Angular

1.Changing DateOfBirth into Blue theme.

2.To make week number off then need to showWeekNumbers:false”­.

example:

this.datePickerConfig = Object.assign({}, { containerClass: "theme-dark-blue" ,showWeekNumbers:false});

3.To maintained min or max date .

Example:

constructor() {

this.datePickerConfig = Object.assign({}, {

containerClass: "theme-dark-blue" ,

showWeekNumbers:false,

minDate:new Date(2018, 0, 1),

maxDate:new Date(2018,11,31)});

}

4.to make date format need to update in ts

constructor() {

this.datePickerConfig = Object.assign({}, {

containerClass: "theme-dark-blue" ,

showWeekNumbers:false,

minDate:new Date(2018, 0, 1),

maxDate:new Date(2018,11,31),

dateInputFormat:'DD/MM/YYYY'

});

}

5.To get by default date need to update in create.ts file.

dateOfBirth: Date= new Date(2018,11,30);

6.to get DateOfBirth normal size not so big as like above all then need to update in create.html file.

Example:

<div class="row">

<div class="form-group col-md-5">

<label for="dateOfBirth">Date of Birth</label>

<input type="text" [(ngModel)]="dateOfBirth" [bsConfig]="datePickerConfig" name="dateOfBirth" class="form-control" bsDatepicker>

</div>

</div>

7.to make DateOfBirth place either left,right,Middel have to change.

<div class="row">

<div class="form-group col-md-5">

<label for="dateOfBirth">Date of Birth</label>

<input type="text" [(ngModel)]="dateOfBirth" [bsConfig]="datePickerConfig"

name="dateOfBirth" class="form-control" bsDatepicker placement="right">

</div>

</div>

Note:

The DatePickerConfig object can be used to configure

Min-date

Max-date

Show/hide Week Numbers

Date Format

Etc...

Lecture:13

**Angular ngif directive**

**Export class create-employee.component{**

previewPhoto= false;

togglePhotoPreview(){

this.previewPhoto= !this.previewPhoto;

}

**}**

<div class="form-group">

<label for="photoPath">Photo Path</label>

<input type="text" class="form-control" id="photoPath" [(ngModel)]="photoPath" name="photoPath">

</div>

<div class="form-group">

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

{{previewPhoto?"Hide":"Show"}} Preview</button>

</div>

<div class="form-group">

<img [src]="photoPath" height="200" width="200" \*ngIf="previewPhoto" />

</div>

1. **Every time clicking show preview button then submit alys executing to fix this issue need to write type=”button” in create.html.**

**Example->**

</div>

<div class="form-group">

<button class="btn btm-primary" type="button" (click)="togglePhotoPreview()" >Show Preview</button>

</div>

2.now some enhacement if photo has been showed then need one button which will behave as a Hide Preview or Show Vise-Versa...

Example->

<div class="form-group">

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

{{previewPhoto?"Hide":"Show"}} Preview</button>

</div>

Lecture 14.

Angular disable browser validation

1. By default Angular 4and later version disable brower native Validation.

How to Enable Browser validation.

2.to make enable

<form #employeeForm="ngForm" ngNativeValidate (ngSubmit)="saveEmployee(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 for="fullName">Full Name</label>

<input id="fullName" required [(ngModel)]="fullName" name="Fullname" type="text" class="form-control" placeholder="Enter your Fullname"

/>

</div>

**Lecture #14.** **Angular form validation**

**Angular** form validation properties->

**1.Touched**

**2.Un-Touched**

**3.Pristine**

**4.Dirty**

**5.Valid**

**6.Invalid**

**HTML5 validation Attributes**

**1.required**

**2.maxLength**

**3.pattern**

**4.min**

**5.max**

**To check validation properties we will do some changes**

<div class="panel-body">

<div class="form-group">

<label for="fullName">Full Name</label>

<input id="fullName" required [(ngModel)]="fullName" name="Fullname" type="text" class="form-control" placeholder="Enter your Fullname"

#fullNameControl="ngModel" />

</div>

<div>

touched ->{{fullNameControl.touched}}<br>

un-touched ->{{fullNameControl.untouched}}

</div>

Note->

1.To include the html5 validation attribute such as requied for example:

<input id="fullName" type=”text”/>

2.Export ngModel directive to a local templte variable

<input id=”fullName” required #fullNameControler=”ngModel” type=”text”/>

3.use the local template variable to access the validation properties (touched,untouched,dirty,valid,etc).

<div>touch{{ fullNameControler .touched}}</div>

<div>untouch{{ fullNameControler .untouched}}</div>

<div>dirty{{ fullNameControler .dirty}}</div>

4.to check entire for validation properties

Export ngForm directive to a local template reference variable.

<form #employeeForm="ngForm" ngNativeValidate (ngSubmit)="saveEmployee(employeeForm)">

5.use the template reference variable to access the validation properties at form lavel (touched,valid,dirty).

