# Angular 6

**Setting up Angular 6 Development environment**  
  
**Step 1 :** Install Node and NPM using the link below. <https://nodejs.org/en/download/>  
  
You need Node version 8.x or greater and NPM version 5.x or greater. To veriify the version of NODE and NPM you have, execute the following 2 commands using the command prompt window as an administrator.

node -v  
npm -v

If you already have NODE and NPM installed and if you want to upgrade to the latest version simply download and run the installer again, and it will automatically update NODE and NPM to the latest version.   
  
**Step 2 :**Install Angular CLI. It is a command line tool that help us create angular applications quickly and easily following angular's best practices and conventions. It's a great tool to increase productivity as an angular developer.   
  
If you do not have Angular CLI installed already, execute the following command and it will install the latest version.

npm install -g @angular/cli

If you already have Angular CLI installed, and you want to upgrade to latest version, then execute following command

npm install -g @angular/cli@latest

**Create a new Angular 6 project using the Angular CLI :**   
Run the command prompt window as an administrator. Execute the following command to create a new project.

ng new Angular6Project -d

**Command Explanation**

|  |  |
| --- | --- |
| ng | Is the Angular CLI |
| new | For generating a new project |
| Angular6Project | Is the name of the project |
| -d | This is dry-run flag. This flag instructs Angular CLI just to list the files it's going to create. It does not actually create them. This is a great option, because it lets us see what files will be generated without actually generating them. |

In our case, we do not want test files to be generated so let's also use --skip-tests option to skip generating test files. The following generates a new Angular 6 project and all the required files. Notice, we have not used -d option, so Angular CLI will create the Angular project and all the required files.

ng new Angular6Project --skip-tests

**Running the Angular 6 Project**   
  
**Step 1 :** In the command prompt window, use the following command to go to the project directory

cd Angular6Project

**Step 2 :** Execute the following command to run the Angular 6 project. This command builds, launches your default browser and serves the application using the default port number 4200 (http://localhost:4200/)

ng serve --open (short cut command : ng s -o)

At this point you will see the following on the browser. This message is coming from the root component

**AppComponent**.  
  
**Welcome to Angular6Project!**  
  
The project structure and the files in angular project have not changed much between Angular 5 and Angular 6. One change I can point out at this time is the Angular CLI configuration file. Prior to Angular 6, the Angular CLI configuration file is called ***angular-cli.json***. In Angular 6, it is renamed to just ***angular.json***. It's not a simple rename, the file format is also slightly different.

The ***src***folder contains the angular application components, templates, services, styles, images, and anything else the application needs. The files outside of this folder are meant to support building, testing, maintaining, documenting, and deploying the angular application.   
  
To confirm this Angular application is using Angular Version 6, open ***package.json***file and notice all the Angular packages version is 6.1.0.

Install bootstrap using the following NPM command  
**npm install bootstrap@3 jquery --save**  
  
This installs Bootstrap in the ***node\_modules***folder. If you do not find the bootstrap folder in node\_modules folder, restart visual studio code and you will find it. You can also execute the following command in the integrated terminal window and search for bootstrap folder.  
**dir node\_modules**   
  
In Angular CLI configuration file include the path to the Bootstrap CSS file in the styles array. Remember in Angular 6, the Angular CLI configuration file is renamed to ***angular.json***from ***angular-cli.json***.

"styles": [

  "src/styles.css",

  "node\_modules/bootstrap/dist/css/bootstrap.min.css"

]

Bootstrap relies on jQuery for some of it's features. If you plan on using those features, please include the path to jQuery and Bootstrap jQuery files in the scripts array in angular.json file.

"scripts": [

  "node\_modules/jquery/dist/jquery.min.js",

  "node\_modules/bootstrap/dist/js/bootstrap.min.js"

]

To verify that you have the correct paths specified, you can request the respective files by running the angular project and pointing your browser to http://localhost:4200/node\_modules/bootstrap/dist/css/bootstrap.min.css   
  
A quick additional test to verify, Bootstrap styles work as expected, include the following button in the root component (app.component.html) file.

<button class="btn btn-primary">

    Bootstrap Styled Button

</button>

# Angular 6 routing

Create following 2 components, so we can setup routing to navigate between these components.

CreateEmployeeComponent, ListEmployeesComponent

ng g c employee/create-employee --spec=false --flat=true  
ng g c employee/list-employees --spec=false --flat=true

**Step 1 : Set <base href="/"> in index.html :** When setting up routing in an angular application, the first step is to set the base path using the base href element. The base path tells the angular router, how to compose navigation URLs. When you create a new Angular 6 project, this is included automatically by the Angular CLI.   
  
**Step 2 : Create a separate routing module :** If routing is in it's own module, it is easier to find and change routing code if required. By convention, the routing module class name is called AppRoutingModule and the file is named app-routing.module.ts.

ng generate module app-routing --flat=true --module=app

ng g m app-routing --flat=true -m=app

**Step 3 : 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.

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';

// last route path is empty. This specifies route to redirect to if client side path is empty.

const appRoutes: Routes = [

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

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

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

];

// Pass configured routes to forRoot() method to let the angular router know about our routes

// Export imported RouterModule so router directives are available to module that imports this AppRoutingModule

@NgModule({

  imports: [ RouterModule.forRoot(appRoutes) ],

  exports: [ RouterModule ]

})

export class AppRoutingModule { }

**Step 4 :** 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>

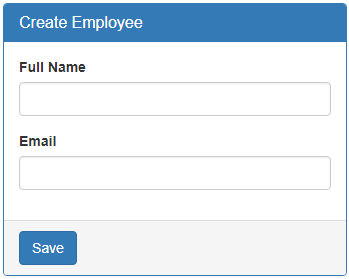
    <router-outlet></router-outlet>

</div>

# Angular Reactive Forms

There are 2 ways to create forms in Angular

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

**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.   
  
**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.   
  
With a reactive form, we create the entire form control tree in the component class code.   
   
**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. create-employee.component.ts

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 FormGroup and it's form controls are created when 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.

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;}

**How are we able to pass a FormControl instance when it is expecting AbstractControl instance.**   
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.  
  
**Binding the FormGroup model and the view :**  
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

<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>

**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.

**Notes:**

When working with reactive forms we create instances of **FormControl**and **FormGroup**classes to create 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**

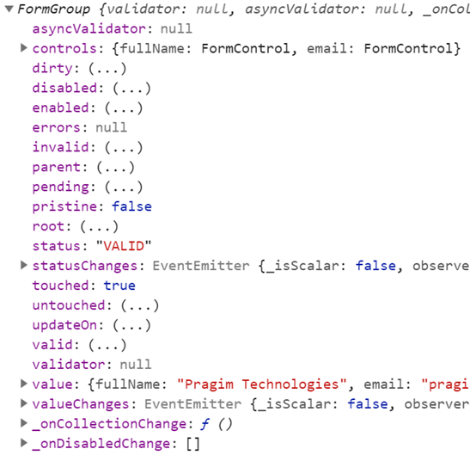
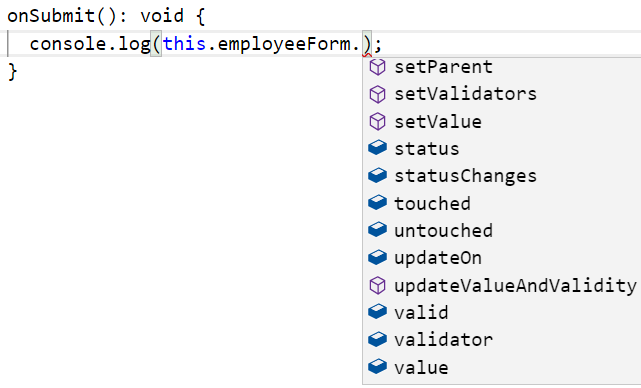
Both **FormControl**and **FormGroup**classes inherit from **AbstractControl**base class

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 form model we created using **FormGroup**and **FormControl**classes, log the employeeForm to the console.   
onSubmit(): void {  
  console.log(this.employeeForm);  
}

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}} <br/>

    </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}}<br/>

    </td>

  </tr>

</table>

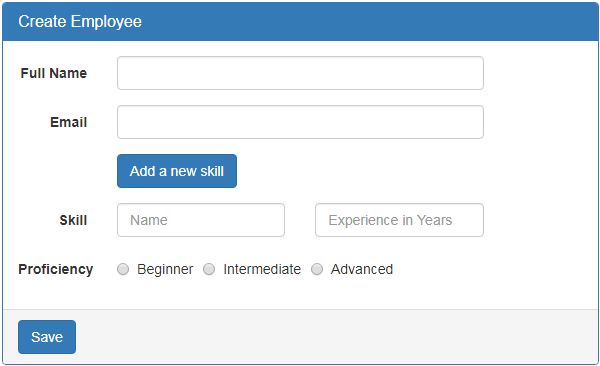
In addition to these properties, **AbstractControl**also provides the following methods. In our upcoming videos we will use these properties and methods for form validation and working with data.

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

## Nested Form Groups

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

* Skill Name
* Experience in Years
* Proficiency

We want to be able to add multiple skills dynamically at run time, by clicking **"Add a new skill"** button and  
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 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. Notice we have created a nested form group with key - **skills**.

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()

    })

  });

}

**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"

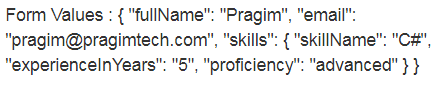
               formControlName="proficiency">Advanced

      </label>

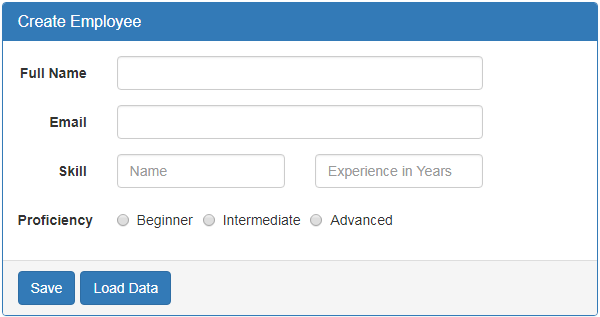
    </div>

  </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.

## Setvalue and Patchvalue Methods

**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**

## Angular formbuilder

**In Angular, there are 2 ways to create reactive forms**

1. Explicitly creating instances of FormGroup and FormControl classes using the **new**keyword.
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

**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 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.

### Reactive Forms Validation

**Step 1 - Import Angular Validators class :** This class has the following validator functions

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

|  |  |
| --- | --- |
| **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. |
| max | Validate that the field value is less than or equal to the provided number. |
| minLength | The number of characters in the field must be greater than or equal to the provided minimum length. |
| 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.

We can also write **custom validator**, if requirements are not met using one of the above built-in validator functions.

