------------

Angular 6 routing tutorial:

**Step 1 : Import the Angular RouterModule into the AppRoutingModule and configure the application routes :**Here is the modified AppRoutingModule file (app-routing.module.ts). Please note that, the CommonModule is not required in the routing module, so I have deleted it's reference. We generally don't declare components in the routing module so, I also deleted declarations array from @NgModule decorator. The rest of the code is commented and self-explanatory.

import { NgModule } from '@angular/core';

import { RouterModule, Routes } from '@angular/router';

// Import the components so they can be referenced in routes

import { CreateEmployeeComponent } from './employee/create-employee.component';

import { ListEmployeesComponent } from './employee/list-employees.component';

// The last route is the empty path route. This specifies

// the route to redirect to if the client side path is empty.

const appRoutes: Routes = [

  { path: 'list', component: ListEmployeesComponent },

  { path: 'create', component: CreateEmployeeComponent },

  { path: '', redirectTo: '/list', pathMatch: 'full' }

];

// Pass the configured routes to the forRoot() method

// to let the angular router know about our routes

// Export the imported RouterModule so router directives

// are available to the module that imports this AppRoutingModule

@NgModule({

  imports: [ RouterModule.forRoot(appRoutes) ],

  exports: [ RouterModule ]

})

export class AppRoutingModule { }

**Step 2 :** In the application root component file (**app.component.html**), create the navigation menu and tie the configured routes to it. The directive tells the router where to display routed views. 

<div class="container">

    <nav class="navbar navbar-default">

        <ul class="nav navbar-nav">

            <li>

                <a routerLinkActive="active" routerLink="list">List</a>

            </li>

            <li>

                <a routerLinkActive="active" routerLink="create">Create</a>

            </li>

        </ul>

    </nav>

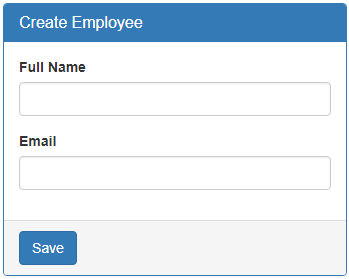
    <router-outlet></router-outlet>

</div>

As the name implies, Template Driven Forms are heavy on the template meaning we create the form completely in HTML. Template driven forms are easy to build and understand. They are great for creating simple forms. However, creating complex forms using template driven approach is not recommended as the HTML can get very complicated and messy. It is not easy to unit test template forms as the logic is in the HTML.   
  
----------------------------------------------------------------Example 2------------------------------------------------------------

**There are 2 ways to create forms in Angular** 

1. Template Driven Forms
2. Reactive Forms (Also called Model Driven Forms)

**Reactive forms**on the other hand allow us to build the form completely in code. This is more flexible and has many benefits over template forms. For example, it is easy to add form input elements dynamically and adjust validation at run-time based on the decisions made in code. It is also easy to unit test as most of the logic and validation is in the component class. The only downside of reactive forms is that they require more code than template forms.   
  
In this video and in our upcoming videos we will discuss everything we need to know to build complex reactive forms.   
  
With a reactive form, we create the entire form control tree in the component class code. Let us understand this by creating a simple form with just 2 form controls as shown below.   
  
   
  
**Creating a form group model :**Two classes that we commonly use to create a form control tree is **FormGroup**and **FormControl**. As the names imply to create a form with a group of controls, we create an instance of FormGroup class and to create each input element i.e a form control, we create an instance of FormControl class. So in the CreateEmployeeComponent (create-employee.component.ts) class modify the code as shown below. 

import { Component, OnInit } from '@angular/core';

// Import FormGroup and FormControl classes

import { FormGroup, FormControl } from '@angular/forms';

@Component({

  selector: 'app-create-employee',

  templateUrl: './create-employee.component.html',

  styleUrls: ['./create-employee.component.css']

})

export class CreateEmployeeComponent implements OnInit {

  // This FormGroup contains fullName and Email form controls

  employeeForm: FormGroup;

  constructor() { }

  // Initialise the FormGroup with the 2 FormControls we need.

  // ngOnInit ensures the FormGroup and it's form controls are

  // created when the component is initialised

  ngOnInit() {

    this.employeeForm = new FormGroup({

      fullName: new FormControl(),

      email: new FormControl()

    });

  }

}

Right click and go to the definition on **FormGroup**class constructor. Notice it has 3 parameters. 

constructor(controls: { [key: string]: AbstractControl;},

validatorOrOpts?: ValidatorFn | ValidatorFn[] | AbstractControlOptions | null,

asyncValidator?: AsyncValidatorFn | AsyncValidatorFn[] | null);

The first parameter (**controls**) is required, but the rest of the 2 parameters are optional. As you can see, the controls parameter is used to pass the collection of child controls. In our case we want 2 child controls in the FormGroup - fullName and email. So we pass an object with key/value pairs. The key is the name for the control and the value is an instance of the FormControl class. But, wait a minute, from the intellisense, I see that the value is AbstractControl and not FormControl. 

constructor(controls: { [key: string]: AbstractControl;}

So the obvious question that comes to our mind is, **how are we able to pass a FormControl instance when it is expecting AbstractControl instance.**   
  
Well, FormControl class inherits from AbstractControl class. This allows us to pass FormControl instance as the value. Both FormControl and FormGroup classes inherit from AbstractControl class. This allows us to pass either a FormControl or a FromGroup instance as the value.    
  
If you are wondering, **why do we need to pass a FromGroup instance as the value**.   
  
Well, a FormGroup can have a nested FormGroup. We will discuss nested form groups in our upcoming videos.   
  
**Binding the FormGroup model and the view :**Copy and paste the following HTML in create-employee.component.html file. This is pure HTML. There is no Angular code in this HTML. We have a <form> element and 2 text input elements (for fullName and email) 

<form class="form-horizontal">

  <div class="panel panel-primary">

    <div class="panel-heading">

      <h3 class="panel-title">Create Employee</h3>

    </div>

    <div class="panel-body">

      <div class="form-group">

        <label class="col-sm-2 control-label" for="fullName">Full Name</label>

        <div class="col-sm-8">

          <input id="fullName" type="text" class="form-control">

        </div>

      </div>

      <div class="form-group">

        <label class="col-sm-2 control-label" for="email">Email</label>

        <div class="col-sm-8">

          <input id="email" type="text" class="form-control">

        </div>

      </div>

    </div>

    <div class="panel-footer">

      <button class="btn btn-primary" type="submit">Save</button>

    </div>

  </div>

</form>

Now we need to **bind the view template to the form group model**we created in the component class. For this we make use of the following 2 directives provided by Angular ReactiveFroms module. 

* formGroup
* formControlName

Here is the modified HTML. Notice the use of **formGroup**and **formControlName**directives in the <form> element and the 2 input elements. 

<form class="form-horizontal" ***[formGroup]="employeeForm"***>

  <div class="panel panel-primary">

    <div class="panel-heading">

      <h3 class="panel-title">Create Employee</h3>

    </div>

    <div class="panel-body">

      <div class="form-group">

        <label class="col-sm-2 control-label" for="fullName">Full Name</label>

        <div class="col-sm-8">

          <input id="fullName" type="text" class="form-control"***formControlName="fullName"***>

        </div>

      </div>

      <div class="form-group">

        <label class="col-sm-2 control-label" for="email">Email</label>

        <div class="col-sm-8">

          <input id="email" type="text" class="form-control" ***formControlName="email"***>

        </div>

      </div>

    </div>

    <div class="panel-footer">

      <button class="btn btn-primary" type="submit">Save</button>

    </div>

  </div>

</form>

**Please note :** 

* To bind the <form> element to the employeeForm group in the component class we use the formGroup directive. Since "employeeForm" is a property we use square brackets around the formGroup directive to indicate that we are binding to a property.
* To bind each input element to the associated FormControl in the FormGroup model, we use formControlName directive. Notice we are not using square brackets with formControlName directive. This is because, in this case we are binding to a form control name which is a string and not a property.