<div>touch{{ employeeForm.touched}}</div>

<div>untouch{{ employeeForm.untouched}}</div>

<div>dirty{{ employeeForm.dirty}}</div>

**Lecture #16 .** **Displaying angular form validation error messages**

**BootStarp classes for styling validation error messages.**

**->has-error**

**->control-label**

**->help-block**

<div class="form-group" [class.has-error]="fullNameControl.invalid && fullNameControl.touched" [class.has-success]="fullNameControl.valid">

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

<input id="fullName" required [(ngModel)]="fullName" name="Fullname" type="text" class="form-control" placeholder="Enter your Fullname"

#fullNameControl="ngModel" />

<span class="help-block" \*ngIf="fullNameControl.invalid && fullNameControl.touched" >FullName Required</span>

</div>

**Note->**

**To make from submit button disable.**

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

**Lecture #17.** **Model binding in angular template driven forms**

1.Eailer discussed Binding Angular form to our own Model.

Error-> can not assign a variable or reference.

At the moment, in **CreateEmployeeComponent**we are using the Angular Auto-generated form model. Instead of using the Angular generated form model, we can use our model class.

In employee.model.ts file in the models folder, we have Employee class. We want to use this class as the model when creating a new employee. Here are the steps.

**Step 1 :** In create-employee.component.ts file, import the Employee model  
import { Employee } from '../models/employee.model';

**Step 2 :** In CreateEmployeeComponent class, include **employee**property. Notice we have set the type to Employee and initialised all properties with NULL value.   
  
export class CreateEmployeeComponent implements OnInit {  
  employee: Employee = {  
    id: null,  
    name: null,  
    gender: null,  
    contactPreference: null,  
    phoneNumber: null,  
    email: null,  
    dateOfBirth: null,  
    department: null,  
    isActive: null,  
    photoPath: null  
  };

**Step 3 :** In the view template, bind the ngModel directive of an input field to it's corresponding property on the employee object. The employee property we created in Step 2 returns an employee object, which is the model for our form.   
  
For example, bind ngModel directive on the email input field to the email property on the employee object.  
[(ngModel)]="employee.email"

Except fullName, bind the ngModel directive of the rest of the input fields with the corresponding properties on the employee object.   
  
In the employee class we do not have **fullName**property. we have **name**instead. On the view template, the corresponding input field name is fullName. To keep things consistent let's change fullName to name on the label and the input field as shown below. 

<div class="form-group" [class.has-error]="name.invalid && name.touched">

  <label for="name" class="control-label">Name</label>

  <input id="name" required type="text" class="form-control" name="name"

         [(ngModel)]="name" #name="ngModel">

  <span class="help-block" \*ngIf="name.invalid && name.touched">

    Name is required

  </span>

</div>

At this point, if you view the page in the browser, you will see the following error.  
**Cannot assign to a reference or variable**   
  
We get this error because, Angular generated form model creates **name** property and we are also creating a local template variable with the same name by exporting **ngModel**to **#name**. Hence we get the error - Cannot assign to a reference or variable.   
  
One way to fix this error is, by giving our local template reference variable a different name other than ~~name~~. So if we change #name="ngModel" to #nameControl="noModel" the error goes away. We discussed this in detail in [Part 15](https://www.youtube.com/watch?v=d8XONHXTv_4) of [Angular CRUD tutorial](https://www.youtube.com/playlist?list=PL6n9fhu94yhXwcl3a6rIfAI7QmGYIkfK5).   
  
The other way to fix this error is by using our own model. Using the **ngModel**directive, bind the **name**property of the employee object to the **name**input field 

[(ngModel)]="employee.name"

At this point, if you view the page in the browser and notice the error is gone and all the properties in the Angular generated form model are NULL as expected.   
  
To see our own employee model, include the following code in the view template file (create-employee.component.html) 

Angular Generated Forom Model : {{employeeForm.value | json}}

<br/>

<br/>

Our Employee Model : {{ employee | json}}

At this point, on the browser we should see both - Angular generated form model and our own employee model. Notice as we change the values in the input fields, the respective properties in both the models are updated as expected.   
  
At the moment, when we click the "Save" button, we are logging the **employeeForm.value**to the console. We instead want to log our employee model object. To do this   
In the view template, pass the employee object to the saveEmployee() method.

<form #employeeForm="ngForm" (ngSubmit)="saveEmployee(employee)">

Modify saveEmployee() method in create-employee.component.ts file as shown below.   
  
saveEmployee(newEmployee: Employee): void {  
  console.log(newEmployee);  
}   
  
At this point, when we click the Save button, the employee object is logged to the console as expected.

**Lecture #18.** **Angular email validation example**

**In this video we will discuss** 

* Validating Email form field in Angular
* Using multiple validators on a single input field
* Angular safe navigation operator

**Email validation in Angular :**There are 2 different ways to validate email form field in Angular. We can either use **pattern validator**or **email validator**. Email validator is introduced in Angular 4. So if you are using Angular 4 or later version you may use email validator or pattern validator to validate email. If you are using Angular 2, then your only choice is to use Pattern validator.   
  
  
In this video we will discuss using the Angular built-in **Email validator**and in our next video we will discuss using the **Pattern validator**.    
  