**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>

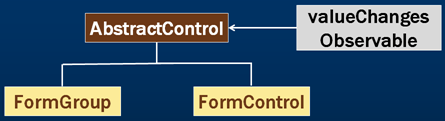
</div>

At moment, the validation messages are still in the template HTML. We can also move them into the component class.

### Form Control Valuechanges

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:**

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

### Loop through all form controls in formgroup

loop through all form controls in a formgroup including nested form groups in a reactive form.   
  
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']

  }),

});

**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

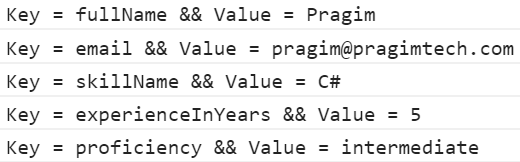
    } else {

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

    }

  });

}

We want every form control key and value to be logged to the console as shown below.   


### Validation Messages in Component Class

There are several benefits to **move validation messages to the component class.**

* Easily unit test validation logic
* Instead of hard-coding validation messages , 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

**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': ''

};

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);

}

**Validation Logic in Component Class**

**create-employee.component.html :**

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

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>

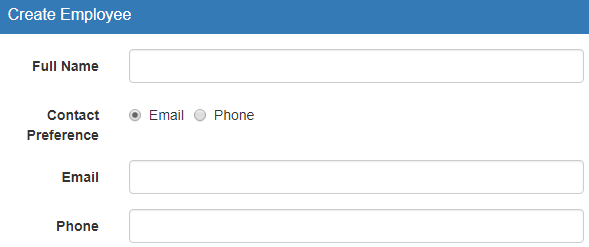
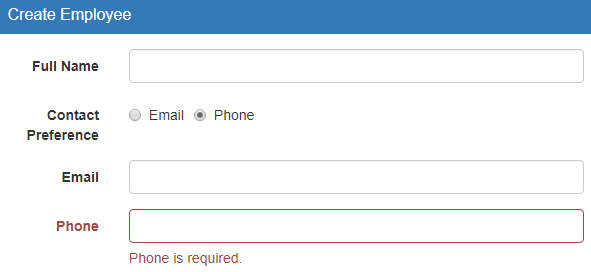
      </div>

    </div>

  </div>

</div>

### Dynamically form control validators

Phone filed is optional.But 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);

});

### custom validator

We want to allow an email address with **pragimtech.com**as the domain.  Any other email domain is invalid.    
Angular reactive form custom validator   
  
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'

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>

### Custom Validator With Parameter

We want to make 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.

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;

**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 inner function doesn’t 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('pragimtech.com')]]

### Reusable Custom Validator

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 };

            }

        };

    }

}

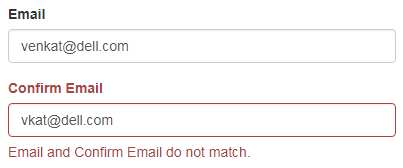
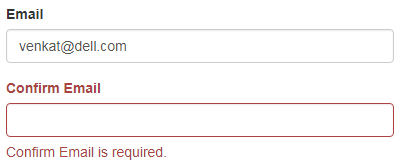
**Using the reusable Custom Validators :**First, Import the CustomValidators class. Just like how we import the built-in Validators class, import the CustomValidators class from custom.validators.ts file.

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

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

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

### Reactive forms cross field validation

Let us understand this with an example. We want to ensure **Email**and **Confirm Email**fields have the same value. If they do not match, we want to display a validation message. **ConfirmEmail**field is required and if no value is present it should display the required validation error.   
  
   
  
**So in short, here is the requirement**

* Confirm Email field is required and if a value is present it should match with the Email field value.
* If no value is entered, it should display required error
* If a value is present and does not match with Email field, it should display do not match validation error

To validate if Email and Confirm Email fields have same value, we need to compare 2 Form Controls. If you look at a Validator function in Angular, it only accepts either a FormGroup or a FormControl as a parameter. We cannot pass 2 form controls to the validator function, but what we can do is group them using a nested formgroup and then pass that nested formgroup as a parameter to the Validator function.  
  
**Changes in the Template (create-employee.component.html)**

<div formGroupName="emailGroup">

  <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="form-group" [ngClass]="{'has-error': formErrors.confirmEmail

                                                || formErrors.emailGroup}">

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

      Confirm Email

    </label>

    <div class="col-sm-8">

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

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

      <span class="help-block"

            \*ngIf="formErrors.confirmEmail || formErrors.emailGroup">

        {{formErrors.confirmEmail ? formErrors.confirmEmail

          : formErrors.emailGroup}}

      </span>

    </div>

  </div>

</div>

Notice **email**and **confirmEmail**form controls are nested in a **formgroup**with name **emailGroup**

* Bootstrap has-error class is conditionally added if either **confirmEmail**or **emailGroup**properties of the **formErrors**object are truthy
* confirmEmail property stores required error - Confirm Email is required.
* emailGroup property stores do not match error - Email and Confirm Email do not match
* We do not have these 2 properties on the **formErrors**object yet. We will add them in the component class in just a bit.
* Similarly, the span element that displays the validation error is bound to **confirmEmail**or **emailGroup**properties of the **formErrors**object. So the span element is displayed only if either of the properties are truthy.

If the **Email**form control has a value and if nothing is filled in the **confirmEmail**form control, we do not want both the required error and do not match error to be displayed. The following interpolation expression, ensures to display the right validation message.

{{formErrors.confirmEmail ? formErrors.confirmEmail : formErrors.emailGroup}}

**Changes in the Component Clas (create-employee.component.ts) :** The changes are commented and self-explanatory.

// Group properties on the formErrors object. The UI will bind to these properties

// to display the respective validation messages

formErrors = {

  'fullName': '',

  'email': '',

  'confirmEmail': '',

  'emailGroup': '',

  'phone': '',

  'skillName': '',

  'experienceInYears': '',

  'proficiency': ''

};

// This structure stoes all the validation messages for the form Include validation

// messages for confirmEmail and emailGroup properties. Notice to store the

// validation message for the emailGroup we are using emailGroup key. This is the

// same key that the matchEmails() validation function below returns, if the email

// and confirm email do not match.

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 dell.com'

  },

  'confirmEmail': {

    'required': 'Confirm Email is required.'

  },

  'emailGroup': {

    'emailMismatch': 'Email and Confirm Email do not match.'

  },

  'phone': {

    'required': 'Phone is required.'

  },

  'skillName': {

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

  },

  'experienceInYears': {

    'required': 'Experience is required.',

  },

  'proficiency': {

    'required': 'Proficiency is required.',

  },

};

// email and confirmEmail form controls are grouped using a nested form group

// Notice, the validator is attached to the nested emailGroup using an object

// with key validator. The value is our validator function matchEmails() which

// is defined below. The important point to keep in mind is when the validation

// fails, the validation key is attached the errors collection of the emailGroup

// This is the reason we added emailGroup key both to formErrors object and

// validationMessages object.

ngOnInit() {

  this.employeeForm = this.fb.group({

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

    contactPreference: ['email'],

    emailGroup: this.fb.group({

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

      confirmEmail: ['', [Validators.required]],

    }, { validator: matchEmails }),

    phone: [''],

    skills: this.fb.group({

      skillName: ['', Validators.required],

      experienceInYears: ['', Validators.required],

      proficiency: ['', Validators.required]

    }),

  });

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

    this.logValidationErrors(this.employeeForm);

  });

  this.employeeForm.get('contactPreference').valueChanges.subscribe((data: string) => {

    this.onContactPrefernceChange(data);

  });

}

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

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

    const abstractControl = group.get(key);

    this.formErrors[key] = '';

    // Loop through nested form groups and form controls to check

    // for validation errors. For the form groups and form controls

    // that have failed validation, retrieve the corresponding

    // validation message from validationMessages object and store

    // it in the formErrors object. The UI binds to the formErrors

    // object properties to display the validation errors.

    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] + ' ';

        }

      }

    }

    if (abstractControl instanceof FormGroup) {

      this.logValidationErrors(abstractControl);

    }

  });

}

Finally, include the following validator function in **create-employee.component.ts** file, after the closing curly brace (}) of the CreateEmployeeComponent class.

// Nested form group (emailGroup) is passed as a parameter. Retrieve email and

// confirmEmail form controls. If the values are equal return null to indicate

// validation passed otherwise an object with emailMismatch key. Please note we

// used this same key in the validationMessages object against emailGroup

// property to store the corresponding validation error message

function matchEmails(group: AbstractControl): { [key: string]: any } | null {

  const emailControl = group.get('email');

  const confirmEmailControl = group.get('confirmEmail');

  if (emailControl.value === confirmEmailControl.value || confirmEmailControl.pristine) {

    return null;

  } else {

    return { 'emailMismatch': true };

  }

}

## **Angular Formarray**

To build an Angular Reactive Form we use three fundamental building blocks **FormControl |** **FormGroup |** **FormArray**

A FormArray is an array of  FormControls | FormGroups | Nested FormArrays.

We usually use an array to hold like items, but a FormArray can contain unlike items as well, i.e a few elements in a given array can be FormControls, a few of them in same array can be FormGroups and rest of them can be FormArrays.  
  
In the example below, we have a FormArray with 1 FormControl | 1 FormGroup | 1 FormArray

const formArray = new FormArray([

  new FormControl('John', Validators.required),

  new FormGroup({

    country: new FormControl('', Validators.required)

  }),

  new FormArray([])

]);

To programmatically find the number of elements in a FormArray use the length property

formArray.length

To iterate over a FormArray you can use a for loop. Use instanceof operator to determine if the control that you are currently dealing with is a FormControl, FormGroup or FormArray.

for (const control of formArray.controls) {

  if (control instanceof FormControl) {

    console.log('control is FormControl');

  }

  if (control instanceof FormGroup) {

    console.log('control is FormGroup');

  }

  if (control instanceof FormArray) {

    console.log('control is FormArray');

  }

}

There are 2 ways to create a FormArray in Angular. Using the new keyword or FormBuilder class.   
  
**Create a FormArray, using the new keyword**

const formArray = new FormArray([

  new FormControl('John', Validators.required),

  new FormControl('IT', Validators.required),

]);

**Create a FormArray, using the array() method of the FormBuilder class**

const formArray = this.fb.array([

  new FormControl('John', Validators.required),

  new FormControl('IT', Validators.required),

]);

Although, we can use FormArray to store unlike items, we generally use it to store like items.

For example, an array of  FormControls | FormGroups | Nested FormArrays.

const formArray = this.fb.array([

  new FormControl('John', Validators.required),

  new FormControl('IT', Validators.required),

  new FormControl('', Validators.required),

]);

The value property of FormArray, returns an array containing values of each child FormControl.

formArray.value returns: ["John", "IT", ""]

We usually use the following properties to determine the state of FormControl or FormGroup or FormArray. For example, if one of the controls in a FormArray is touched, the entire array becomes touched. Similarly, if one of the controls is invalid, the entire array becomes invalid.