At this point, if you view the page in the browser, you will see the following error in the browser console.  
**Can't bind to 'formGroup' since it isn't a known property of 'form'**   
  
This is because, the 2 directives (formGroup and formControlName) are in ReactiveForms module, but we have not yet imported it in our root module. So in the AppModule (**app.module.ts file**), import ReactiveFormsModule and include it in the **imports**array. 

import { ReactiveFormsModule } from '@angular/forms';

**Accessing form data :**To access form data, bind to the ngSubmit event on the <form>element. This ngSubmit event is raised when a button with input type=submit is clicked. 

<form class="form-horizontal" [formGroup]="employeeForm"

      (ngSubmit)="onSubmit()">

In the component class (**create-employee.component.ts**), include **onSubmit()** method as shown below. 

onSubmit(): void {

  console.log(this.employeeForm.value);

}

At this point, fill out the form and click **Save**button. Notice, the **Formgroup**value property is logged to the console. The value property of the **FormGroup**contains each form control name and it's associated value.  
  
At the moment our reactive form is a very simple form with just 2 text box input controls. As we progress through this course will discuss working with checkboxes, radio buttons, dropdownlists etc. We will also discuss form validation, nested form groups, dynamically creating form controls etc.

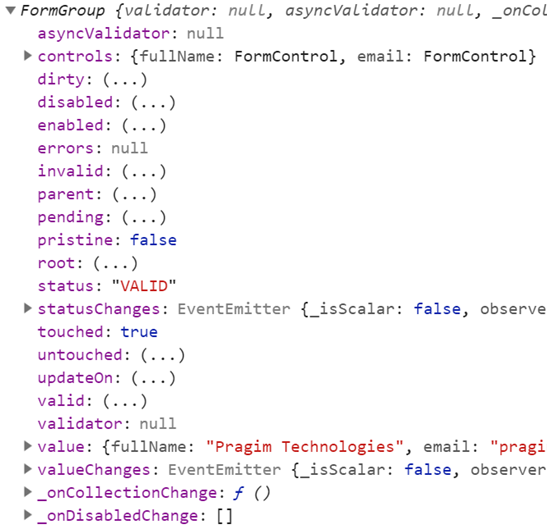
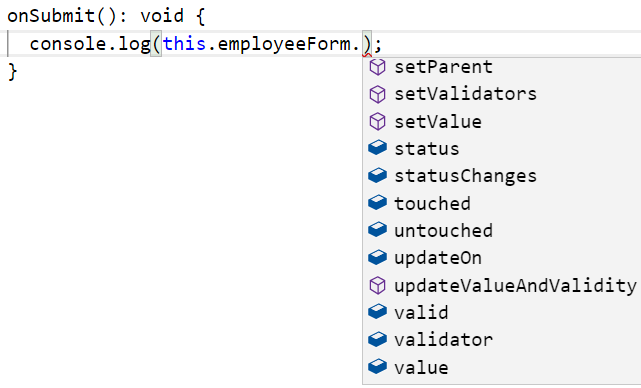
----------------------------------------------------------------Example 3------------------------------------------------------------

we will discuss **FormControl**and **FormGroup**classes

* When working with reactive forms we create instances of **FormControl**and **FormGroup**classes to create a form model.
* To bind an HTML <form> tag in the template to the **FromGroup**instance in the component class, we use **formGroup**directive
* To bind an  HTML <input> element in the template to the **FormControl**instance in the component class, we use **formControlName**directive
* **formGroup**and **formControlName**directives are provided by the **ReactiveFormsModule**
* Both **FormControl**and **FormGroup**classes inherit from **AbstractControl**base class
* The **AbstractControl**class has properties that help us track both **FormControl**and **FormGroup**value and state

**The following are some of the useful properties provided by the AbstractControl class** 

* value
* errors
* valid
* invalid
* dirty
* pristine
* touched
* untouched

**FormControl**instance tracks the value and state of the individual html element it is associated with  
**FormGroup**instance tracks the value and state of all the form controls in it's group  
  
To see the form model we created using **FormGroup**and **FormControl**classes, log the employeeForm to the console.   
  
onSubmit(): void {  
  console.log(this.employeeForm);  
}   
  
On Save button click, you should see the following form model in the browser console.   
  
   
  
To access the FormGroup properties use, **employeeForm**property in the component class. When you press DOT on the employeeForm property you can see all the available properties and methods.   
  
  
  
To access a **FormControl**in a **FormGroup**, we can use one of the following 2 ways. 

employeeForm.controls.fullName.value  
employeeForm.get('fullName').value

**Note:**This same code works, both in the template and component class.  
  
Please include the following HTML, just after the <form> tag, in the template, and you can see the property values change as you interact with the form controls on the form. 

<table border="1">

  <tr>

    <th style="padding: 10px">FormGroup</th>

    <th style="padding: 10px">FormControl (fullName)</th>

  </tr>

  <tr>

    <td style="padding: 10px">

      touched : {{ employeeForm.touched }}

      <br/> dirty : {{ employeeForm.dirty }}

      <br/> valid : {{ employeeForm.valid }}

      <br/> Form Values : {{employeeForm.value | json}}

    </td>

    <td style="padding: 10px">

      touched : {{ employeeForm.get('fullName').touched }}

      <br/> dirty : {{ employeeForm.get('fullName').dirty }}

      <br/> valid : {{ employeeForm.get('fullName').valid }}

      <br/> FullName Value : {{employeeForm.get('fullName').value}}

    </td>

  </tr>

</table>

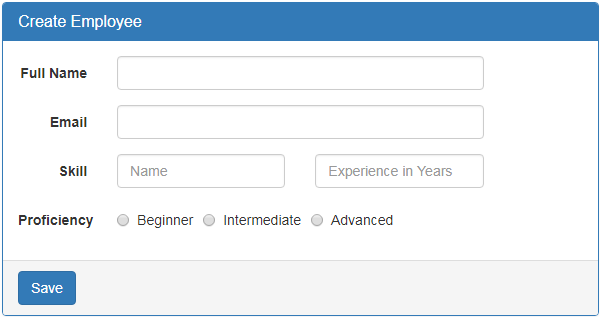
In addition to these properties, **AbstractControl**also provides the following methods. In our upcoming videos we will use these properties and methods for form validation and working with data. 

* setValidators()
* clearValidators()
* updateValueAndValidity()
* setValue()
* patchValue()
* Reset()

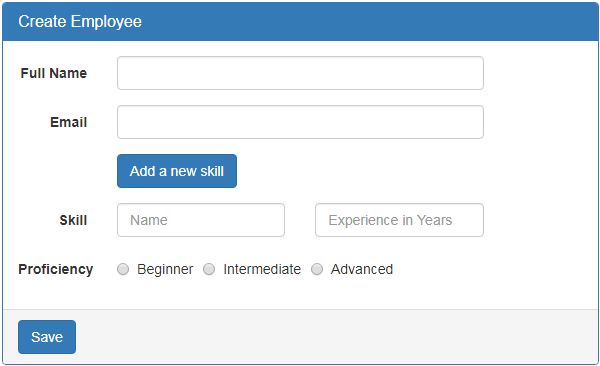
----------------------------------------------------------------Example 4------------------------------------------------------------

[Angular nested form groups](https://www.youtube.com/watch?v=XKwct01QdoI)

Let's understand, **creating nested form groups with an example**.