**Consider the following HTML. Notice we are using Bootstrap classes for styling.**

<div class="form-group">

  <label for="email">Email</label>

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

          [(ngModel)]="employee.email">

</div>

The above HTML would produce the following Email input field   
   
  
We want to validate this email input field for 2 things 

* Email is required and
* Valid email must be provided

To make email, a required field modify the HTML as shown below

<div class="form-group" [class.has-error]="email.invalid && email.touched">

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

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

          [(ngModel)]="employee.email" #email="ngModel">

  <span class="help-block" \*ngIf="email.invalid && email.touched">

    Email is required

  </span>

</div>

**Code Explanation :** 

* [class.has-error]="email.invalid && email.touched". This is class binding in angular. If the email field is touched and invalid, then the Bootstrap class has-error is added to the div element, else the class is removed.
* On the label that displays "Email" text, we applied control-label Bootstrap class. This class turns the label text to red if there is a validation error.
* \*ngIf="email.invalid && email.touched". Notice the \*ngIf structural directive on the span element. If the email field is touched and invalid the span element is added to the DOM, else it is removed. The Bootstrap help-block class on the span element is for styling.

At this point, if you touch the email field and leave it without typing in anything, you will see the validation error message "Email is required"

   
  
We also want to make sure the user enters a valid email. If someone types ABC, that is not a valid email. Angular 4 has built-in email validator, that we can use to validate if the user has entered a valid email. Here are the steps.  
  
**Step 1 :**On the email input field, place the **email** directive 

<input id="email" required email

**Step 2 :**Use the following HTML, to display the validation error message. If the email is invalid, angular attaches email key to the errors collection. On the other hand, if the email field is valid, the key email will not be in the errors collection. The question mark here is called the safe navigation operator. It protects against null and undefined values in property paths. It is generally used when we are not sure if a property exists or not. It safely handles null and undefined values, and very useful to prevent null-reference exceptions.

<span class="help-block" \*ngIf="email.errors?.email && email.touched">

  Email is Invalid

</span>

Here is the complete HTML that makes the email filed required and also checks if the email has a valid format

<div class="form-group" [class.has-error]="email.invalid && email.touched">

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

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

          [(ngModel)]="employee.email" #email="ngModel">

  <span class="help-block" \*ngIf="email.errors?.required && email.touched">

    Email is required

  </span>

  <span class="help-block" \*ngIf="email.errors?.email && email.touched">

    Email is Invalid

  </span>

</div>

As of this recording, email validator provided by Angular does not allow null or empty values. When we leave the email field empty, the email validator is still fired. This is wrong. Checking NULL and empty values should be the job of the required validator. The following is the work around.  
  
Bind email directive to a boolean expression. The email validator is only added when the email field value is not an empty string. This ensures that, when we type something in the email field, the email validator is attached to the input field and it validates if the email format is valid or not.   
  
<input id="email" required [email]="employee.email!==''"  
  
**Please note :**Do not forget to initialise the email property in the employee object to an empty string.s

### #Lecture19. Angular regular expression validation

In this video we will discuss using **pattern validator in angular** to meet most of your application complex validation requirements.    
  
  
**With the pattern validator we use a regular expression**. Regular expressions are extremely useful when you want to validate if a given string conforms to a specified pattern.   
  
For example, you can use regular expressions to check if a given email conforms to a a valid email format. Similarly you can also check if provided postcode conforms to a specific country postcode format.   
  
Apart from checking conformity with a pattern, they can also be used to extract sub-strings from a given input string.  
  
  
**To validate** if the provided email has a valid email pattern we can use the pattern validator in angular. To use the pattern validator use the pattern attribute along with the regular expression on the input field you want to validate.

<input pattern="^[a-zA-Z0-9\_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$" type="text"

       name="email" [(ngModel)]="employee.email" #email="ngModel">

It is easy to learn regular expressions. Initially they may appear complicated, but if you get the basics right it is very easy to understand them. However, you can also find the commonly used regular expressions on the internet. For example, if you want to find a regular expression to validate email address, simply search the internet with the following string  
Regular expression for email validation  
  
**Use the following HTML**, to display the validation error message. If the email is invalid, angular attaches pattern key to the errors collection. On the other hand, if the email field is valid, the key pattern will not be in the errors collection. The question mark here is called the **safe navigation operator**. We discussed this operator in detail in our previous video. If you are new to this operator, please check out our previous video.

<span class="help-block" \*ngIf="email.errors?.pattern && email.touched">

  Email is Invalid

</span>

The following example, shows both **required**and **pattern**validators on the Email input field.

<div class="form-group" [class.has-error]="email.invalid && email.touched">

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

  <input required pattern="^[a-zA-Z0-9\_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$"

         id="email" type="text" class="form-control" name="email"

         [(ngModel)]="employee.email" #email="ngModel">

  <span class="help-block" \*ngIf="email.errors?.required && email.touched">

    Email is required

  </span>

  <span class="help-block" \*ngIf="email.errors?.pattern && email.touched">

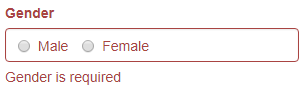
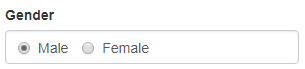
    Email is Invalid

  </span>

</div>

### Let's take this pattern validation to the next level. I want to validate emails against a specific domain. For example pragimtech.com is the only valid domain that I want to allow. Any other domain should be considered invalid. This can be very easily achieved with the following regular expression. ^[a-zA-Z0-9\_.+-]+@(?:(?:[a-zA-Z0-9-]+\.)?[a-zA-Z]+\.)?(pragimtech)\.com$

### Lecture 20->Angular radio button validation

**Example :** Gender is a required field. If one of the gender radio button is not checked, we want to validate and display "Gender is required" validation error message.   
  