* touched
* untouched
* dirty
* pristine
* valid
* invalid

**Useful FormArray methods**

|  |  |
| --- | --- |
| **Method** | **Purpose** |
| push | Inserts the control at the end of the array |
| insert | Inserts the control at the specified index in the array |
| removeAt | Removes the control at the specified index in the array |
| setControl | Replace an existing control at the specified index in the array |
| at | Return the control at the specified index in the array |

We can also use a FormGroup to create a group of FormControls. Notice, in the example below, we are using the group() method of the FormBuilder class to create a FormGroup.

const formGroup = this.fb.group([

  new FormControl('John', Validators.required),

  new FormControl('IT', Validators.required),

  new FormControl('', Validators.required),

]);

**What is the difference between a FormGroup and a FormArray?**  
Well, in many aspects they are similar. However, one major difference is that a FormArray data is serialized as an array where as a FormGroup is serialized as an object.    
  
To see this, log the FormGroup and FormArray instance to the browser console.

In the case of FormArray, controls property contains an array of form controls.

In the case of FormGroup, controls property contains an object with key/value pairs, where key is the name of the form control and value is an instance of FormControl.

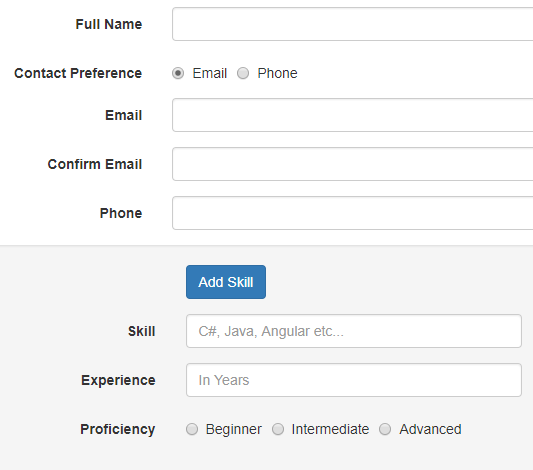
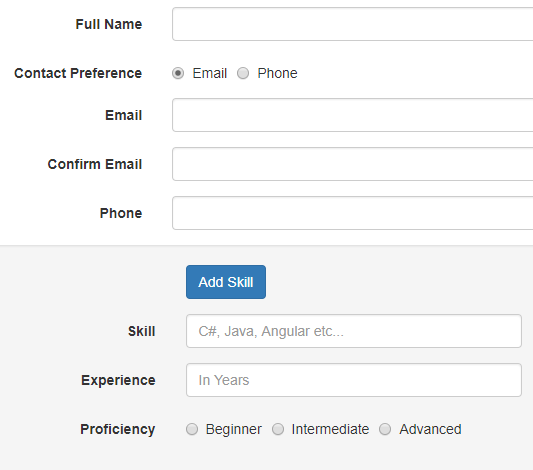
console.log(formArray.value);   
**Output:** [FormControl, FormControl, FormControl]

console.log(formGroup.value);  
**Output:** {0: FormControl, 1: FormControl, 2: FormControl}

The fact that FormArray tracks FormControls as part of an array is very useful, when we want to generate FormControls and FormGroups dynamically. For example, let's say you are filling an employment form, and you want to add multiple skills. We cannot have a fixed number of skill related fields on the form as they are dependant on the employee experience. This is one example where we need to generate form controls dynamically, and a FormArray is a perfect choice for implementing this. 

### Creating formarray of formgroup objects

**If you are wondering, why are we doing this?**  
Well, this is preparation for dynamically creating **FormGroups**at runtime. Every time we click, "Add Skill" button on the "Employee Form" below, we want to dynamically generate a new set of skill related form fields.

   
**Component Class Code :**

constructor(private fb: FormBuilder) { }

ngOnInit() {

  this.employeeForm = this.fb.group({

    fullName: ['', [Validators.required]],

    contactPreference: ['email'],

    // Other Form Controls..

    // Create skills FormArray using injected FormBuilder class array() method.

    // At the moment, in the created FormArray we only have one FormGroup instance that is

    // returned by addSkillFormGroup() method

    skills: this.fb.array([

      this.addSkillFormGroup()

    ])

  });

  // Rest of the code

}

addSkillFormGroup(): FormGroup {

  return this.fb.group({

    skillName: ['', Validators.required],

    experienceInYears: ['', Validators.required],

    proficiency: ['', Validators.required]

  });

}

In the template, use the formArrayName directive to bind to the skills FormArray. In component class, we only have one FormGroup instance in the skills FormArray. That one FormGroup instance is present at index position ZERO in the FormArray. This is the reason we have set formGroupName="0".   
  
**HTML in the view template**

<div class="well">

  <div formArrayName="skills">

    <div formGroupName="0">

      <!-- Skill Name Label & Form Control HTML

        Experience Label & Form Control HTML

        Proficiency Label & Form Control HTML -->

    </div>

  </div>

</div>

With the above 2 changes, the validation is broken. To fix it, modify the code in logValidationErrors() method

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

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

    const abstractControl = group.get(key);

    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] + ' ';

        }

      }

    }

    if (abstractControl instanceof FormGroup) {

      this.logValidationErrors(abstractControl);

    }

    // We need this additional check to get to the FormGroup

    // in the FormArray and then recursively call this

    // logValidationErrors() method to fix the broken validation

    if (abstractControl instanceof FormArray) {

      for (const control of abstractControl.controls) {

        if (control instanceof FormGroup) {

          this.logValidationErrors(control);

        }

      }

    }

  });

}

**Next video :** We will discuss generating skill realted FormGroups and FormControls dynamically at runtime.

### Angular dynamic forms

**generating FormGroups and FormControls dynamically at runtime**.    
  
**Step 1 :**Include Add Skill button  
  
Place the following HTML inside the <div> element with class well. Notice the clickevent is bound to addSkillButtonClick() method. We will create this method in the component class next.

<div class="form-group">

  <div class="col-md-offset-2 col-md-4">

    <button type="button" class="btn btn-primary" (click)="addSkillButtonClick()">

      Add Skill

    </button>

  </div>

</div>

**Step 2 :**Include addSkillButtonClick() method in the component class

* From the root FormGroup "employeeForm" get a reference to the skillsFormArray. Notice we have passed the name of the FormArray (skills) as a parameter to the get() method.
* The get() method returns the FormArray as an AbstractControl. We know it's a FormArray so we are type casting it to FormArray.
* We are then calling the push() method of the FormArray to push a new FormGroup into the FormArray
* The push() method calls addSkillFormGroup() method which returns an instance of the FormGroup with the 3 skill related form controls (skill, experience & proficiency)

addSkillButtonClick(): void {

  (<FormArray>this.employeeForm.get('skills')).push(this.addSkillFormGroup());

}

**Step 3 :** Loop over "skills" FormArray to dynamically generate the HTML input elements.

* Notice we are using \*ngFor structural directive to loop over the "skills"FormArray
* For each FormGroup in FormArray, HTML input elements (skill, experience & proficiency) will be generated
* We are also binding the formGroupName directive to the loop variable i
* Since we are binding to a variable do not forget to use square brackets around [formGroupName] directive, otherwise you will get a run-time error - Cannot find control with path: 'skills -> i'

<div formArrayName="skills"

     \*ngFor="let skill of employeeForm.get('skills').controls; let i = index">

  <div [formGroupName]="i">

      <!-- Skill Name Label & Form Control HTML

      Experience Label & Form Control HTML

      Proficiency Label & Form Control HTML -->

  </div>

</div>

At the moment there are several problems with the dynamically generated HTML input elements

* The dynamically generated HTML input elements have the same id. For example, all the "skillName" textboxes have the same id "skillName". Not only the id, even the for attribute value of the labels is also the same.
* As a result when we click on a label of an input element, the wrong textbox receives focus.
* Also, the validation is broken.
* We will discuss fixing these issues in our next upcoming videos.

### unique id for dynamically created form controls

IDs on an HTML page are meant to be unique. Having duplication IDs is a terrible idea. Although it may work in some cases, it is semantically incorrect to do so. In our case the application behaves erratically, because of **duplicate element IDs**. When we click on a label, we expect it's corresponding input element to receive focus, but, in our case, a different element receives focus. To fix this we have to **generate unique ID values for the input elements**.

<div formArrayName="skills"

      \*ngFor="let skill of employeeForm.get('skills').controls; let i = index">

  <div [formGroupName]="i">

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

      <label class="col-sm-2 control-label" attr.for="{{'skillName'+i}}">Skill</label>

      <div class="col-sm-4">

        <input type="text" class="form-control" id="{{'skillName'+i}}"

                formControlName="skillName" (blur)="logValidationErrors()"

                placeholder="C#, Java, Angular etc...">

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

      </div>

    </div>

    <!-- Experience Label & Form Control HTML

         Proficiency Label & Form Control HTML -->

  </div>

</div>

* \*ngFor directive on the formArrayName <div> element loops through all the dynamically generated form groups.
* The variable **i** value will be ZERO when we are looping through the first FormGroup in the skills form array.
* The value of **i** will be ONE, for the second FormGroup, TWO for the third FormGroup so on and so forth.

Notice the id attribute of the skillName textbox. We are dynamically computing it's ID, by appening **i** variable value to the string skillName. id="{{'skillName'+i}}"

This will generate an ID of skillName0 for the first skillName input element, ID of skillName1 for the second skillName input element, so on so forth, to ensure unique ID values are assigned to all the dynamically generated skillName input elements.

In the above expression we are using interpolation. We could also achieve the same using property binding syntax instead of interpolation. [id]="'skillName'+i"

Also notice, we are dynamically setting the value of the for attribute of the Skill label. attr.for="{{'skillName'+i}}"

Since the for attribute does not have a corresponding DOM property, we are using Angular's attribute binding. With the attribute binding, we are using interpolation. We could also achieve the same using property binding syntax.

[attr.for]="'skillName'+i"

Similarly set a unique value for the **experienceInYears**input element ID and it's associated label for attribute.

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

  <label class="col-sm-2 control-label" attr.for="{{'experienceInYears'+i}}">Experience

  </label>

  <div class="col-sm-4">

    <input type="text" class="form-control" id="{{'experienceInYears'+i}}"

            formControlName="experienceInYears"

            placeholder="In Years" (blur)="logValidationErrors()">

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

      {{formErrors.experienceInYears}}

    </span>

  </div>

</div>

### Angular dynamic forms validation

Modify the skillName form control HTML as shown below.  