  
  
  
In addition to **fullName**and **email**, we want to add the following 3 fields to "Create Employee" form. 

* Skill Name
* Experience in Years
* Proficiency

What we want to be able to ultimately do is add multiple skills dynamically at run time, by clicking **"Add a new skill"** button.   
  
  
  
When the button is clicked, we want to dynamically add another set of skill related fields i.e  

* Skill Name
* Experience in Years and
* Proficiency

Also, another additional requirement is to keep "Add a new skill" button disabled, until all the skill related fields are properly filled and valid.   
  
So in short, the requirement is to dynamically create a group of form fields and also validate them as a single group so "Add a new skill" button can be enabled or disabled based on the validation state of the group. This can be very easily achieved using a nested form group. So, first let's create a nested form group for skill related fields in the component class.   
  
**Step 1: Creating a nested form group in the component class :**Form groups can accept both form control and form group instances as children. This allows us to create a nested form group. Modify the code in ngOnInit() life cycle hook as shown below. 

ngOnInit() {

  this.employeeForm = new FormGroup({

    fullName: new FormControl(),

    email: new FormControl(),

    // Create skills form group

    skills: new FormGroup({

      skillName: new FormControl(),

      experienceInYears: new FormControl(),

      proficiency: new FormControl()

    })

  });

}

Notice we have created a nested form group with key - **skills**. This nested form group contains 3 form controls. 

* skillName,
* experienceInYears and
* proficiency

**Step 2: Grouping the nested form in the template :** To group the form elements in the HTML, encapsulate the form elements in a <div> element and use the **formGroupName**directive on that container <div> element. Bind the **formGroupName**directive to the skills **FormGroup**instance in the component class. Bind each input element in the HTML, to the corresponding **FormControl**instance using the **formControlName**directive. 

<div formGroupName="skills">

  <div class="form-group">

    <label class="col-sm-2 control-label" for="skillName">

      Skill

    </label>

    <div class="col-sm-4">

      <input type="text" class="form-control" id="skillName"

          placeholder="Name" formControlName="skillName">

    </div>

    <div class="col-sm-4">

      <input type="text" placeholder="Experience in Years"

          class="form-control" formControlName="experienceInYears">

    </div>

  </div>

  <div class="form-group">

    <label class="col-md-2 control-label">Proficiency</label>

    <div class="col-md-8">

      <label class="radio-inline">

        <input id="proficiency" type="radio" value="beginner"

               formControlName="proficiency">Beginner

      </label>

      <label class="radio-inline">

        <input id="proficiency" type="radio" value="intermediate"

               formControlName="proficiency">Intermediate

      </label>

      <label class="radio-inline">

        <input id="proficiency" type="radio" value="advanced"

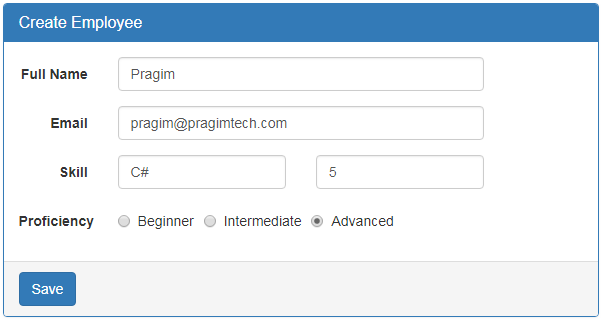
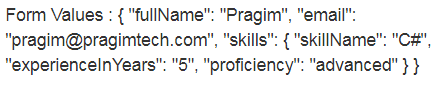
               formControlName="proficiency">Advanced

      </label>

    </div>

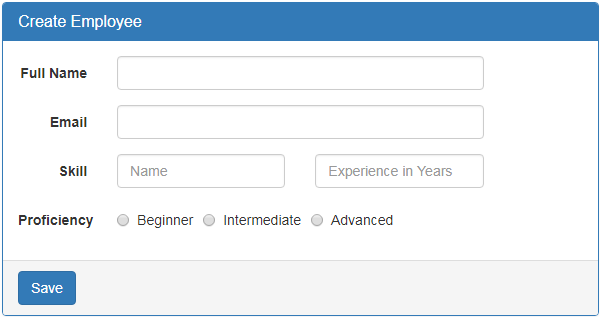
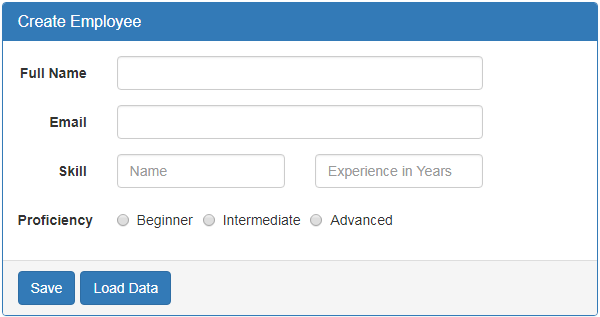
  </div>

</div>

At this point, save all the changes and when you fill out the form.   
  
   
  
skills nested formgroup value is reflected on the page.    
  
   
  
**Please note :** If you do not see the nested **formgroup**value displayed, make sure you have the following in the template after the closing <form> element.  
Form Values : {{employeeForm.value}}   
  
In our upcoming sessions we will discuss, form validation and dynamically adding form controls.

----------------------------------------------------------------Example 5------------------------------------------------------------

[Angular setvalue and patchvalue methods](https://www.youtube.com/watch?v=BRdhAPozNlI)

we will discuss **how to update HTML elements on a form with new data**.   
  
  
First let's understand **why we need to update HTML elements on a form with new data**. Let's say, we are using the form below to edit an existing employee. To be able to edit an existing employee details we have to retrieve data from a server and then update the form controls on the form with that retrieved data.   
  
  
  
  
This can be very easily achieved using **setValue()** method.   
  
Modify the HTML in **"create-employee.component.html"** file to include **"Load Data"**button   
  
  
  
I included **"Load Data"** button in the bootstrap panel footer. Please note that, I have wrapped both the buttons in a <div> element with "btn-toolbar" class so we get a space between the buttons. Otherwise, they will be joined together. 

<div class="panel-footer">

  <div class="btn-toolbar">

  <button class="btn btn-primary" type="submit">Save</button>

  <button class="btn btn-primary" type="button"

          (click)="onLoadDataClick()">Load Data</button>

</div>

In the component class, include **onLoadDataClick()** method 

onLoadDataClick(): void {

  this.employeeForm.setValue({

    fullName: 'Pragim Technologies',

    email: 'pragim@pragimtech.com',

    skills: {

      skillName: 'C#',

      experienceInYears: 5,

      proficiency: 'beginner'

    }

  });

}

At this point, when **"Load Data"** button is clicked, the form controls are updated with the form model data specified in **onLoadDataClick()** event handler.   
  
**Updating only a sub-set of HTML elements on the form :**If I want to update only a sub-set of HTML elements on the form, can I still use **setValue()** method. The answer is NO. Let's see what happens if I use **setValue()** method and try to update only **fullName**and **email**fields.  
  
Comment the code in **onLoadDataClick()** event handler as shown below. 

onLoadDataClick(): void {

  this.employeeForm.setValue({

    fullName: 'Pragim Technologies',

    email: 'pragim@pragimtech.com',

    // skills: {

    //   skillName: 'C#',

    //   experienceInYears: 5,

    //   proficiency: 'beginner'

    // }

  });

}

At this point, when **"Load Data"** button is clicked, you will see the following error in the browser developer tools.  
Must supply a value for form control with name: 'skills'   
  
If you want to update only a sub-set of form controls, then use **patchValue()** method instead of **setValue()**. 

onLoadDataClick(): void {

  this.employeeForm.patchValue({

    fullName: 'Pragim Technologies',

    email: 'pragim@pragimtech.com',

    // skills: {

    //   skillName: 'C#',

    //   experienceInYears: 5,

    //   proficiency: 'beginner'

    // }

  });

}

At this point, when **"Load Data"** button is clicked, fullName and email form controls are updated as expected.   
  