   
  
  
As soon as one of the "Gender" radio button is selected, the validation error message should disappear.   
  
   
  
Here is the HTML that makes this possible. 

<div class="form-group" [class.has-error]="gender.invalid">

  <label class="control-label">Gender</label>

  <div class="form-control">

    <label class="radio-inline">

      <input type="radio" name="gender" required #gender="ngModel"

             value="male" [(ngModel)]="employee.gender"> Male

    </label>

    <label class="radio-inline">

      <input type="radio" name="gender" required #gender="ngModel"

             value="female" [(ngModel)]="employee.gender"> Female

    </label>

  </div>

  <span class="help-block" \*ngIf="gender.invalid">

    Gender is required

  </span>

</div>

**Code Explanation :** 

* Notice we have required attribute on both the radio buttons (Male and Female). This attribute makes the "Gender" field required.
* #gender="ngModel". This creates a template reference variable. We can now this variable (gender) to check if the field is invalid. Notice #gender="ngModel" is placed on the both the radio buttons.
* [class.has-error]="gender.invalid". This class binding adds the has-errorbootstrap css class when the field is invalid and removes it when the field is valid. This class is used for styling the validation error messages.
* On the label element that displays the static text "Gender" we have "control-label" class. This class turns the text "Gender" to red when there is a validation error.
* \*ngIf="gender.invalid". Notice the \*ngIf structural directive on the span element. If the gender field is invalid the span element is added to the DOM, else it is removed. The Bootstrap help-block class on the span element is for styling.

The span element that displays the validation error message can also be coded as shown below. Notice, instead of using "gender.invalid" as the expression for \*ngIf, we are using "gender.errors?.required". When the required validation fails, Angular attaches the required key to the errors collection property of the gender field. The key is removed if the field passes validation. So we can check for the existence of this key, to control the display of the validation error message.

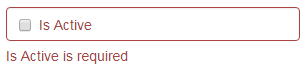
<span class="help-block" \*ngIf="gender.errors?.required">

  Gender is required

</span>

### Lecture 21-> Add required attribute dynamically in angular

# Lecture 22->Angular checkbox validation

In this video we will discuss **checkbox validation in Angular**with example.    
  
  
We want to make the following **"Is Active" check box a required field**. If the checkbox is not checked, we want to validate and display "Is Active is required" validation error message. As soon as the checkbox is checked, the validation error message should disappear.   
  
   
  
  
**Consider the following HTML**

<div class="form-group" [class.has-error]="isActive.invalid && isActive.touched">

  <div class="form-control">

    <label class="checkbox-inline control-label">

      <input type="checkbox" required name="isActive"

             #isActive="ngModel" [(ngModel)]="employee.isActive">

      Is Active

    </label>

  </div>

  <span class="help-block"

        \*ngIf="isActive.errors?.required && isActive.touched">

    Is Active is required

  </span>

</div>

**Code Explanation :** 

* The required attribute makes "Is Active" field required.
* #isActive="ngModel". This creates a template reference variable. We can now this variable (isActive) to check if the field is invalid, touched, dirty etc.
* [class.has-error]="isActive.invalid && isActive.touched". This class binding adds the has-error bootstrap css class when the field is invalid and touched and removes it when the field is valid. This class is used for styling the validation error messages.
* On the label element that displays the static text "Is Active" we have "control-label" class. This class turns the text "Is Active" to red when there is a validation error.
* \*ngIf="isActive.errors?.required && isActive.touched". Notice the \*ngIf structural directive on the span element. If the "Is Active" field fails required validation and touched, the span element is added to the DOM, else it is removed. The Bootstrap help-block class on the span element is for styling.

**At this point,**

* If you tab into the checkbox control and leave it, without checking it, you will see the validation error message
* If you select the checkbox box, the error goes away
* If you unselecet the checkbox, the required validation error appears again

This implementation of the checkbox validation is useful, when you want to force the user to select a checkbox. For example, on many web sites, you might have seen a checkbox with the following text. Only when you agree by checking that checkbox, you will be able to proceed. Otherwise you will have to cancel that specific action.

**I Agree to the terms and conditions**  
  
**What if the employee is terminated or resigned?** In that case we do not want the checkbox to be checked. But at the moment, the required validator is forcing us to have the checkbox checked. To fix this modify the required attribute as shown below. Notice, we are binding a boolean expression to the required attribute. If the expression is true the required validator is attached, otherwise it is removed.  
  
[required]="employee.isActive==null"   
  
**With this change** 

* When the form first loads, isActive property on the employee object is null. So the required attribute is attached to the checkbox.
* If we tab into the checkbox and levae it without selecting it, we see the required validation error message as expected
* If we select the checkbox box, the error goes away
* If we unselecet the checkbox, notice we don't get the required validation. This is because, when the checkbox is unchecked, the value of isActive property on the employee object is false and not NULL. So the boolean expression bound to the required attribute returns false. Hence the required attribute is removed from the checkbox field and we do not see the required validation error.