<div formArrayName="skills"

    \*ngFor="let skill of employeeForm.get('skills').controls; let i = index">

  <div [formGroupName]="i">

    <div class="form-group" [ngClass]="{'has-error':

          skill.get('skillName').invalid && skill.get('skillName').touched}">

      <label class="col-sm-2 control-label" [attr.for]="'skillName'+i">

        Skill

      </label>

      <div class="col-sm-4">

        <input type="text" class="form-control" [id]="'skillName'+i"

                formControlName="skillName" placeholder="C#, Java, Angular etc...">

        <span class="help-block" \*ngIf="skill.get('skillName').errors?.required &&

                                              skill.get('skillName').touched">

          Skill Name is required

        </span>

      </div>

    </div>

  </div>

</div>

Notice, we are using the loop variable **skill**, to check if the dynamically generated **skillName**form control is invalid and touched. If so, the has-error bootstrap style class is applied. To get to the **skillName**form control in the skill FormGroup, we are using the get() method on the FormGroup and passing it the form control name.

[ngClass]="{'has-error': skill.get('skillName').invalid &&

                         skill.get('skillName').touched}"

To get to the **skillName**form control in the **skill**FormGroup, we can also use controls property on the FormGroup and then the **skillName**form control.

[ngClass]="{'has-error': skill.controls.skillName.invalid &&

                          skill.controls.skillName.touched}"

Even here, we are using the loop variable **skill**, to check if the dynamically generated **skillName**form control has failed required validation and touched. If so, the <span>element displays the validation error, otherwise hides it.

<span class="help-block" \*ngIf="skill.get('skillName').errors?.required &&

                                      skill.get('skillName').touched">

  Skill Name is required

</span>

Make sure to use the safe navigation operator between errors and required properties. This is because, when the **skillName**form control does not have any validation errors, the errors property will be null and trying to check for required key on a null object will result in **Cannot read property 'required' of null error.**

skill.get('skillName').errors?.required

Make similar changes on **experienceInYears**and **proficiency**form controls.

<div class="form-group" [ngClass]="{'has-error':

skill.get('experienceInYears').invalid && skill.get('experienceInYears').touched}">

  <label class="col-sm-2 control-label" [attr.for]="'experienceInYears'+i">

    Experience

  </label>

  <div class="col-sm-4">

    <input type="text" class="form-control" [id]="'experienceInYears'+i"

            formControlName="experienceInYears" placeholder="In Years">

    <span class="help-block" \*ngIf="skill.get('experienceInYears').errors?.required &&

                                    skill.get('experienceInYears').touched">

      Experience is required

    </span>

  </div>

</div>

<div class="form-group" [ngClass]="{'has-error':

skill.get('proficiency').invalid && skill.get('proficiency').touched}">

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

  <div class="col-sm-8">

    <label class="radio-inline">

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

    </label>

    <label class="radio-inline">

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

    </label>

    <label class="radio-inline">

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

    </label>

    <span class="help-block" \*ngIf="skill.get('proficiency').errors?.required &&

                                    skill.get('proficiency').touched">

      Proficiency is required

    </span>

  </div>

</div>

All the **skill**form controls validation messages are in the template. So delete the validation messages from the **validationMessages**object in the component class.

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 dell.com'

  },

  'confirmEmail': {

    'required': 'Confirm Email is required.',

  },

  'emailGroup': {

    'emailMismatch': 'Email and Confirm Email do not match',

  },

  'phone': {

    'required': 'Phone is required.'

  },

  // 'skillName': {

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

  // },

  // 'experienceInYears': {

  //   'required': 'Experience is required.',

  // },

  // 'proficiency': {

  //   'required': 'Proficiency is required.',

  // },

};

On the **formErrors**object also, delete the **skill**related properties. In fact, we do not need any of the properties on the **formErrors**object, as they will be dynamically added when the corresponding form control fails validation. Notice, I have commented all the properties on the **formErrors**object.

formErrors = {

  // 'fullName': '',

  // 'email': '',

  // 'confirmEmail': '',

  // 'emailGroup': '',

  // 'phone': '',

  // 'skillName': '',

  // 'experienceInYears': '',

  // 'proficiency': ''

};

Since the validation messages for the **skill**form controls are in the template, we do not have to loop through the **skill form groups**in the form array. So I have commented the block of code that loops through the FormArray.

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

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

    const abstractControl = group.get(key);

    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] + ' ';

        }

      }

    }

    if (abstractControl instanceof FormGroup) {

      this.logValidationErrors(abstractControl);

    }

    // if (abstractControl instanceof FormArray) {

    //   for (const control of abstractControl.controls) {

    //     if (control instanceof FormGroup) {

    //       this.logValidationErrors(control);

    //     }

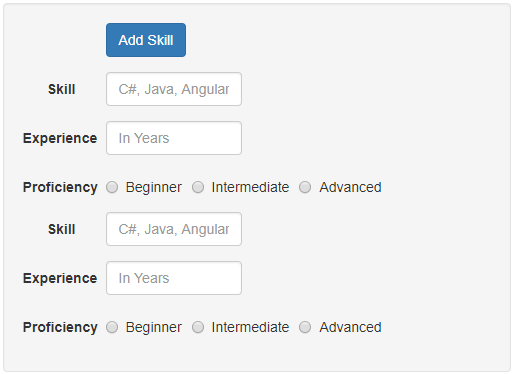
    //   }

    // }

  });

}

### Angular formarray validation

* A FormArray can contain form controls, form groups or nested form arrays.
* In our case, the "skills" form array, contains form groups. Each form group contains, 3 form controls.
* If all the 3 form controls are valid, then the form group is valid.
* If all the form groups are valid, then the form array is valid.
* Even if a single form control in a form group is invalid, then the form array is invalid.

Here is what we want to do. Until, all the skill related form controls are properly filled and valid, we want to **keep "Add Skill" button disabled**.    
   
This is very easy to achieve. We already know, even if a single form control is invalid, the entire form array is invalid. So to keep the "Add Skill" button disabled, bind the button disabled property to "skills" form array invalid property as shown below.

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

        (click)="addSkillButtonClick()"

        [disabled]="employeeForm.get('skills').invalid">

  Add Skill

</button>

While we are here, let's also include an <hr> element to separate each skills form group.   
   
We want the <hr> element to be generated, only if we have more than 1 skills form group, hence we have bound \*ngIf directive on the <hr> element to i>0

<div formArrayName="skills"

     \*ngFor="let skill of employeeForm.get('skills').controls; let i = index">

  <hr \*ngIf="i>0">

<div [formGroupName]="i">

In the CSS file, include the following style for the <hr> element

hr {

    border: 1px solid silver;

}

### Remove dynamically created form controls angular remove dynamic formcontrol from formgroup

<div class="col-sm-6" \*ngIf="employeeForm.get('skills').length>1">

  <button type="button" class="btn btn-danger btn-sm pull-right"

          title="Delete Skill" (click)="removeSkillButtonClick(i)">

    <span class="glyphicon glyphicon-remove"></span>

  </button>

</div>

**Code Explanation :**   
The following \*ngIf expression ensures, the DELETE SKILL button is only displayed if we have more than one skill FormGroup in the skills FormArray

\*ngIf="employeeForm.get('skills').length>1"

When the button is clicked, removeSkillButtonClick(i) method is called. Notice we are passing the loop variable **i** to the method. This is the index of the FormGroup that we want to remove from the skills FormArray.

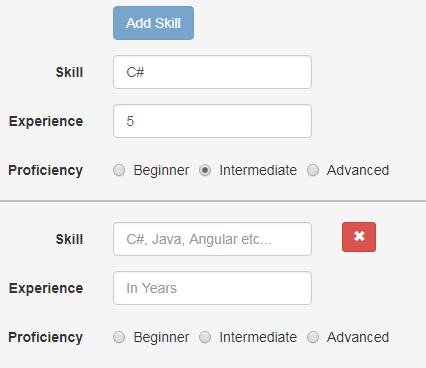
(click)="removeSkillButtonClick(i)"

The title attribute on the button, displays "Delete Skill" tooltip when you hover the mouse pointer over the button

title="Delete Skill"

To display the red cross on the button, we are using Bootstrap glyphicon and glyphicon-remove classes.

<span class="glyphicon glyphicon-remove"></span>

If you do not want to display the DELETE button for the first "skills" form group   
   
  
Then use the following ngIf expression on the <div> element that contains the "DELETE" button. This will ensure that the DELETE button will only be displayed for all the dynamically generated SKILL form groups except the first one.

<div class="col-sm-6" \*ngIf="i>0">

Include the following method in the component class.

removeSkillButtonClick(skillGroupIndex: number): void {

  (<FormArray>this.employeeForm.get('skills')).removeAt(skillGroupIndex);

}

**Code Explanation**

* Notice we are using get() on the root form group (employeeForm) and passing it the name of our skills FormArray.
* The get() method returns the FormArray as an abstract control
* To be able to use removeAt() method of the FormArray class, we are type casting the AbstractControl type to FormArray using the type caste operator <FormArray>
* To the removeAt() method we pass the index of the skill FormGroup we want to remove from the skills FormArray

## **rxjs operators in angular services**

First let's create a fake online REST API. Execute the following NPM command to install JSON server

npm install -g json-server

Create **db.json** file in the root project folder. Copy and paste the following JSON data in the file.

{

    "employees": [

        {

            "id": 1,

            "fullName": "Mark",

            "contactPreference": "email",

            "email": "mark@email.com",

            "phone": "5641238971",

            "skills": [

                {

                    "skillName": "C#",

                    "experienceInYears": 1,

                    "proficiency": "beginner"

                },

                {

                    "skillName": "Java",

                    "experienceInYears": 2,

                    "proficiency": "intermediate"

                }

            ]

        },

        {

            "id": 2,

            "fullName": "John",

            "contactPreference": "phone",

            "email": "john@email.com",

            "phone": "3242138971",

            "skills": [

                {

                    "skillName": "Angular",

                    "experienceInYears": 2,

                    "proficiency": "beginner"

                },

                {

                    "skillName": "HTML",

                    "experienceInYears": 2,

                    "proficiency": "intermediate"

                },

                {

                    "skillName": "LINQ",

                    "experienceInYears": 3,

                    "proficiency": "advanced"

                }

            ]

        }

    ]

}

Execute the following command to start the server

json-server --watch db.json

At this point, fire up the browser and navigate to http://localhost:3000/employees/ to see the list of all employees along with their skills. You can test this REST API using a tool like fiddler.   
  
**Creating the required interfaces to represent Employee and Skill types**

Add a file in the employee folder with name **ISkill.ts**. Copy and paste the following code.

export interface ISkill {  
    skillName: string;  
    experienceInYears: number;  
    proficiency: string;  
}

Add a file in the employee folder with name **IEmployee.ts**. Copy and paste the following code.

import { ISkill } from './ISkill';

export interface IEmployee {

    id: number;

    fullName: string;

    email: string;

    phone?: number;

    contactPreference: string;

    skills: ISkill[];

}

**Creating Angular Service**  
  
Add a file in the employee folder with name **employee.service.ts**. Copy and paste the following code.