**Can I use patchValue() to update all the formControls**  
Yes, you can use patchValue() to either update all the formControls or only a sub-set of form controls. In either cases, patchValue() method succeeds without any error. setValue() on the other hand can only be used to update all the form controls. You cannot use it to update a sub-set of form controls. If you try to, you will get an error.    
  
So setValue() is very useful when we want to update all the form controls. If we accidentally miss a value for a formcontrol, setValue() fails with an error so we know we are missing something. patchValue() on the other hand silently fails without an error. So, you may not realise you have missed something, especially when you have a very large form group.  
  
So in short, **use setValue() to update all form controls and patchValue() to update a sub-set of form controls**

----------------------------------------------------------------Example 6------------------------------------------------------------

[Angular formbuilder example](https://www.youtube.com/watch?v=AiNfK36Xn1s)

will discuss an easier way of **creating reactive forms using the FormBuilder class**   
  
  
**In Angular, there are 2 ways to create reactive forms** 

1. Explicitly creating instances of FormGroup and FormControl classes using the **new**keyword. We discussed this in [Part 4](https://www.youtube.com/watch?v=VLYc3ACWL-E) and [Part 6](https://www.youtube.com/watch?v=XKwct01QdoI) of [Angular 6 tutorial](https://www.youtube.com/playlist?list=PL6n9fhu94yhWNJaDgh0mfae_9xoQ4E_Zj).
2. Using the FormBuilder class

The **FormBuilder**class provides syntactic sugar that shortens creating instances of a FormControl, FormGroup, or FormArray. It reduces the amount of code we have to write to build complex reactive forms. The FormBuilder service has three methods:  

* control() - Construct a new FormControl instance
* group() - Construct a new FormGroup instance
* array() - Construct a new FormArray instance

We will discuss FormArrays in our upcoming videos.   
  
**Step 1 :** Import FormBuilder    
  
The FormBuilder class is provided as a service, so first let's import the service 

import { FormBuilder } from '@angular/forms';

**Step 2 :** Inject the FormBuilder service    
  
Once the FormBuilder service is imported, inject it into the component using the constructor 

constructor(private fb: FormBuilder) { }

**Step 3 :**Use the FormBuilder  

* Notice in the example below, we are using the FormBuilder group() method to create a FormGroup instance.
* To the method we pass an object that contains a collection of child controls.
* For each child control we specify a key and value.
* Key is the name of the form control and the value is an array.
* The first element of the array is used to specify an initial value for the form control.
* The second and third elements of the array are used to specify synchronous and asynchronous validators for the form control. We will discuss these when we discuss form validation in our upcoming videos.
* For now, we have defined just the initial value using the first element of the array.
* We have specified an empty string as the default value for all the controls except proficiency radio buttons.
* For proficiency we have a default value of **beginner**. So the respective radio button is selected when the form loads.

this.employeeForm = this.fb.group({

  fullName: [''],

  email: [''],

  skills: this.fb.group({

    skillName: [''],

    experienceInYears: [''],

    proficiency: ['beginner']

  }),

});

FormBuilder reduces the amount of boilerplate code we have to write to build complex reactive forms.

----------------------------------------------------------------Example 7------------------------------------------------------------

[Angular reactive forms validation](https://www.youtube.com/watch?v=MPuXl1DS1vU)

**Here is what we want to do. We want to make Full Name**  
1. Required and  
2. The number of characters must be between 2 and 10   
  
  
**Step 1 - Import Angular Validators class :** The first thing to do, when implementing validation in a reactive form is to import angular's Validators class.  

import { Validators } from '@angular/forms';

**This class has the following validator functions**

|  |  |
| --- | --- |
| **Function** | **Description** |
| required | Validate that a field has a value. Used for required fields. For example, Name is required. |
| requiredTrue | Validate that the field value is true. This validator is commonly used on a required checkbox. For example, "I Agree to the terms" checkbox must be checked to submit the form. |
| email | Validate that the field value has a valid email pattern. For example, abc is not a valid email. |
| pattern | Validate that the field value matches the specified regex pattern. |
| min | Validate that the field value is greater than or equal to the provided number. For example, minimum age to vote is 18. |
| max | Validate that the field value is less than or equal to the provided number. For example, people over the age of 90 are not eligible for this insurance policy. |
| minLength | The number of characters in the field must be greater than or equal to the provided minimum length. For example, Full Name must be at least 3 characters. |
| maxLength | The number of characters in the field must be less than or equal to the provided maximum length. For example, Description cannot exceed 500 characters. |

Most of our validation requirements can be met using one or more of the following built-in validator functions. You can also write your own **custom validator**, if your requirements are not met using one of the above built-in validator functions. We will discuss, implementing a custom validator in a reactive form in our upcoming videos.   
  
**Step 2 - Specify the validators on the fullName Field :** Notice, along with a default value of empty string, we are passing an array of validator functions. In our case 3 - required, minLength and maxLength. 

this.employeeForm = this.fb.group({

  fullName: ['', [Validators.required, Validators.minLength(2), Validators.maxLength(10)]],

  // OtherFields...

});

**Step 3 - Modify the fullName field in the template to display validation error messages.**

<div class="form-group"

      [ngClass]="{'has-error': ((employeeForm.get('fullName').touched ||

                                 employeeForm.get('fullName').dirty) &&

                                 employeeForm.get('fullName').errors)}">

  <label class="col-sm-2 control-label" for="fullName">Full Name</label>

  <div class="col-sm-8">

    <input id="fullName" type="text" class="form-control"formControlName="fullName">

    <span class="help-block"

          \*ngIf="((employeeForm.get('fullName').touched ||

                   employeeForm.get('fullName').dirty) &&

                   employeeForm.get('fullName').errors)">

      <span \*ngIf="employeeForm.get('fullName').errors.required">

        Full Name is required

      </span>

      <span \*ngIf="employeeForm.get('fullName').errors.minlength ||

                   employeeForm.get('fullName').errors.maxlength">

          Full Name must be greater than 2 characters and less than 10 characters

      </span>

    </span>

  </div>

</div>

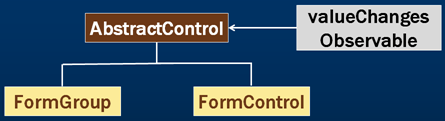
At the moment, the validation messages are still in the template HTML. In our upcoming videos, we will discuss, how to move them into the component class.

----------------------------------------------------------------Example 8------------------------------------------------------------

### Angular form control valuechanges

we will discuss **how to monitor and react when a form control or form group value changes.**  
  
  
**Angular valueChanges Observable** 

* Both FormControl and FormGroup classes inherit from the base AbstractControl class
* AbstractControl class has valueChanges property
* valueChanges property is an observable that emits an event every time the value of the control changes
* To be able to monitor and react when the FormControl or FormGroup value changes, subscribe to the valueChanges observable

   
  
  
**For example**, if you want to monitor and log to the console as the value of a **fullName**form control changes, subscribe to it's valueChanges observable as shown below. 

ngOnInit() {

  this.employeeForm = this.fb.group({

    fullName: ['',

      [

        Validators.required,

        Validators.minLength(2),

        Validators.maxLength(10)]

    ],

    email: [''],

    skills: this.fb.group({

      skillName: ['C#'],

      experienceInYears: [''],

      proficiency: ['beginner']

    }),

  });

  // Subscribe to valueChanges observable

  this.employeeForm.get('fullName').valueChanges.subscribe(

    value => {

      console.log(value);

    }

  );

}

We placed the code to subscribe to the valueChanges Observable in ngOnInit lifecycle hook. This is because, we want to start monitoring and reacting to fullName form control value immediately after the component is initialised.  
  