At the moment, the user interface is confusing. If the employee you are creating is not active, you have to first check the (Is Active) checkbox and then un-check it. To make this less confusing there 2 options for us.

* Remove the required validator on the (Is Active) checkbox, and treat NULL as false.
* Use 2 radio buttons (Yes or No), instead of a single checkbox.
* So by fixxing this issue I will remove all validation call.

<div class="form-group">

<div class="form-control">

<label class="checkbox-inline">

<input type="checkbox" [(ngModel)]="employee.isActive" name="isActive"> IsActive

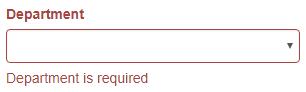
</label>

</div>

</div>

### 

# Lecture 23->Angular select list validation

In this video, we will discuss **Dropdown list validation in Angular** with example.    
  
  
**Example :** We want to make "Department" Dropdownlist a required field. If a department is not selected, we want to validate and display "Department is required"validation error message. As soon as a department is checked, the validation error message should disappear.   
  
   
  
  
**Consider the following HTML :** 

<div class="form-group"

     [class.has-error]="department.touched && department.invalid">

  <label for="department" class="control-label">Department</label>

  <select id="department" required #department="ngModel"

          name="department" [(ngModel)]="employee.department"

          class="form-control">

    <option \*ngFor="let dept of departments" [value]="dept.id">

      {{dept.name}}

    </option>

  </select>

  <span class="help-block"

        \*ngIf="department.touched && department.invalid">

    Department is required

  </span>

</div>

**Code Explanation :** 

* The required attribute makes "Department" field required.
* #department="ngModel". This creates a template reference variable. We can now this variable (department) to check if the field is invalid, touched, dirty etc.
* [class.has-error]="department.touched && department.invalid". This class binding adds the has-error bootstrap css class when the field is invalid and touched and removes it when the field is valid. This class is used for styling the validation error messages.
* On the label element that displays the static text "Department" we have "control-label" class. This class turns the text "Department" to red when there is a validation error.
* \*ngIf="department.touched && department.invalid". Notice the \*ngIf structural directive on the span element. If the "Department" field fails required validation and touched, the span element is added to the DOM, else it is removed. The Bootstrap help-block class on the span element is for styling.

At this point, the dropdown list validation works as expected. However, in most of the real world applications, you might see one of the following options as the first option in a dropdown list.

* Please select
* Select Department
* etc...

Modify the HTML to include "Select Department" as the first option. Notice the value of this option is set to**'-1'**, to indicate that it is not a valid department selection. The change is highlighted in YELLOW.

<div class="form-group"

     [class.has-error]="department.touched && department.invalid">

  <label for="department" class="control-label">

    Department

  </label>

  <select id="department" required #department="ngModel" name="department"

          [(ngModel)]="employee.department" class="form-control">

    <option value="-1">Select Department</option>

    <option \*ngFor="let dept of departments" [value]="dept.id">

      {{dept.name}}

    </option>

  </select>

  <span class="help-block"

        \*ngIf="department.touched && department.invalid">

    Department is required

  </span>

</div>

### In the component class (create-employee.component.ts), initialise department property with a value of '-1'. This will ensure that, when the "Department" dropdownlist is loaded, the first default option 'Select Department' is selected. employee: Employee = {   id: null,   name: null,   gender: null,   contactPreference: null,   phoneNumber: null,   email: '',   dateOfBirth: null,   department: '-1',   isActive: null,   photoPath: null }; At this point, view the page in the browser. The dropdownlist REQUIRED validation does not work as expected. The default first option, 'Select Department' is treated as a valid department selection. We will discuss, how to fix this in our next video.

# #Lecture 24. Angular value vs ngvalue

<div class="form-group" [class.has-error]="department.invalid && department.touched">

<label for="department" class="control-lable">Department</label>

<select id="department" required #department="ngModel" name="department" class="form-control" [(ngModel)]="employee.department"

name="department">

<option [ngValue]="null" disabled>Select Department</option>

<option \*ngFor="let dept of departments" [ngValue]="dept">

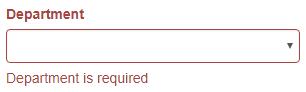
{{dept.name}}

</option>

</select>

<span \*ngIf="department.invalid && department.touched">Department is Required</span>

</div>

In this video, we will discuss **Dropdown list validation in Angular** with example.    
  
  
**Example :** We want to make "Department" Dropdownlist a required field. If a department is not selected, we want to validate and display "Department is required"validation error message. As soon as a department is checked, the validation error message should disappear.   
  
   
  
  
**Consider the following HTML :** 

<div class="form-group"

     [class.has-error]="department.touched && department.invalid">

  <label for="department" class="control-label">Department</label>

  <select id="department" required #department="ngModel"

          name="department" [(ngModel)]="employee.department"

          class="form-control">

    <option \*ngFor="let dept of departments" [value]="dept.id">

      {{dept.name}}

    </option>

  </select>

  <span class="help-block"

        \*ngIf="department.touched && department.invalid">

    Department is required

  </span>

</div>

**Code Explanation :** 

* The required attribute makes "Department" field required.
* #department="ngModel". This creates a template reference variable. We can now this variable (department) to check if the field is invalid, touched, dirty etc.
* [class.has-error]="department.touched && department.invalid". This class binding adds the has-error bootstrap css class when the field is invalid and touched and removes it when the field is valid. This class is used for styling the validation error messages.
* On the label element that displays the static text "Department" we have "control-label" class. This class turns the text "Department" to red when there is a validation error.
* \*ngIf="department.touched && department.invalid". Notice the \*ngIf structural directive on the span element. If the "Department" field fails required validation and touched, the span element is added to the DOM, else it is removed. The Bootstrap help-block class on the span element is for styling.