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

import { IEmployee } from './IEmployee';

import { HttpClient, HttpErrorResponse, HttpHeaders } from '@angular/common/http';

import { Observable, throwError } from 'rxjs';

import { catchError } from 'rxjs/operators';

@Injectable()

export class EmployeeService {

    baseUrl = 'http://localhost:3000/employees';

    constructor(private httpClient: HttpClient) {

    }

    getEmployees(): Observable<IEmployee[]> {

        return this.httpClient.get<IEmployee[]>(this.baseUrl)

            .pipe(catchError(this.handleError));

    }

    private handleError(errorResponse: HttpErrorResponse) {

        if (errorResponse.error instanceof ErrorEvent) {

            console.error('Client Side Error :', errorResponse.error.message);

        } else {

            console.error('Server Side Error :', errorResponse);

        }

        return throwError('There is a problem with the service. We are notified & working on it. Please try again later.');

    }

    getEmployee(id: number): Observable<IEmployee> {

        return this.httpClient.get<IEmployee>(`${this.baseUrl}/${id}`)

            .pipe(catchError(this.handleError));

    }

    addEmployee(employee: IEmployee): Observable<IEmployee> {

        return this.httpClient.post<IEmployee>(this.baseUrl, employee, {

            headers: new HttpHeaders({

                'Content-Type': 'application/json'

            })

        })

        .pipe(catchError(this.handleError));

    }

    updateEmployee(employee: IEmployee): Observable<void> {

        return this.httpClient.put<void>(`${this.baseUrl}/${employee.id}`, employee, {

            headers: new HttpHeaders({

                'Content-Type': 'application/json'

            })

        })

            .pipe(catchError(this.handleError));

    }

    deleteEmployee(id: number): Observable<void> {

        return this.httpClient.delete<void>(`${this.baseUrl}/${id}`)

            .pipe(catchError(this.handleError));

    }

}

**RxJS 5 vs 6**  
An Angular 6 project, by default uses RxJS version 6. RxJS 6 has some breaking changes compared to RxJS 5.5 and older versions.   
  
The way we import some of the classes like Observable and Subject has changed in RxJS 6.

In RxJS 5, we import Observable and Subject classes as shown below.

import { Observable } from 'rxjs/Observable';

import { Subject } from 'rxjs/Subject';

In **RxJS 6**, this has changed to

import { Observable, Subject } from 'rxjs';

Similarly, the way we import operators also changed in RxJS 6. To import catchError operator in RxJS 5, we use

import { catchError } from 'rxjs/operators/catchError';

In **RxJS 6**, it has changed to the following

import { catchError } from 'rxjs/operators';

In RxJS 6, we import all the operators from 'rxjs/operators'

import { map, delay, catchError } from 'rxjs/operators';

Many classes like ArrayObservable, EmptyObservable, ErrorObservable etc are also removed from v6, in favour of existing or new operators that perform the same operations.   
  
For example, in v5 to create an ErrorObservable we might use one of the following

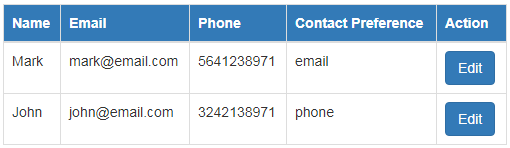
new ErrorObservable('Your error message');  
OR  
ErrorObservable.create('Your error message');

In v6, we use throwError() function to achieve this.

return throwError('Your error message');

How do I know, I have to use throwError() function instead of ErrorObservable class. Well, the following GitHub article contains all the differences between RxJS v5.x and v6. A quick search (CTRL + F) on the page for ErrorObservable shows, it has been removed in favour of throwError() function.  
<https://github.com/ReactiveX/rxjs/blob/master/docs_app/content/guide/v6/migration.md>  
  
in v6, import throwError function from rxjs. Since ErrorObservable class is replaced by throwError function, we import it the same way we import other classes like Observable and Subject from rxjs.

import { Observable, throwError } from 'rxjs';

**Implementing ListEmployeesComponent :**    
   
**list-employees.component.html**.

<div class="table-responsive">

  <table class="table table-bordered" \*ngIf="employees && employees.length">

    <thead>

      <tr class="bg-primary">

        <th>Name</th>

        <th>Email</th>

        <th>Phone</th>

        <th>Contact Preference</th>

        <th>Action</th>

      </tr>

    </thead>

    <tbody>

      <tr \*ngFor="let employee of employees">

        <td>{{ employee.fullName }}</td>

        <td>{{ employee.email }}</td>

        <td>{{ employee.phone }}</td>

        <td>{{ employee.contactPreference }}</td>

        <td> <button class="btn btn-primary">Edit</button> </td>

      </tr>

    </tbody>

  </table>

</div>

**list-employees.component.ts**

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

import { EmployeeService } from './employee.service';

import { IEmployee } from './IEmployee';

@Component({

  selector: 'app-list-employees',

  templateUrl: './list-employees.component.html',

  styleUrls: ['./list-employees.component.css']

})

export class ListEmployeesComponent implements OnInit {

  employees: IEmployee[];

  constructor(private \_employeeService: EmployeeService) { }

  ngOnInit() {

    this.\_employeeService.getEmployees().subscribe(

      (employeeList) => this.employees = employeeList,

      (err) => console.log(err)

    );

  }

}

**Changes in app.module.ts file**  
To be able to use EmployeeService in ListEmployeesComponent we have to register it. Since we want EmployeeService to be available across the entire application, Let's register it in the application root module AppModule.  
  
Import EmployeeService and include it in the providers array of @NgModule decorator of AppModule. Employee service uses angular's HttpClient service. To be able to use this we have to import HttpClientModule in the AppModule and include it in the imports array.

import { EmployeeService } from './employee/employee.service';

import { HttpClientModule } from '@angular/common/http';

@NgModule({

  declarations: [

    AppComponent,

    CreateEmployeeComponent,

    ListEmployeesComponent

  ],

  imports: [

    BrowserModule,

    AppRoutingModule,

    HttpClientModule,

    ReactiveFormsModule

  ],

  providers: [EmployeeService],

  bootstrap: [AppComponent]

})

export class AppModule { }

## Reactive Edit Forms

We will use **Create Employee Form**for both creating a new employee as well as editing an existing employee details.   
  
**Changes in app-routing.module.ts**

* Include a new route for editing an existing employee details in **app-routing.module.ts**.
* We will use **"create"** route to create a new employee.
* The new **"edit"** route will be for editing an existing employee details.
* Notice to the **"edit"** route we are passing the **id**of the employee we want to edit.

const appRoutes: Routes = [

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

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

  { path: 'edit/:id', component: CreateEmployeeComponent },

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

];

**Changes in list-employees.component.html :** Include click event binding on the **Edit**button.

<td><button class="btn btn-primary" (click)="editButtonClick(employee.id)">Edit</button></td>

**Changes in list-employees.component.ts :**Import Angular Router

import { Router } from '@angular/router';

Inject it into the component class using the constructor

constructor(private \_employeeService: EmployeeService,

            private \_router: Router) { }

Include editButtonClick() event handler method in the component class. When the Edit button is clicked, the user will be redirected to the "edit" route, passing it the id of the employee we want to edit.

editButtonClick(employeeId: number) {

  this.\_router.navigate(['/edit', employeeId]);

}

**Changes in create-employee.component.ts to support editing an existing employee**  
  
Import ActivatedRoute, EmployeeService, IEmployee and ISkill types

import { ActivatedRoute } from '@angular/router';

import { EmployeeService } from './employee.service';

import { IEmployee } from './IEmployee';

import { ISkill } from './ISkill';

Inject ActivatedRoute and EmployeeService into the component class using the constructor

constructor(private fb: FormBuilder,

            private route: ActivatedRoute,

            private employeeService: EmployeeService) { }

In ngOnInit(), read the id route parameter value. If it is truthy i.e if the id value exists on the route, then call the geEmployee(id) method passing it the employee id.    
  
Once employee data is retrieved and mapped to IEmployee type, it is passed to editEmployee(employee: IEmployee) method.

ngOnInit() {

  // Other existing code...

  this.route.paramMap.subscribe(params => {

    const empId = +params.get('id');

    if (empId) {

      this.getEmployee(empId);

    }

  });

}

getEmployee(id: number) method calls the EmployeeService and retrieves the existing employee details.

getEmployee(id: number) {

  this.employeeService.getEmployee(id)

    .subscribe(

      (employee: IEmployee) => this.editEmployee(employee),

      (err: any) => console.log(err)

    );

}

The following editEmployee() method updates the form controls with the employee data, so the data is displayed on the form and the end user can edit it.  
  
Notice we are using patchValue() method, to update the form controls with the employee data retrieved from server.

editEmployee(employee: IEmployee) {

  this.employeeForm.patchValue({

    fullName: employee.fullName,

    contactPreference: employee.contactPreference,

    emailGroup: {

      email: employee.email,

      confirmEmail: employee.email

    },

    phone: employee.phone

  });

}

**At the moment, we have 3 minor bugs when editing an existing employee.**  
  
Save button not disabled if the form is invalid : To disable the Save button bind disabled property of the button to the invalid property of the employeeForm

<button class="btn btn-primary" type="submit" [disabled]="employeeForm.invalid">Save</button>

We have a custom email validator attached to the email form control. This custom email validator fails validation if the email domain is not dell.com. However, the validation error message is not displayed until the form control is touched or dirty. As we are editing the existing employee details, it makes more sense to display the validation errors if the data is invalid rather than waiting until the email form control is touched or dirty.

To fix this, modify **logValidationErrors()** function as shown below.

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

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

    const abstractControl = group.get(key);

    this.formErrors[key] = '';

    // abstractControl.value !== '' (This condition ensures if there is a value in the

    // form control and it is not valid, then display the validation error)

    if (abstractControl && !abstractControl.valid &&

        (abstractControl.touched || abstractControl.dirty || abstractControl.value !== '')) {

      const messages = this.validationMessages[key];

      for (const errorKey in abstractControl.errors) {

        if (errorKey) {

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

        }

      }

    }

    if (abstractControl instanceof FormGroup) {

      this.logValidationErrors(abstractControl);

    }

  });

}

Similarly, if email and confirm email are not equal, the validation error is not displayed until the confirm email form control is dirty. To fix this include the following condition in matchEmail() function.

confirmEmailControl.value === ''

Here is the matchEmail() function with the above condition included.

function matchEmail(group: AbstractControl): { [key: string]: any } | null {

  const emailControl = group.get('email');

  const confirmEmailControl = group.get('confirmEmail');

  // If confirm email control value is not an empty string, and if the value

  // does not match with email control value, then the validation fails

  if (emailControl.value === confirmEmailControl.value

    || (confirmEmailControl.pristine && confirmEmailControl.value === '')) {

    return null;

  } else {

    return { 'emailMismatch': true };

  }

}

At the moment, the skills form array is not populated with the employees existing skills. We will discuss how to do this in our next video.