Every time the value of the fullName form control changes, the value is passed as a parameter to the subscribe method and the associated code is executed.  
  
Since FormGroup class also inherit from AbstractControl class, we can also subscribe to the FormGroup valueChanges observable. This allows us to monitor and react when any control value in that FormGroup changes. 

// Subscribe to FormGroup valueChanges observable

this.employeeForm.valueChanges.subscribe(

  value => {

    console.log(JSON.stringify(value));

  }

);

**Subscribing to valueChanges observable and there by monitoring a form control or form group allow us to do several things like** 

* Implementing auto-complete feature
* Dynamically validating form controls
* Move validation messages from the view template to the component class

We will discuss, how to do all of the above in our upcoming videos

----------------------------------------------------------------Example 9------------------------------------------------------------

### Loop through all form controls in formgroup in reactive form

we will discuss, **how to loop through all form controls in a formgroup including nested form groups in a reactive form.**  
  
  
**Understanding this technique is very useful**, as it can help us perform the following on all the form controls in a reactive form 

* Reset form controls
* Enable or disable all form controls or just the nested formgroup controls
* Set validators and clear validators
* Mark form controls as dirty, touched, untouched, pristine etc.
* Move validation messages and the logic to show and hide them into the component class from the view template.

Here is what we want to do : Loop through each form control in the following employeeForm form group including nested skills form group and log the form control key and value to the console. 

this.employeeForm = this.fb.group({

  fullName: [''],

  email: [''],

  skills: this.fb.group({

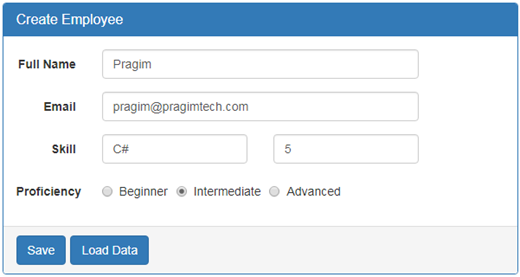
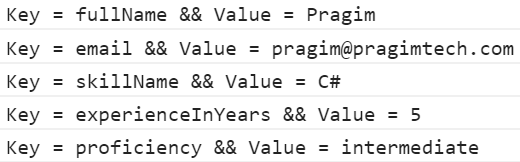
    skillName: [''],

    experienceInYears: [''],

    proficiency: ['beginner']

  }),

});

For example, if we have the following sample data on the form   
  
   
  
We want every form control key and value to be logged to the console as shown below.   
  
   
  
**Here is the method. It is commented and self-explanatory** 

logKeyValuePairs(group: FormGroup): void {

  // loop through each key in the FormGroup

  Object.keys(group.controls).forEach((key: string) => {

    // Get a reference to the control using the FormGroup.get() method

    const abstractControl = group.get(key);

    // If the control is an instance of FormGroup i.e a nested FormGroup

    // then recursively call this same method (logKeyValuePairs) passing it

    // the FormGroup so we can get to the form controls in it

    if (abstractControl instanceof FormGroup) {

      this.logKeyValuePairs(abstractControl);

      // If the control is not a FormGroup then we know it's a FormControl

    } else {

      console.log('Key = ' + key + ' && Value = ' + abstractControl.value);

    }

  });

}

**In our next video we will discuss,** moving validation messages into the component class from the view template.

----------------------------------------------------------------Example 10-----------------------------------------------------------

[Move validation messages to the component class in reactive form](https://www.youtube.com/watch?v=AvqP3FblMA8)

 we will discuss, how to **move validation messages to the component class.** There are several benefits to this 

* Easily unit test validation logic
* Instead of hard-coding validation messages in the application, we can load them from an external source like a database or a file.
* Change validation dynamically at run-time based on the decisions made in code or user selections

In this video let's move the validation messages to the component class and in our next video we will discuss moving the logic to show and hide these messages.   
  
  
**Changes in create-employee.component.ts file :**

// This object will hold the messages to be displayed to the user

// Notice, each key in this object has the same name as the

// corresponding form control

formErrors = {

  'fullName': '',

  'email': '',

  'skillName': '',

  'experienceInYears': '',

  'proficiency': ''

};

// This object contains all the validation messages for this form

validationMessages = {

  'fullName': {

    'required': 'Full Name is required.',

    'minlength': 'Full Name must be greater than 2 characters.',

    'maxlength': 'Full Name must be less than 10 characters.'

  },

  'email': {

    'required': 'Email is required.'

  },

  'skillName': {

    'required': 'Skill Name is required.',

  },

  'experienceInYears': {

    'required': 'Experience is required.',

  },

  'proficiency': {

    'required': 'Proficiency is required.',

  },

};

ngOnInit() {

  // Modify the code to include required validators on

  // all form controls

  this.employeeForm = this.fb.group({

    fullName: ['', [Validators.required,

    Validators.minLength(2), Validators.maxLength(10)]],

    email: ['', Validators.required],

    skills: this.fb.group({

      skillName: ['', Validators.required],

      experienceInYears: ['', Validators.required],

      proficiency: ['', Validators.required]

    }),

  });

}

logValidationErrors(group: FormGroup): void {

  // Loop through each control key in the FormGroup

  Object.keys(group.controls).forEach((key: string) => {

    // Get the control. The control can be a nested form group

    const abstractControl = group.get(key);

    // If the control is nested form group, recursively call

    // this same method

    if (abstractControl instanceof FormGroup) {

      this.logValidationErrors(abstractControl);

      // If the control is a FormControl

    } else {

      // Clear the existing validation errors

      this.formErrors[key] = '';

      if (abstractControl && !abstractControl.valid) {

        // Get all the validation messages of the form control

        // that has failed the validation

        const messages = this.validationMessages[key];

        // Find which validation has failed. For example required,

        // minlength or maxlength. Store that error message in the

        // formErrors object. The UI will bind to this object to

        // display the validation errors

        for (const errorKey in abstractControl.errors) {

          if (errorKey) {

            this.formErrors[key] += messages[errorKey] + ' ';

          }

        }

      }

    }

  });

}

onLoadDataClick(): void {

  this.logValidationErrors(this.employeeForm);

  console.log(this.formErrors);

}

----------------------------------------------------------------Example 11-----------------------------------------------------------

[Move validation logic to the component class in reactive form](https://www.youtube.com/watch?v=2V50gFV-BEE)

 we will discuss, **how to move the logic to show and hide validation messages from the view template into the component class**. This is continuation to [Part 12](https://www.youtube.com/watch?v=AvqP3FblMA8) where we discussed moving validation messages.   
  
  
**create-employee.component.html :**Consider the following HTML in **CreateEmployeeComponent**view template. This HTML is for the **"Full Name"** field. The logic to add or remove **has-error**class is in the template. Also the validation message and the logic to show and hide it is also in the template at the moment. 