At this point, the dropdown list validation works as expected. However, in most of the real world applications, you might see one of the following options as the first option in a dropdown list.

* Please select
* Select Department
* etc...

Modify the HTML to include "Select Department" as the first option. Notice the value of this option is set to**'-1'**, to indicate that it is not a valid department selection. The change is highlighted in YELLOW.

<div class="form-group"

     [class.has-error]="department.touched && department.invalid">

  <label for="department" class="control-label">

    Department

  </label>

  <select id="department" required #department="ngModel" name="department"

          [(ngModel)]="employee.department" class="form-control">

    <option value="-1">Select Department</option>

    <option \*ngFor="let dept of departments" [value]="dept.id">

      {{dept.name}}

    </option>

  </select>

  <span class="help-block"

        \*ngIf="department.touched && department.invalid">

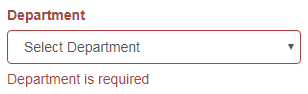
    Department is required

  </span>

</div>

In the component class (create-employee.component.ts), initialise department property with a value of **'-1'**. This will ensure that, when the "Department" dropdownlist is loaded, the first default option 'Select Department' is selected.  
  
employee: Employee = {  
  id: null,  
  name: null,  
  gender: null,  
  contactPreference: null,  
  phoneNumber: null,  
  email: '',  
  dateOfBirth: null,  
  department: '-1',  
  isActive: null,  
  photoPath: null  
};  
  
At this point, view the page in the browser. The dropdownlist REQUIRED validation does not work as expected. The default first option, 'Select Department' is treated as a valid department selection. We will discuss, how to fix this in our next video.

# #25. Angular custom validator example template driven forms

In this video, we will discuss creating a **custom validator in angular with an example**. We will make this validator reusable and configurable. Along the way, we will also learn how to create a **custom directive in angular**.   
  
  
**Example :**When you have a default option like **"Select Department"** in a SELECT list, the REQUIRED validation does not work, if the default option value is anything else other than NULL. So if the default option value is not null, then that default option is also treated as a valid selection, and we do not get to see the required validation error. If "Select Department" is selected, we want the validation to fail, and display "Department is required" validation error message. To make this work the way we want, we implement a custom validator.    
  
   
  
  
**Create a custom Directive**  
To use a custom validator in template driven forms, we have to create the validator as a directive. Once we have the directive created, we can then use that directive as an attribute on the select element that we want to validate. This is going to be a configurable and reusable validator. We can use it with any SELECT list in an angular application. So create a **"shared"** folder in the **"app"** folder. In the "shared" folder create a file with name "select-required-validator.directive.ts". Copy and paste the folllowing code. 

import { Validator, AbstractControl, NG\_VALIDATORS } from '@angular/forms';

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

@Directive({

    selector: '[appSelectValidator]',

    providers: [

        {

            provide: NG\_VALIDATORS,

            useExisting: SelectRequiredValidatorDirective,

            multi: true

        }]

})

export class SelectRequiredValidatorDirective implements Validator {

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

        return control.value === '-1' ? { 'defaultSelected': true } : null;

    }

}

**Code Exaplanation :**  
Since we are creating a directive, we decorate the class with **@Directive**decorator   
  
**NG\_VALIDATORS** is a collection of validators. It already contains all the built-in validators like required, pattern etc. Before we can use our custom validator we have to add it to the list of validators by adding it to NG\_VALIDATORS token. To specify that we want to add our validator to the list of validators, we set **multi** property to true   
  
providers: [  
    {  
        provide: NG\_VALIDATORS,  
        useExisting: SelectRequiredValidatorDirective,  
        multi: true  
    }]   
  
Implement **Validator** interface as we are creating a custom validator  
export class SelectRequiredValidatorDirective implements Validator  
  
Since we are implementing validator interface, we have to provide implementation for the interface **validate()** method. This method has one input parameter and it's type is **AbstractControl**. AbstractControl extends both **FormControl** and **FormGroup**. In some cases you may want to validate a Formgroup instead of a single FormControl. So to cater for both scenarios, the parent type - AbstractControl is specified. This function returns an object if the validation fails or null if the validation succeeds. The object that is returned when the validation fails contains a key/value pair. The key is a string and the value can be anything.  
validate(control: AbstractControl): { [key: string]: any } | null   
  
If the selected value in the SELECT list is the default value (-1), then we return an object with key 'defaultSelected' and value of true to indicate that the validation has failed. Otherwise we return NULL to indicate validation succeeded. In the HTML we can use the "defaultSelected" key to display the validation error specific to this custom validator.  
return control.value === '-1' ? { 'defaultSelected': true } : null;  
  
**Import the custom directive in a module where you want to use it.**  
  
At the moment we only have one module - Root module. So in app.module.ts file include the following import statement  
import { SelectRequiredValidatorDirective } from './shared/select-required-validator.directive';  
  
Also include SelectRequiredValidatorDirective in the declarations array of the NgModule() decorator  
  
**Using the custom required validator on the SELECT element**  
  
Modify the "Department" SELECT element in create-employee.component.html file as shown below.