## **Angular populate formarray**

Populate angular formarray with existing data. We want to populate skills **FormArray**, with existing skills an employee has. When we load this data on a form to edit this employee details, we want to populate skills FormArray.   
To achieve this modify editEmployee() method in create-employee.component.ts as shown below.   
  
Notice we are using setControl() method to replace the skills FormArray with the FormArray that setExistingSkills() method returns.

editEmployee(employee: IEmployee) {

  this.employeeForm.patchValue({

    fullName: employee.fullName,

    contactPreference: employee.contactPreference,

    emailGroup: {

      email: employee.email,

      confirmEmail: employee.email

    },

    phone: employee.phone

  });

  this.employeeForm.setControl('skills', this.setExistingSkills(employee.skills));

}

**setExistingSkills() method**

* loops through each skill object of an employee
* Creates a FormGroup with 3 form controls (skillName, experienceInYears, proficiency)
* Pushes the FormGroup into the FormArray
* Finally, this FormArray with the existing skills of the employee is returned

setExistingSkills(skillSets: ISkill[]): FormArray {

  const formArray = new FormArray([]);

  skillSets.forEach(s => {

    formArray.push(this.fb.group({

      skillName: s.skillName,

      experienceInYears: s.experienceInYears,

      proficiency: s.proficiency

    }));

  });

  return formArray;

}

**Important - Programmatically changing a formarray in angular does not change dirty state :**At this point, if you remove one of the skill groups from the FormArray by clicking the "Delete Skill" button, notice the dirty and touched state of the form is still false.    
This is because, the state properties like dirty, touched etc are designed to indicate whether a user has interacted with the form.   
By default, programmatic change to value of a form control will not flip the value of these properties. However, in some cases you may need to mark form controls, form groups and form arrays as touched, dirty etc. In such cases you can explicitly do so by calling markAsDirty() and markAsTouched() methods.  
In our case, when a SKILL form group is removed from the FormArray we want to mark the formArray as touched and dirty. To achieve this, we are using markAsDirty() and markAsTouched() methods.

removeSkillButtonClick(skillGroupIndex: number): void {

  const skillsFormArray = <FormArray>this.employeeForm.get('skills');

  skillsFormArray.removeAt(skillGroupIndex);

  skillsFormArray.markAsDirty();

  skillsFormArray.markAsTouched();

}

## **Angular reactive forms put**

At component class level, include property of type IEmployee which holds data loaded from server for editing.

getEmployee(id: number) {

  this.employeeService.getEmployee(id)

    .subscribe(

      (employee: IEmployee) => {

        this.employee = employee; // Store employee object returned by API in employee property

        this.editEmployee(employee);

      },

      (err: any) => console.log(err)

    );

}

In the view template we have ngSubmit event bound to onSubmit() method

<form [formGroup]="employeeForm" (ngSubmit)="onSubmit()" class="form-horizontal">

onSubmit(): void {

  this.mapFormValuesToEmployeeModel();

  this.employeeService.updateEmployee(this.employee).subscribe(

    () => this.router.navigate(['list']),

    (err: any) => console.log(err)

  );

}

mapFormValuesToEmployeeModel() {

  this.employee.fullName = this.employeeForm.value.fullName;

  this.employee.contactPreference = this.employeeForm.value.contactPreference;

  this.employee.email = this.employeeForm.value.emailGroup.email;

  this.employee.phone = this.employeeForm.value.phone;

  this.employee.skills = this.employeeForm.value.skills;

}

At this point, you may be thinking can't I simply type cast employeeForm.value to IEmployee type.

this.employee = <IEmployee>this.employeeForm.value;

**No,** employeeForm.value does not match with the shape of IEmployee. In employeeForm.value, we do not have id property. Also, email and conifrmEmail properties are present in a nested FormGroup called emailGroup in the employeeForm, where as in the IEmployee interface we do not have such an email group property. We only have email property on th IEmployee interface. confirmEmail form control in the employeeForm is only there for validation. We do not need to save it on the server.   
  
Another approach is to use Object.assign() method as shown below. But this approach also will not work for us, because the employeeForm.value has an additional emailGroup property but not on the IEmploye interface.

this.employee = Object.assign({}, this.employee, this.employeeForm.value);

The updateEmployee() method of Angular EmployeeService issues a PUT request to the server side REST API.

updateEmployee(employee: IEmployee): Observable<void> {

    return this.httpClient.**put**<void>(`${this.baseUrl}/${employee.id}`, employee, {

        headers: new HttpHeaders({

            'Content-Type': 'application/json'

        })

    })

        .pipe(catchError(this.handleError));

}

When REST API call completes successfully, navigate user to list route.

Angular Router service is required to navigate the user to the list route. So, please make sure to import and inject Angular Router service into the component class.

import { Router } from '@angular/router';

constructor(private fb: FormBuilder,

  private route: ActivatedRoute,

  private employeeService: EmployeeService,

  private router: Router) { }

click the **save**button, the edited data should be saved and redirected to list route. see saved changes in **db.json**file.

## **Angular Reactive Forms Post**

If we get to the CreateEmployeeComponent and in the route if we do not have ID parameter, then we know we are creating a new employee and not editing an existing employee. modify **ngOnInit()** as shown below. Notice in the ELSE block, we are initialising a new empty Employee object.

this.route.paramMap.subscribe(params => {

  const empId = +params.get('id');

  if (empId) {

    this.pageTitle = 'Edit Employee';

    this.getEmployee(empId);

  } else {

    this.pageTitle = 'Create Employee';

    this.employee = {

      id: null,

      fullName: '',

      contactPreference: '',

      email: '',

      phone: null,

      skills: []

    };

  }

});

modify code in **onSubmit()** method. Check if the id property on employee object is truthy. IF it is, then we know we are editing an employee, so call updateEmployee() of the EmployeeService which issues a PUT request to the REST API.

ELSE, we know we are creating a new employee. So in this case, call addEmployee()

onSubmit(): void {

  this.mapFormValuesToEmployeeModel();

  if (this.employee.id) {

    this.employeeService.updateEmployee(this.employee).subscribe(

      () => this.router.navigate(['list']),

      (err: any) => console.log(err)

    );

  } else {

    this.employeeService.addEmployee(this.employee).subscribe(

      () => this.router.navigate(['list']),

      (err: any) => console.log(err)

    );

  }

}

Here is the addEmployee() method of the EmployeeService for your reference.

addEmployee(employee: IEmployee): Observable<IEmployee> {

    return this.httpClient.post<IEmployee>(this.baseUrl, employee, {

        headers: new HttpHeaders({

            'Content-Type': 'application/json'

        })

    })

    .pipe(catchError(this.handleError));

}

<div class="panel-heading">

  <h3 class="panel-title">{{pageTitle}}</h3>

</div>

# Angular Modules

Angular Moduleis a class decorated with @NgModuledecorator. It is a mechanism to group components, directives, pipes and services that are related to a feature area of an angular application.   
  
For example, if you are building application to manage employees, you might have following features in application.

|  |  |
| --- | --- |
| **Application Feature** | **Description** |
| Employee Feature | Deals with creating, reading, updating and deleting employees |
| Login Feature | Deals with login, logout, authenticate and authorize users |
| Report Feature | Deals with generating employee reports like total number of employees by department, |

To group the components, directives, pipes and services related to a specific feature area, we create a module for each feature area. These modules are called **feature modules**.   
  
In addition to feature modules, an Angular application also contains the following modules.

|  |  |
| --- | --- |
| **Module Type** | **Description** |
| Root Module | Every Angular application has at least one module, the root module(**AppModule)**. We bootstrap this root module to launch the application. If the application that you are building is a simple application with a few components, then all you need is the root module. As the application starts to grow and become complex, in addition to the root module, we may add several feature modules. We then import these feature modules to root module. |
| Core Module | The most use of this module is to include the providers of http services. Services in Angular are usually singletons. So to ensure that, only one instance of a given service is created across the entire application, we include all our singleton service providers in the core module. In most cases, a CoreModule is a pure services module with no declarations. The core module is then imported into the root module (AppModule) only. CoreModule should never be imported in any other module. |
| Shared Module | This module contains reusable components, directives, and pipes. The Shared module is then imported into specific Feature Modules as needed. The Shared module might also export the commonly used Angular modules like CommonModule, FormsModule etc. so they can be easily used across your application, without importing them in every Feature Module. |
| Routing Modules | We can have one or more routing modules for application level routes and feature module routes |

**What are the advantages of splitting an angular application into multiple Angular Modules**

|  |  |
| --- | --- |
| **Benefit** | **Description** |
| Organizing Angular Application | Modules are a great way to organise an angular application. Every feature area is present in it's own feature module. All Shared pieces (like components, directives & pipes) are present in a Shared module. All Singleton services are present in a core module. As we clearly know what is present in each module, it's easier to understand, find and change code if required |
| Code Reuse | Modules are great way to reuse code. For example, if you have components, directives or pipes that you want to reuse, you include them in a Shared module and import it into the module where you need them rather than duplicating code. |
| Code Maintenance | Since Angular Modules promote code reuse and separation of concerns, they are essential for writing maintainable code in angular projects |
| Performance | Angular modules can be loaded either eagerly when the application starts or lazily on demand when they are actually needed or in the background. Lazy loading angular modules can significantly boost the application start up time. |

**@NgModule Decorator**   
The @NgModule decorator has the following properties: declarations, bootstrap, providers, imports, exports

In preparation for refactoring our application into multiple modules, let's create the following 2 components

HomeComponent and PageNotFoundComponent

ng g c home --flat

ng g c page-not-found --flat

**home.component.html**

<div class="panel panel-primary">

  <div class="panel-heading">

    <h3 class="panel-title">Employee Management System</h3>

  </div>

  <div class="panel-body">

    <img src="../assets/images/Employees.jpg" class="img-responsive"/>

  </div>

</div>

**Please note :**Create **images**folder in the **assets**folder. Download **Employees.jpg** and place it in the **images**folder.   
[](https://3.bp.blogspot.com/-ZHkznRxGNrU/XAP3RcV8O4I/AAAAAAAArcU/AlrfBPZcmKAioa7xX4a926LVRlY-ob56gCLcBGAs/s1600/Employees.jpg)   
**page-not-found.component.html**

<h1>The page you are looking for cannot be found.</h1>

Include home and wild card routes in **app-routing.module.ts**

import { HomeComponent } from './home.component';

import { PageNotFoundComponent } from './page-not-found.component';

const appRoutes: Routes = [

  { path: 'home', component: HomeComponent },   // home route

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

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

  { path: 'edit/:id', component: CreateEmployeeComponent },

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

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

  { path: '\*\*', component: PageNotFoundComponent }// wild card route

];

In the root component (**app.component.html**) include a menu item for the home route

<div class="container">

    <nav class="navbar navbar-default">

        <ul class="nav navbar-nav">

            <!-- Include this Home menu item for the home route-->

            <li><a routerLinkActive="active" routerLink="home">Home</a></li>

            <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>

## Creating Feature Module

At the moment all our application components, directives, pipes and services are in this one module in Root module (AppModule) app.module.ts. As we add more features, this module is going to get more complex and extremely difficult to maintain.   
  