<div class="form-group"

      [ngClass]="{'has-error': ((employeeForm.get('fullName').touched ||

                                employeeForm.get('fullName').dirty) &&

                                employeeForm.get('fullName').errors)}">

  <label class="col-sm-2 control-label" for="fullName">Full Name</label>

  <div class="col-sm-8">

    <input id="fullName" type="text" class="form-control"formControlName="fullName">

    <span class="help-block" \*ngIf="((employeeForm.get('fullName').touched ||

                                      employeeForm.get('fullName').dirty) &&

                                      employeeForm.get('fullName').errors)">

      <span \*ngIf="employeeForm.get('fullName').errors.required">

        Full Name is required

      </span>

      <span \*ngIf="employeeForm.get('fullName').errors.minlength ||

                    employeeForm.get('fullName').errors.maxlength">

        Full Name must be greater than 2 characters and less than 10 characters

      </span>

    </span>

  </div>

</div>

Now, let's move all of this into the component class. Modify the HTML as shown below. Notice, now we are binding to **formErrors.fullName**property. All the complex logic is moved to the component class. Notice the HTML here is much less than what we have had before. 

<div class="form-group" [ngClass]="{'has-error': formErrors.fullName}">

  <label class="col-sm-2 control-label" for="fullName">Full Name</label>

  <div class="col-sm-8">

    <input id="fullName" type="text" class="form-control"formControlName="fullName">

    <span class="help-block" \*ngIf="formErrors.fullName">

      {{formErrors.fullName}}

    </span>

  </div>

</div>

**Changes in create-employee.component.ts file :**The changes are commented and self-explanatory 

formErrors = {

  'fullName': '',

  'email': '',

  'skillName': '',

  'experienceInYears': '',

  'proficiency': ''

};

validationMessages = {

  'fullName': {

    'required': 'Full Name is required.',

    'minlength': 'Full Name must be greater than 2 characters.',

    'maxlength': 'Full Name must be less than 2 characters.',

  },

  'email': {

    'required': 'Email is required.'

  },

  'skillName': {

    'required': 'Skill Name is required.',

  },

  'experienceInYears': {

    'required': 'Experience is required.',

  },

  'proficiency': {

    'required': 'Proficiency is required.',

  },

};

ngOnInit() {

  this.employeeForm = this.fb.group({

    fullName: ['', [Validators.required, Validators.minLength(2), Validators.maxLength(10)]],

    email: ['', Validators.required],

    skills: this.fb.group({

      skillName: ['', Validators.required],

      experienceInYears: ['', Validators.required],

      proficiency: ['', Validators.required]

    }),

  });

  // When any of the form control value in employee form changes

  // our validation function logValidationErrors() is called

  this.employeeForm.valueChanges.subscribe((data) => {

    this.logValidationErrors(this.employeeForm);

  });

}

logValidationErrors(group: FormGroup = this.employeeForm): void {

  Object.keys(group.controls).forEach((key: string) => {

    const abstractControl = group.get(key);

    if (abstractControl instanceof FormGroup) {

      this.logValidationErrors(abstractControl);

    } else {

      this.formErrors[key] = '';

      if (abstractControl && !abstractControl.valid

          && (abstractControl.touched || abstractControl.dirty)) {

        const messages = this.validationMessages[key];

        for (const errorKey in abstractControl.errors) {

          if (errorKey) {

            this.formErrors[key] += messages[errorKey] + ' ';

          }

        }

      }

    }

  });

}

The only problem at the moment is that when a control loses focus, our validation is not triggered. This is because **valueChanges**observable does not emit an event when the control loses focus. It only emits an event when the value changes.   
  
One work around for this is to bind to the **blur**event and call validation function (logValidationErrors()) manually. 

<input id="fullName" type="text" class="form-control"

        formControlName="fullName" (blur)="logValidationErrors()">

Here is the HTML for **Email, Skill Name, Experience**and **Proficiency**input elements. 

<div class="form-group" [ngClass]="{'has-error': formErrors.email}">

  <label class="col-sm-2 control-label" for="email">Email</label>

  <div class="col-sm-8">

    <input id="email" type="text" class="form-control"

           formControlName="email" (blur)="logValidationErrors()">

    <span class="help-block" \*ngIf="formErrors.email">

      {{formErrors.email}}

    </span>

  </div>

</div>

<div class="well">

  <div formGroupName="skills">

    <div class="form-group" [ngClass]="{'has-error': formErrors.skillName}">

      <label class="col-sm-2 control-label" for="skillName">

        Skill

      </label>

      <div class="col-sm-4">

        <input type="text" class="form-control" id="skillName"formControlName="skillName"

               (blur)="logValidationErrors()" placeholder="C#, Java, Angular etc...">

        <span class="help-block" \*ngIf="formErrors.skillName">

          {{formErrors.skillName}}

        </span>

      </div>

    </div>

    <div class="form-group" [ngClass]="{'has-error': formErrors.experienceInYears}">

      <label class="col-sm-2 control-label" for="experienceInYears">

        Experience

      </label>

      <div class="col-sm-4">

        <input type="text" class="form-control" id="experienceInYears"

               formControlName="experienceInYears" placeholder="In Years"

              (blur)="logValidationErrors()">

        <span class="help-block" \*ngIf="formErrors.experienceInYears">

          {{formErrors.experienceInYears}}

        </span>

      </div>

    </div>

    <div class="form-group" [ngClass]="{'has-error': formErrors.proficiency}">

      <label class="col-md-2 control-label">Proficiency</label>

      <div class="col-md-8">

        <label class="radio-inline">

          <input type="radio" value="beginner" formControlName="proficiency"

                 (blur)="logValidationErrors()">Beginner

        </label>

        <label class="radio-inline">

          <input type="radio" value="intermediate" formControlName="proficiency"

                 (blur)="logValidationErrors()">Intermediate

        </label>

        <label class="radio-inline">

          <input type="radio" value="advanced" formControlName="proficiency"

                 (blur)="logValidationErrors()">Advanced

        </label>

        <span class="help-block" \*ngIf="formErrors.experienceInYears">

          {{formErrors.proficiency}}

        </span>

      </div>

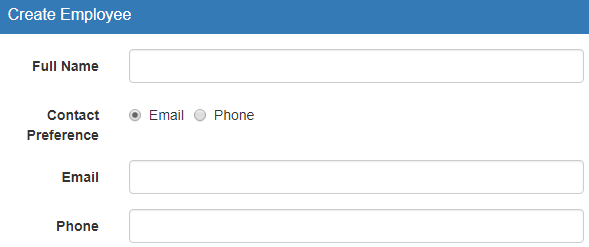
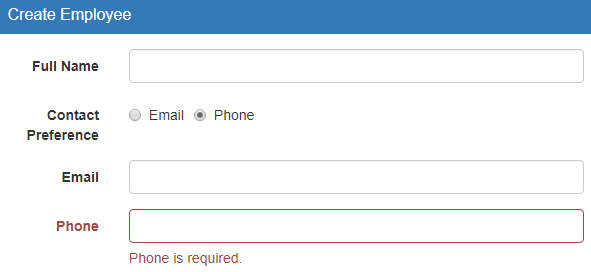
    </div>

  </div>

</div>

----------------------------------------------------------------Example 12-----------------------------------------------------------

[Dynamically adding or removing form control validators in reactive form](https://www.youtube.com/watch?v=BP6E4rW5EPA)

we will discuss **how to add or remove validators dynamically at runtime**.   
  
  
Let us understand this with an example. To start with, **Phone filed is optional.**  
  
  
  
  
However, if we select "Phone" as the contact preference, then it should become a required field.   
  
  
  
**So here is our requirement**  

* **Add the "required"** validator to the Phone form control when the user selects "Phone" as their contact preference
* On the other hand, **remove the "required"** validator from the Phone form control, when the user selects "Email" as their contact preference
* So on the "Phone" form control, we have to dynamically add or remove the required validator function

This can be very easily achieved using the following 3 functions 

* setValidators()
* clearValidators()
* updateValueAndValidity()

These methods are available in the **AbstractControl**class. Since **FormControl**inherits from **AbstractControl**, these methods are also available to **FormControl**class.   
  