<div class="form-group"

     [class.has-error]="department.touched && department.errors?.defaultSelected">

  <label for="department" class="control-label">Department</label>

  <select id="department" #department="ngModel" name="department"

          [(ngModel)]="employee.department" appSelectValidator

          class="form-control">

    <option value="-1">Select Department</option>

    <option \*ngFor="let dept of departments" [value]="dept.id">

      {{dept.name}}

    </option>

  </select>

  <span class="help-block"

        \*ngIf="department.touched && department.errors?.defaultSelected">

    Department is required

  </span>

</div>

**Code Explanation :** 

* [class.has-error]="department.touched && department.errors?.defaultSelected". Notice, in this conditional class binding we are using the key "defaultSelected" to style the Department SELECT element and it's label when there is a validation error. This key is set by the required custom validator when the validation fails.
* \*ngIf="department.touched && department.errors?.defaultSelected". Notice here also we are using the key "defaultSelected" to display the validation error message.
* appSelectValidator is the selector we gave for our custom validation directive and we are using it as an attribute on the SELECT list that we want to validate.

At the moment, we can only use this custom validator with a SELECT list whose default option value is -1. We will discuss how to make this custom validator configurable and reusable in our next video.

#Lecture->26

In this video we will discuss, **how to make the select list custom required validator configurable and reusable**. This is continuation to [Part 25](https://www.youtube.com/watch?v=2AAUf32pKy8). Please watch [Part 25](https://www.youtube.com/watch?v=2AAUf32pKy8) from [Angular CRUD tutorial](https://www.youtube.com/playlist?list=PL6n9fhu94yhXwcl3a6rIfAI7QmGYIkfK5) before proceeding.   
  
  
Here is the SELECT list **custom required validator**we implemented in our previous video 

import { Validator, AbstractControl, NG\_VALIDATORS } from '@angular/forms';

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

@Directive({

    selector: '[appSelectValidator]',

    providers: [{

        provide: NG\_VALIDATORS,

        useExisting: SelectRequiredValidatorDirective,

        multi: true

    }]

})

export class SelectRequiredValidatorDirective implements Validator {

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

        return control.value === '-1' ? { 'defaultSelected': true } : null;

    }

}

**Consider this line of code :**Notice we have hard-coded the default option value '-1'. Because of this hard-coded value, we will not be able to reuse this validator with another SELECT list if it has a different default option value other than '-1'.    
  
return control.value === '-1' ? { 'defaultSelected': true } : null;  
  
To make this custom validator reusable, we want to be able to do pass the default option value from the template to our custom validator as shown below. Notice we are using our custom validator selector and passing it the default option value. In this case we are passing -101. If you have another SELECT list, and if it's default option value is -1, you simply pass that value.  
<select appSelectValidator="-101" #department="ngModel" ....>  
  
For this to work, we have to create a corresponding input property in the custom validator class. Modify SelectRequiredValidatorDirective as shown below. The changes are commented and self-explanatory

import { Validator, AbstractControl, NG\_VALIDATORS } from '@angular/forms';

// Import input from @angular/core package

import { Directive, Input } from '@angular/core';

@Directive({

    selector: '[appSelectValidator]',

    providers: [{

        provide: NG\_VALIDATORS,

        useExisting: SelectRequiredValidatorDirective,

        multi: true

    }]

})

export class SelectRequiredValidatorDirective implements Validator {

    // Create input property to receive the

    // specified default option value

    @Input() appSelectValidator: string;

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

        // Remove the hard-coded value and use the input property instead

        return control.value === this.appSelectValidator ?

                                    { 'defaultSelected': true } : null;

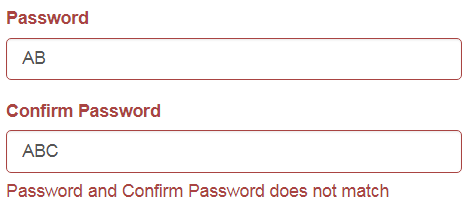
    }

}

**Please note :**Since this is a directive input property, the input property name and the selector name must match.   
@Input() appSelectValidator: string;  
  
For some reason if you do not like the input property name, you can use an alias as shown below.   
@Input('appSelectValidator') defaultValue: string;  
  
We are now able to specify the default option value using the directive input property. This makes our custom validator configurable and reusable. We can now use this custom required validator to validate any SELECT list in our Angular application.