What we want to do now is move all the **EMPLOYEE**feature related components, directives, pipes and services into a separate feature module. Let's name this new feature module - **EmployeeModule**. 

ng g m employee/employee --flat -m app

* Creates the EmployeeModule in a file with name employee.module.ts.
* Imports EmployeeModule into the root module - AppModule.

**employee.module.ts**

// employee.module.ts

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

// Exports all the basic Angular directives and pipes such as NgIf, NgFor, DecimalPipe etc.

import { CommonModule } from '@angular/common';

// CreateEmployeeComponent uses ReactiveFormsModule directives such as

// formGroup so ReactiveFormsModule needs to be imported into this Module

// An alternative approach would be to create a Shared module and export

// the ReactiveFormsModule from it, so any other module that needs

// ReactiveFormsModule can import it from the SharedModule.

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

// Import and declare the components that belong to this Employee Module

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

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

@NgModule({

  imports: [

    CommonModule,

    ReactiveFormsModule

  ],

  declarations: [

    CreateEmployeeComponent,

    ListEmployeesComponent

  ],

  // If you want the components that belong to this module, available to

  // other modules, that import this module, then include all those

  // components in the exports array. Similarly you can also export the

  // imported Angular Modules

  // exports: [

  //   CreateEmployeeComponent,

  //   ReactiveFormsModule

  // ]

})

export class EmployeeModule { }

**Browser Module v/s Common Module**  
In the root module(AppModule), we do not have to import CommonModule explicitly because the BrowserModule imports and re-exports CommonModule. So all the directives and pipes provided by the CommonModule are available in the RootModule, because root module imports BrowserModule.   
  
BrowserModule provides services that are essential to launch and run a browser application. BrowserModule should be imported only once and that too only by the root module.  
  
Since we have moved the following components to EmployeeModule, we can remove them from the RootModule.

* CreateEmployeeComponent
* ListEmployeesComponent

Similarly ReactiveFormsModule is also not required in the root module. ReactiveFormsModule components, directives and pipes are only needed in EmployeeModule so we moved it there.

// app.module.ts

import { BrowserModule } from '@angular/platform-browser';

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

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

import { HttpClientModule } from '@angular/common/http';

import { AppRoutingModule } from './app-routing.module';

import { AppComponent } from './app.component';

import { HomeComponent } from './home.component';

import { PageNotFoundComponent } from './page-not-found.component';

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

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

import { EmployeeModule } from './employee/employee.module';

import { EmployeeService } from './employee/employee.service';

@NgModule({

  declarations: [

    AppComponent,

~~CreateEmployeeComponent,~~

~~ListEmployeesComponent,~~

    HomeComponent,

    PageNotFoundComponent

  ],

  imports: [

    BrowserModule,

    AppRoutingModule,

~~ReactiveFormsModule,~~

    HttpClientModule,

EmployeeModule

  ],

  providers: [EmployeeService],

  bootstrap: [AppComponent]

})

export class AppModule { }

We can still simplify the code in root module. Notice the EmployeeService is still in the root module. We can move this service to the EmployeeModule or CoreModule. We will discuss this in our upcoming videos.  
  
At the moment all our application routes, including the routes to **LIST, CREATE & EDIT**employees are in one routing module - AppRoutingModule. We will discuss moving EMPLOYEE feature related routes to a separate routing module.

## **Creating Feature Routing Module**

At the moment, all our application routes, including the EMPLOYEE feature module routes like LIST, CREATE, EDIT are present in AppRoutingModule. This module is in **app-routing.module.ts** file.   
  
For separation of concerns and ease of maintenance, here is what we want to do.

Include all the application level routes like HOME, EMPTY PATH & WILD CARD routes in the AppRoutingModule

Include all the EMPLOYEE feature module routes like LIST, CREATE & EDIT in a separate EmployeeRoutingModule

**Creating Employee feature routing module:**  
The following is the common naming convention used when creating a separate routing module for a feature module.  
Feature routing module file name convention: featureModule-routing.module.ts  
Feature routing module class name convention: FeatureModuleRoutingModule  
  
**For example if your feature module name is employee, then the**

Routing Module File Name is employee-routing.module.ts

Routing Module Class Name is EmployeeRoutingModule

**ng g m employee/employee-routing --flat -m app**

employee-routing.module.ts

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

// Import RouterModule & Routes type

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

// Import all the components that we will be referencing in the route definitions

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

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

// Define the routes

const appRoutes: Routes = [

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

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

  { path: 'edit/:id', component: CreateEmployeeComponent },

];

// In a feature module forChild() method must be used to register routes

// Export RouterModule, so the it's directives like RouterLink, RouterOutlet

// are available to the EmployeeModule that imports this module

@NgModule({

  imports: [ RouterModule.forChild(appRoutes) ],

  exports: [ RouterModule ]

})

export class EmployeeRoutingModule { }

**RouterModule forRoot vs forChild**  
  
forRoot() method registers the specified routes. It also creates an instance of the Router service and registers it with the angular's dependency injector.  
  
forChild() method on the other hand only registers the additional specified routes and tells angular to reuse the Router service instance that forRoot has created.

**Angular services are singletons**. So, to ensure that, there is only one instance of Router service, forRoot() method should be called only once in the main application routing module.   
  
In all the feature routing modules forChild() method should be used to register the additional routes.

When forChild() method is called, Angular Router knows it has to only register the additional specified routes and not to re-register the Angular Router service.  
  
**Step 3 :**Import EmployeeRoutingModule into EmployeeModule

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

import { CommonModule } from '@angular/common';

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

// Import the EmployeeRoutingModule

import { EmployeeRoutingModule } from './employee-routing.module';

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

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

@NgModule({

  imports: [

    CommonModule,

    ReactiveFormsModule,

    // Add EmployeeRoutingModule to the imports array

    EmployeeRoutingModule

  ],

  declarations: [

    CreateEmployeeComponent,

    ListEmployeesComponent

  ]

})

export class EmployeeModule { }

**Step 4 :**Remove the EMPLOYEE feature module routes from AppRoutingModule (app-routing.module.ts)

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

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

import { HomeComponent } from './home.component';

import { PageNotFoundComponent } from './page-not-found.component';

const appRoutes: Routes = [

  { path: 'home', component: HomeComponent },

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

  { path: '\*\*', component: PageNotFoundComponent }

];

@NgModule({

  imports: [ RouterModule.forRoot(appRoutes) ],

  exports: [ RouterModule ]

})

export class AppRoutingModule { }

**All angular feature modules should be imported before AppRoutingModule**   
At this point if you launch the application, except the HOME route, all the routes (LIST, CREATE, EDIT) displays the PageNotFoundComponent template. 

When you comment the following WILD CARD route in the AppRoutingModule, the rest of the routes (LIST, CREATE, EDIT) work. If you uncomment the WILD CARD route, the EMPLOYEE feature module routes stop working again.

const appRoutes: Routes = [

  { path: 'home', component: HomeComponent },

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

  // { path: '\*\*', component: PageNotFoundComponent }

];

**Why are the EMPLOYEE feature module routes (LIST, CREATE, EDIT) not working**  
This is because of the order in which AppRoutingModule and EmployeeModule are imported in the root module AppModule. Notice at the moment, AppRoutingModule is imported before EmployeeModule in the AppModule.

@NgModule({

  declarations: [

    AppComponent,

    HomeComponent,

    PageNotFoundComponent

  ],

  imports: [

    BrowserModule,

    AppRoutingModule,

    HttpClientModule,

    EmployeeModule

  ],

  providers: [EmployeeService],

  bootstrap: [AppComponent]

})

export class AppModule { }

Because AppRoutingModule is imported before EmployeeModule, the routes will be ordered as shown below. Notice, the WILD CARD route is before all the EMPLOYEE feature module routes. Because of this we will never be able to get to the LIST,CREATE and EDIT routes.

{ path: 'home', component: HomeComponent },

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

{ path: '\*\*', component: PageNotFoundComponent },

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

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

{ path: 'edit/:id', component: CreateEmployeeComponent },

To fix this all we have to do is, change the import order of the modules in the AppModule. Import EmployeeModule before AppRoutingModule as shown below.

  imports: [

    BrowserModule,

    EmployeeModule,

    AppRoutingModule,

    HttpClientModule,

  ],

With the above change the routes will be ordered as shown below and all our EMPLOYEE feature module routes work as expected. For this reason all feature modules should be imported before AppRoutingModule.

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

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

{ path: 'edit/:id', component: CreateEmployeeComponent },

{ path: 'home', component: HomeComponent },

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

{ path: '\*\*', component: PageNotFoundComponent },

## Creating Shared Module

Shared module contains all the commonly used directives, pipes, and components that we want to share with other modules that import this shared module.    
  
**Things to consider when creating a shared module**

* The SharedModule may re-export other common angular modules, such as CommonModule, FormsModule,

ReactiveFormsModule etc. Instead of writing the same code in every feature module to import these commonly used Angular modules we can re-export them from a SharedModule, so these commonly used Angular Modules are available to all the feature modules that import this SharedModule.

* The SharedModule should not have providers. This is because, lazy loaded modules create their own branch on the Dependency Injection tree. As a result of this, if a lazy loaded module imports the shared module, we end up with more than one instance of the service provided by the shared module. For this same reason, the SharedModule should not import or re-export modules that have providers.
* The SharedModule is then imported by all the FeatureModules where we need the shared functionality. The SharedModule can be imported by both - eager loaded FeatureModules as well as lazy loaded FeatureModules. We will discuss eager and lazy loading modules in our upcoming videos.

**SharedModule (shared.module.ts)**

ng g m shared/shared --flat -m employee/employee

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

import { CommonModule } from '@angular/common';

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

@NgModule({

  imports: [

    // At the moment we do not have any components in this SharedModule that require directives

// of CommonModule or ReactiveFormsModule. So we did not include them here in the imports

// array. We can export a module without importing it first

  ],

  declarations: [],

  // Our Employee FeatureModule requires the CommonModule directives such as ngIf, ngFor etc.

// Similarly, Employee FeatureModule also requires ReactiveFormsModule directives. So export

// CommonModule and ReactiveFormsModule.

  exports: [

    CommonModule,ReactiveFormsModule

  ]

})

export class SharedModule { }

In EmployeeModule (employee.module.ts), remove CommonModule and ReactiveFormsModule references as these modules are now provided by the imported SharedModule.