**Here is the HTML** 

<!-- Notice the click event handler on both the radio buttons. When "Email"

radio button is clicked "email" string is passed to the event handler

function. Similarly, when "Phone" radio button is clicked "phone"

string is passed to the event handler function -->

<div class="form-group">

  <label class="col-md-2 control-label">Contact Preference</label>

  <div class="col-md-8">

    <label class="radio-inline">

      <input type="radio" value="email" formControlName="contactPreference"

              (click)="onContactPrefernceChange('email')">Email

    </label>

    <label class="radio-inline">

      <input type="radio" value="phone" formControlName="contactPreference"

              (click)="onContactPrefernceChange('phone')">Phone

    </label>

  </div>

</div>

<!-- Email input element -->

<div class="form-group" [ngClass]="{'has-error': formErrors.email}">

  <label class="col-sm-2 control-label" for="email">Email</label>

  <div class="col-sm-8">

    <input id="email" type="text" class="form-control"

            formControlName="email" (blur)="logValidationErrors()">

    <span class="help-block" \*ngIf="formErrors.email">

      {{formErrors.email}}

    </span>

  </div>

</div>

<!-- Phone input element -->

<div class="form-group" [ngClass]="{'has-error': formErrors.phone}">

  <label class="col-sm-2 control-label" for="email">Phone</label>

  <div class="col-sm-8">

    <input id="phone" type="text" class="form-control"

            formControlName="phone" (blur)="logValidationErrors()">

    <span class="help-block" \*ngIf="formErrors.phone">

      {{formErrors.phone}}

    </span>

  </div>

</div>

**Component class code**

// Include phone property

formErrors = {

  'fullName': '',

  'email': '',

  'phone': '',

  'skillName': '',

  'experienceInYears': '',

  'proficiency': ''

};

// Include required error message for phone form control

validationMessages = {

  'fullName': {

    'required': 'Full Name is required.',

    'minlength': 'Full Name must be greater than 2 characters',

    'maxlength': 'Full Name must be less than 10 characters.',

  },

  'email': {

    'required': 'Email is required.',

    'emailDomain': 'Email domian should be prgaimtech.com'

  },

  'phone': {

    'required': 'Phone is required.'

  },

  'skillName': {

    'required': 'Skill Name is required.',

  },

  'experienceInYears': {

    'required': 'Experience is required.',

  },

  'proficiency': {

    'required': 'Proficiency is required.',

  },

};

ngOnInit() {

  // Include FormControls for contactPreference, email & phone

  // contactPreference has email as the default value

  this.employeeForm = this.fb.group({

    fullName: ['', [Validators.required, Validators.minLength(2), Validators.maxLength(10)]],

    contactPreference: ['email'],

    email: ['', Validators.required],

    phone: [''],

    skills: this.fb.group({

      skillName: ['', Validators.required],

      experienceInYears: ['', Validators.required],

      proficiency: ['', Validators.required]

    }),

  });

  this.employeeForm.valueChanges.subscribe((data) => {

    this.logValidationErrors(this.employeeForm);

  });

}

// If the Selected Radio Button value is "phone", then add the

// required validator function otherwise remove it

onContactPrefernceChange(selectedValue: string) {

  const phoneFormControl = this.employeeForm.get('phone');

  if (selectedValue === 'phone') {

    phoneFormControl.setValidators(Validators.required);

  } else {

    phoneFormControl.clearValidators();

  }

  phoneFormControl.updateValueAndValidity();

}

We can also achieve the same thing by subscribing to the **valueChanges**observable of **contactPreference**radio button in code, instead of binding to the **click event**in the HTML. The benefit of this approach is that, our code is easier to unit test.   
  
**Here are the steps**   
  
**Step 1 :** In the HTML remove click event binding from both the radio buttons (email and phone) 

<div class="form-group">

  <label class="col-md-2 control-label">Contact Preference</label>

  <div class="col-md-8">

    <label class="radio-inline">

      <input type="radio" value="email" formControlName="contactPreference">Email

    </label>

    <label class="radio-inline">

      <input type="radio" value="phone" formControlName="contactPreference">Phone

    </label>

  </div>

</div>

**Step 2 :**Subscribe to contactPreference form control **valueChanges observable**

this.employeeForm.get('contactPreference')

                 .valueChanges.subscribe((data: string) => {

  this.onContactPrefernceChange(data);

});

----------------------------------------------------------------Example 13-----------------------------------------------------------

### Angular reactive form custom validator

we will discuss, **creating and using a custom validator in an Angular reactive form.**  
  
  
Angular provides several built-in validator functions like required, pattern, minLength, maxLength, etc. Most of our application validation requirements can be met using one or more of the these built-in validator functions. However, sometimes we may need custom validation logic.   
  
  
**For example,**  
let's say we only want to allow an email address with **pragimtech.com**as the domain.    
  
Angular reactive form custom validator   
  
Any other email domain is invalid.    
  
   
  
We can achieve this very easily using a custom validator. Here are the steps.   
  
**Step 1 :** Create the custom validator function 

function emailDomain(control: AbstractControl): { [key: string]: any } | null {

  const email: string = control.value;

  const domain = email.substring(email.lastIndexOf('@') + 1);

  if (email === '' || domain.toLowerCase() === 'pragimtech.com') {

    return null;

  } else {

    return { 'emailDomain': true };

  }

}

Just like a builtin validator, a custom validator is also a function. If you take a look at the required built-in function, notice it takes AbstractControl as a parameter. Both FormControl and FormGroup inherits from AbstractControl class. Specifying AbstractControl as parameter type, allows us to pass either a FormControl or a FormGroup to validate. 

required(control: AbstractControl): ValidationErrors | null;

Notice the return type is either ValidationErrors object or null. The method returns null if the control passes validation otherwise ValidationErrors object. If you take a look at the definition of ValidationErrors type, it is an object with a key and a value. Key is a string and value can be anything. But we usually specify a value of true to indicate that there is a validation error.

{ [key: string]: any }

In the template, we use this same key to display the validation error message.   
  
**Step 2 :**Attach the custom validator function to the control that we want to validate

email: ['', [Validators.required, emailDomain]]

**Step 3 :** Display the validation error message  
  
If you want the validation error message and logic in the template, then check for emailDomin key on the errors collection of email form control 

<span \*ngIf="employeeForm.get('email').errors.emailDomain">

  Email domian should be prgaimtech.com

</span>

On the other hand, if you want the validation error message and logic in the component class, then include the validation message in validationMessages object as shown below. 

validationMessages = {

  'fullName': {...

  },

  'email': {

    'required': 'Email is required.',

    'emailDomain': 'Email domian should be pragimtech.com'

  },

  'phone': {...

  },

  'skillName': {...

  },

  'experienceInYears': {...

  },

  'proficiency': {...