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

import { EmployeeRoutingModule } from './employee-routing.module';

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

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

import { SharedModule } from '../shared/shared.module';

@NgModule({

  imports: [

    EmployeeRoutingModule, SharedModule,

  ],

  declarations: [

    CreateEmployeeComponent, ListEmployeesComponent

  ]

})

export class EmployeeModule { }

## **Grouping & creating ComponentLess route**

**employee-routing.module.ts**

const appRoutes: Routes = [

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

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

  { path: 'edit/:id', component: CreateEmployeeComponent },

];

All the routes in an angular module that you want to lazy load should have the same route prefix. At the moment, the above 3 routes does not have a common route prefix.

const appRoutes: Routes = [

  {

    path: 'employees', // parent route with path employees.

    children: [ // parent 'employees' route has 3 child routes

      { path: '', component: ListEmployeesComponent },

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

      { path: 'edit/:id', component: CreateEmployeeComponent },

    ]

  }

];

All the 3 child routes will be pre-fixed with the parent route path – employees.

Parent route(employees) doesn’t have component associated with it. This route is called a **component less route**.

With above route configuration, we have following routes. All routes in this module have same route prefix.

|  |  |
| --- | --- |
| **Route Path** | **Description** |
| /employees | Displays the list of all employees |
| /employees/create | Allows to create a new employee |
| /employees/edit/1 | Allows to create a edit an existing employee |

Update the routes in the navigation menu in **app.component.html**

<li><a routerLinkActive="active" routerLink="employees">List</a></li>

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

In **list-employees.component.ts**, modify the code to redirect to the new EDIT route (employees/edit/id)

editButtonClick(employeeId: number) {

  this.\_router.navigate(['/employees/edit', employeeId]);

}

In **create-employee.component.ts**, modify the code to redirect to the new 'employees'route

onSubmit(): void {

  this.mapFormValuesToEmployeeModel();

  if (this.employee.id) {

    this.employeeService.updateEmployee(this.employee).subscribe(

      () => this.router.navigate(['employees']),

      (err: any) => console.log(err)

    );

  } else {

    this.employeeService.addEmployee(this.employee).subscribe(

      () => this.router.navigate(['employees']),

      (err: any) => console.log(err)

    );

  }

}

## **Lazy Loading Angular Modules**

At the moment we have 2 modules: Root application module – AppModule, Feature module - EmployeeModule  
As you add more features to application you will have more feature modules like ReportsModule, AdminModule etc

Overall application size will grow. At some point you'll reach tipping point where application takes long time to load.    
  
Unless, you are using lazy loading, all the modules are eagerly loaded. All the modules and their associated components, directives, pipes and services must be downloaded from the server, when user first visits the application.   
  
To address this problem, we use **asynchronous routing**, which loads feature modules lazily, on demand. This can significantly reduce the initial load time of your application.   
  
At the moment, the 2 modules (AppModule & EmployeeModule) in our application are eagerly loaded.

We want to lazily load EmployeeModule. To lazy load a module, it has to meet 2 requirements.

* All the routes in the angular module that you want to lazy load should have the same route prefix
* The module should not be referenced in any other module. If it is referenced, the module loader will eagerly load it instead of lazily loading it.

At the moment, our application does not meet the second requirement.EmployeeModule that we want to lazy load is referenced in the AppModule. So please **delete** the EmployeeModule references in the AppModule (app.module.ts)   
At this point, if we navigate to any of the EmployeeModule routes (i.e /employees or employees/create or employees/edit/2), we see the PageNotFoundComponent template displayed.   
  
To fix this, **we have to lazy load the EmployeeModule**. To achieve this, include the following route in app-routing.module.ts file. This new route, lazily loads the EmployeeModule. Make sure the below route is before the wild card route in the AppRoutingModule. Otherwise we would not be able to get to any of the EmployeeModule routes.

{ path: 'employees', loadChildren: './employee/employee.module#EmployeeModule' }

At this point, if we navigate to any EmployeeModule routes (i.e /employees or employees/create or employees/edit/2),

we see either an empty page or the PageNotFoundComponent template displayed.   
  
But if we include an extra /employees in the path as shown below, then all EmployeeModule routes work as expected

|  |
| --- |
| /employees/employees |
| /employees/employees/create |
| /employees/employees/edit/2 |

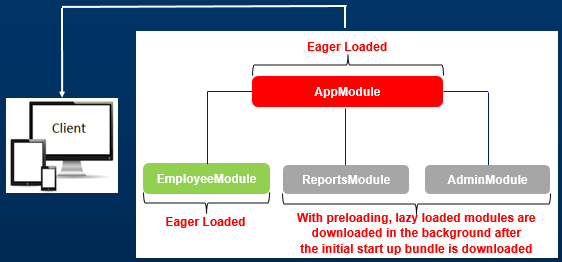
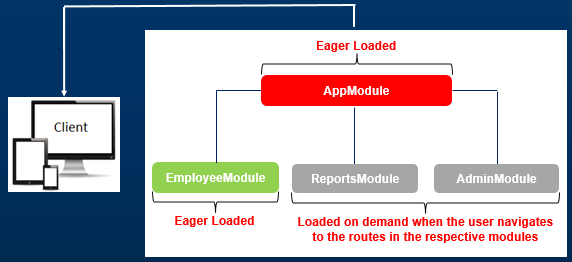
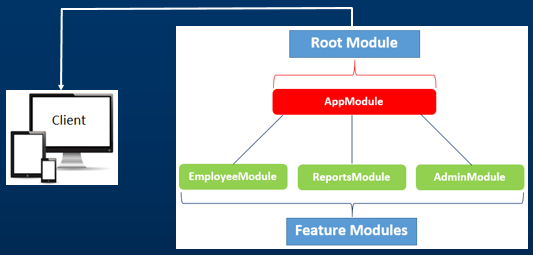
To fix this, in **employee-routing.module.ts** file, change following routes. We do not need parent route - 'employees'

const appRoutes: Routes = [  
~~{~~  
~~path: 'employees',  
    children: [~~  
      { path: '', component: ListEmployeesComponent },  
      { path: 'create', component: CreateEmployeeComponent },  
      { path: 'edit/:id', component: CreateEmployeeComponent },  
~~]  
  }~~  
];

With the above changes, the **Employee feature module should be lazily loaded**. You can confirm this by looking at the **Network**tab on the **Browser Developer Tools**.

## **Preloading Angular Modules**

Module loading strategies in Angular: **Eager** Loading, **Lazy** Loading, **Pre**loading



**Eager loading :**All modules must be downloaded onto the client machine before the application starts.

* If we do not do anything special, by default, the angular modules are eagerly loaded
* The root application module (AppModule) is always eagerly loaded
* Only the first request to the application takes a long time, but the subsequent requests from that same client will be faster. This is because, with eager loading, all the modules must be loaded before the application starts.
* There is nothing special that we have to do, for an Angular module to be eager loaded. It just needs to be referenced in the application using imports metadata of @NgModule decorator.

**Lazy loading :** Lazy loaded modules are loaded when user navigates to routes in those respective modules.

* To lazy load a module, it should not be referenced in any other module. If it is referenced, the module loader will eagerly load it instead of lazily loading it.
* Only the root module and other essential modules that user expects to see when the application first starts are loaded. In our example here, only the root module and employee module are loaded when the application starts. The rest of the modules i.e ReportsModule and AdminModule are configured to be lazy loaded so they are not loaded when the application starts.

**Preloading :** Application can download lazy loaded module in the background after the initial bundle that is required to start the application is downloaded. Preloading is same as lazy loading but happens slightly differently.

* First, the module to bootstrap the application and eager loaded modules are downloaded.
* At this point, we have the application up and running and the user is interacting with the application.
* While application has nothing to download, it downloads angular modules configured to preload in background.
* So, by time user navigates to a route in a lazy loaded module, it is already pre-loaded, so the user does not have to wait, and sees the component associated with that route right away. Preloading is also often called Eager Lazy Loading

**Configuring Preloading in Angular :**  
**Step 1 :**Import PreloadAllModules type from @angular/router package

import { PreloadAllModules } from '@angular/router';

**Step 2 :** Set preloadingStrategy to PreloadAllModules. We do this in the configuration object that we pass as a second parameter to the forRoot() method of the RouterModule class.

@NgModule({  
  imports: [  
    RouterModule.forRoot(appRoutes, { preloadingStrategy: PreloadAllModules })  
  ],  
  exports: [ RouterModule ]  
})  
export class **AppRoutingModule** { }

**The value for preloadingStrategy property can be one of the following**

|  |  |
| --- | --- |
| **Value** | **Description** |
| NoPreloading | This is the default and does not preload any modules |
| PreloadAllModules | Preloads all modules as quickly as possible in the background |
| Custom Preload Strategy | We can also specify our own custom preloading strategy. |

## **Custom Preloading Strategy**

To control when a lazy loaded module is downloaded onto the client machine, we use preloadingStrategy property.   
Let us say in our Angular project, we have 2 lazy loaded modules: EmployeeModule, AdminModule.

If we set, preloadingStrategy: PreloadAllModules, then both lazy loaded modules is preloaded in background.

RouterModule.forRoot(appRoutes, {preloadingStrategy: PreloadAllModules})

If we set preloadingStrategy:NoPreloading, then none of the lazy loaded modules will be preloaded.

RouterModule.forRoot(appRoutes, {preloadingStrategy: NoPreloading})

We have to create **Custom Preloading Strategy** if we want to preload EmployeeModule, but not the AdminModule  
  
**Step 1 :** create a service that implements Angular's built-in PreloadingStrategy abstract class

ng g s CustomPreloading

**custom-preloading.service.ts**.

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

import { PreloadingStrategy, Route } from '@angular/router';

import { Observable, of } from 'rxjs';

@Injectable({

  providedIn: 'root'

})

export class CustomPreloadingService implements PreloadingStrategy {

  constructor() { }

  preload(route: Route, fn: () => Observable<any>): Observable<any> {

    // If data property exists on route of lazy loaded module and if that data property

// also has preload property set to true, then return the fn() which preloads the module

    if (route.data && route.data['preload']) {

      return fn();

    // If data property does not exist or preload property is set to

    // false, then return Observable of null, so the module is not preloaded in the background

    } else {

      return of(null);

    }

  }

}

**app-routing.module.ts**, import CustomPreloadingService, Set CustomPreloadingService as Preloading Strategy

import { CustomPreloadingService } from './custom-preloading.service';

RouterModule.forRoot(appRoutes, {  
  preloadingStrategy: CustomPreloadingService  
})

Modify **'employees'** route, and set preload property to true or false. Set it to true if you want EmployeeModule

to be preloaded else false.

const appRoutes: Routes = [

  { path: 'home', component: HomeComponent },

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

  {

    path: 'employees',

    data: { preload: true },

    loadChildren: './employee/employee.module#EmployeeModule'

  },

  { path: '\*\*', component: PageNotFoundComponent }

];