  },

};

Here is the formErrors object which holds the messages to display. The template will bind to this object. 

formErrors = {

  'fullName': '',

  'email': '',

  'phone': '',

  'skillName': '',

  'experienceInYears': '',

  'proficiency': ''

};

This logValidationErrors() method checks if a control has failed validation. If it has, it populates the formErrors object, with the validation error message using the form control name as the key. 

logValidationErrors(group: FormGroup = this.employeeForm): void {

  Object.keys(group.controls).forEach((key: string) => {

    const abstractControl = group.get(key);

    if (abstractControl instanceof FormGroup) {

      this.logValidationErrors(abstractControl);

    } else {

      this.formErrors[key] = '';

      if (abstractControl && !abstractControl.valid

        && (abstractControl.touched || abstractControl.dirty)) {

        const messages = this.validationMessages[key];

        for (const errorKey in abstractControl.errors) {

          if (errorKey) {

            this.formErrors[key] += messages[errorKey] + ' ';

          }

        }

      }

    }

  });

}

In the template bind to the email property on the formErrors object 

<div class="form-group" [ngClass]="{'has-error': formErrors.email}">

  <label class="col-sm-2 control-label" for="email">Email</label>

  <div class="col-sm-8">

    <input id="email" type="text" class="form-control"

            formControlName="email" (blur)="logValidationErrors()">

    <span class="help-block" \*ngIf="formErrors.email">

      {{formErrors.email}}

    </span>

  </div>

</div>

----------------------------------------------------------------Example 14-----------------------------------------------------------

### Angular reactive form custom validator with parameter

we discussed creating a custom email domain validator. The following is that validator function. 

function emailDomain(control: AbstractControl): { [key: string]: any } | null {

  const email: string = control.value;

  const domain = email.substring(email.lastIndexOf('@') + 1);

  if (email === '' || domain.toLowerCase() === 'pragimtech.com') {

    return null;

  } else {

    return { 'emailDomain': true };

  }

}

Notice, the domain name 'pragimtech.com' is hard coded. So this custom validator, only works if you want to check if the domain is pragimtech.com. What if you want to check another domain like microsoft.com. We want to make this custom validator reusable with any domain name. We should be able to pass the domain name as a parameter to the emailDomain custom validator function.   
  
Notice in the example below, we are passing pragimtech.com as the domain name. If you want to check for a different domain, you simply pass that domain name as a parameter. 

email: ['', [emailDomain('pragimtech.com')]]

The following built-in validators have parameters. 

* min
* max
* minlength
* maxlength

Notice the min() built-in validator function. It takes in a number as a parameter and returns ValidatorFn. 

min(min: number): ValidatorFn;

**So, what is ValidatorFn?**  
**ValidatorFn**stands from **validator function**. So this min() function is taking in a number as a parameter and returns a validator function. If you understand the concept of closure in JavaScript, then this is very easy to understand. We discussed closures in detail in [Parts 27](https://www.youtube.com/watch?v=HjJQ-lvTgWg&list=PL6n9fhu94yhUA99nOsJkKXBqokT3MBK0b&t=0s&index=28) and [28](https://www.youtube.com/watch?v=w1s9PgtEoJs&list=PL6n9fhu94yhUA99nOsJkKXBqokT3MBK0b&t=0s&index=29) of [JavaScript tutorial](https://www.youtube.com/playlist?list=PL6n9fhu94yhUA99nOsJkKXBqokT3MBK0b).  
  
In simple terms, you can thinks of a closure as, a **function inside another function** i.e an inner function and an outer function. The inner function has access to the outer function’s variables and parameters in addition to it's own variables and parameters.  
  
Now that task at hand for us, is to convert our emailDomain() function to take in the domain name as a parameter and return a validator function. To be able to do this we are going to take the advantage of [closures in JavaScript](https://www.youtube.com/watch?v=HjJQ-lvTgWg&list=PL6n9fhu94yhUA99nOsJkKXBqokT3MBK0b&t=0s&index=28).  
  
ValidatorFn is an interface and the signature of the function it returns is as shown below. It takes the AbstractControl that we want to validate as an input parameter and returns null or ValidationErrors object. Null if the validation succeeds and a ValidationErrors object is the validation has failed. 

(c: AbstractControl): ValidationErrors | null;

**Custom Validator with parameter** 

function emailDomain(domainName: string) {

  return (control: AbstractControl): { [key: string]: any } | null => {

    const email: string = control.value;

    const domain = email.substring(email.lastIndexOf('@') + 1);

    if (email === '' || domain.toLowerCase() === domainName.toLowerCase()) {

      return null;

    } else {

      return { 'emailDomain': true };

    }

  };

}

**Code Explanation** 

* We have 2 functions here. An inner function and an outer function.
* The outer function has a name (emailDomain), but the inner function does not have a name. It is an anonymous function.
* The inner anonymous function has access to the outer function parameter domainName.
* You can have as many parameters as you want in the outer function, then inner function will have access to all of them in addition to it's own parameters.

**Passing the value for the custom validator parameter** 

email: ['', [Validators.required, emailDomain('dell.com')]]

Finally do not forget to updated the validation error message in the **validationMessages**structure 

validationMessages = {

  'email': {

    'required': 'Email is required.',

    'emailDomain': 'Email domian should be dell.com'

  },

  'proficiency': {

    'required': 'Proficiency is required.',

  },

};

----------------------------------------------------------------Example 15-----------------------------------------------------------

### Angular Reusable Custom Validator

will discuss **how to make a custom validator reusable in Angular**.   
  
  
The built-in validators in angular like the following, are reusable. This means we can use them with any form control on any angular form.  

* required
* min
* max
* minlength
* maxlength
* pattern

To be able to use one of the built-in angular validator, all we have to do is import the Validators class from '@angular/forms' package. 

import { Validators } from '@angular/forms';

Once the Validators class is imported, use the validator functions on the form control that you want to validate. 

fullName: ['', [Validators.required, Validators.minLength(2)]]

All the **buil-in validator** functions are marked as static functions in the Validators class. This allows us to use the validator functions, without the need to create an instance of the Validators class.   
  
Along the same lines let's make our emailDomain custom validator function reusable by including it as a static function in a separate class.   
  
We want to make this validator function available to all form controls on all forms. So, create a **shared**folder. In the shared folder, create a file with name **custom.validators.ts**and include the following code. 

import { AbstractControl } from '@angular/forms';

export class CustomValidators {

    static emailDomain(domainName: string) {

        return (control: AbstractControl): { [key: string]: any } | null => {

            const email: string = control.value;

            const domain = email.substring(email.lastIndexOf('@') + 1);

            if (email === '' || domain.toLowerCase() === domainName.toLowerCase()) {

                return null;

            } else {

                return { 'emailDomain': true };

            }

        };

    }

}

**Using the reusable Custom Validators :**First, Import the CustomValidators class. Just like how we import the built-in Validators class, import the CustomValidators class from custom.validators.ts file. 

import { CustomValidators } from '../shared/custom.validators';

Tie the validator function to a form control that you want to validate. 

email: ['', [CustomValidators.emailDomain('dell.com')]]

----------------------------------------------------------------Example 16-----------------------------------------------------------

----------------------------------------------------------------Example 17-----------------------------------------------------------

----------------------------------------------------------------Example 18-----------------------------------------------------------

----------------------------------------------------------------Example 19-----------------------------------------------------------

----------------------------------------------------------------Example 20-----------------------------------------------------------

----------------------------------------------------------------Example 21-----------------------------------------------------------

----------------------------------------------------------------Example 22-----------------------------------------------------------

----------------------------------------------------------------Example 23-----------------------------------------------------------

----------------------------------------------------------------Example 24-----------------------------------------------------------

----------------------------------------------------------------Example 25-----------------------------------------------------------

----------------------------------------------------------------Example 26-----------------------------------------------------------

----------------------------------------------------------------Example 27-----------------------------------------------------------

----------------------------------------------------------------Example 28-----------------------------------------------------------

----------------------------------------------------------------Example 29-----------------------------------------------------------

----------------------------------------------------------------Example 30-----------------------------------------------------------

----------------------------------------------------------------Example 31-----------------------------------------------------------

----------------------------------------------------------------Example 32-----------------------------------------------------------

----------------------------------------------------------------Example 33-----------------------------------------------------------

----------------------------------------------------------------Example 34-----------------------------------------------------------

----------------------------------------------------------------Example 35-----------------------------------------------------------

----------------------------------------------------------------Example 36-----------------------------------------------------------