### Lecture 27->Angular password and confirm password validation

### Angular password and confirm password validation

**Suggested Videos**  
[Part 24 - Angular value vs ngValue](https://www.youtube.com/watch?v=dyif1Xy9GY8) | [Text](http://csharp-video-tutorials.blogspot.com/2018/02/angular-value-vs-ngvalue.html) | [Slides](http://csharp-video-tutorials.blogspot.com/2018/02/angular-value-vs-ngvalue-slides.html)  
[Part 25 - Angular custom validator example template driven forms](https://www.youtube.com/watch?v=2AAUf32pKy8) | [Text](http://csharp-video-tutorials.blogspot.com/2018/02/angular-custom-validator-example_27.html) | [Slides](http://csharp-video-tutorials.blogspot.com/2018/02/angular-custom-validator-example_81.html)   
[Part 26 - Angular select list required custom validator](https://www.youtube.com/watch?v=BjsaaUNw4lk) | [Text](http://csharp-video-tutorials.blogspot.com/2018/02/angular-select-list-required-custom.html) | [Slides](http://csharp-video-tutorials.blogspot.com/2018/02/angular-select-list-required-custom_28.html)  
  
In this video we will discuss how to **compare password and confirm password fields**and validate if they are equal. If they are not equal we want to display "Password and Confirm Password does not match" validation error.   
  
   
  
  
This is also commonly called as **cross field validation in angular**. We cannot use any of the buil-in angular validators to perform cross-field validation. So let's create a custom validator. To use a custom validator in template driven forms, we create the validator as a directive. We discussed creating custom validators and directives in Parts [25](https://www.youtube.com/watch?v=2AAUf32pKy8&list=PL6n9fhu94yhXwcl3a6rIfAI7QmGYIkfK5&index=25&t=0s) and [26](https://www.youtube.com/watch?v=BjsaaUNw4lk&list=PL6n9fhu94yhXwcl3a6rIfAI7QmGYIkfK5&index=26&t=0s) of [Angular CRUD tutorial](https://www.youtube.com/playlist?list=PL6n9fhu94yhXwcl3a6rIfAI7QmGYIkfK5). If you are new to these concepts, please check out those videos.   
  
We will make this custom validator a reusable validator, so we could use it to compare any 2 input fields for equality. For example, we can use this same custom validator to compare if EMAIL and CONFIRM EMAIL fields are equal.   
  
  
**Create a custom Directive**  
Add a new TypeScript file to the "shared" folder. Name it confirm-equal-validator.directive.ts.Copy and paste the folllowing code.

import { Validator, AbstractControl, NG\_VALIDATORS } from '@angular/forms';

import { Directive, Input } from '@angular/core';

@Directive({

    selector: '[appConfirmEqualValidator]',

    providers: [{

        provide: NG\_VALIDATORS,

        useExisting: ConfirmEqualValidatorDirective,

        multi: true

    }]

})

export class ConfirmEqualValidatorDirective implements Validator {

    @Input() appConfirmEqualValidator: string;

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

        const controlToCompare = control.parent.get(this.appConfirmEqualValidator);

        if (controlToCompare && controlToCompare.value !== control.value) {

            return { 'notEqual': true };

        }

        return null;

    }

}

**Code Exaplanation :**  
Since we are creating a directive, we decorate the class with @Directive decorator  
  
This selector will be used as a directive on one of the 2 input fields that we want to compare for equality. In our case we will use it on the **Confirm Password** field.  
selector: '[appConfirmEqualValidator]',  
  
**NG\_VALIDATORS** is a collection of validators. It contains all the built-in validators like required, pattern etc. Before we can use our custom validator we have to add it to the list of validators by adding it to NG\_VALIDATORS token. To specify that we want to add our validator to the list of validators, we set multi property to true  
  
providers: [{  
    provide: NG\_VALIDATORS,  
    useExisting: ConfirmEqualValidatorDirective,  
    multi: true  
}]  
  
Implement Validator interface as we are creating a custom validator  
export class ConfirmEqualValidatorDirective implements Validator  
  
Since our custom validator class is implementing validator interface, we have to provide implementation for the interface validate() method. This method has one input parameter and it's type is AbstractControl. AbstractControl extends both FormControl and FormGroup. In some case you may want to validate a Formgroup instead of a single FormControl. So to cater for both scenarios, the parent type - AbstractControl is specified. This function returns an object if the validation fails or null if the validation succeeds. The object that is returned when the validation fails contains a key/value pair. The key is a string and the value can be anything.  
validate(control: AbstractControl): { [key: string]: any } | null  
  
The following line creates an input property. Since this is a directive input property, the input property name and the selector name must match.   
@Input() appConfirmEqualValidator: string;  
  
We will use this custom directive (appConfirmEqualValidator), as an attribute either on the PASSWORD field or CONFIRM PASSWORD FIELD. If we use this on the  CONFIRM PASSWORD field, we will also pass the field that we want to compare with. In this case the PASSWORD field.   
  
So the input property that we have created above receives the control that we want to compare CONFIRM PASSWORD field with. This input property prevents the need to hard code the name of the control that we want to compare with. Hence it makes our custom validator reusable. We can use it to compare any 2 input fields for equality.  
<input name="confirmPassword" appConfirmEqualValidator="password"

* In the validate() method implementation, we first retrieve the control that we want to compare CONFIRM PASSWORD field with. That field in our case is the PASSWORD field.
* Both PASSWORD and CONFIRM PASSWORD fields are siblings. So to get the PASSWORD field, we go one level up from the CONFIRM PASSWORD field using the parent property. The parent property returns the root FormGroup.
* On the root FormGroup we call the get() method passing it, the input property. The input property receives the name of the PASSWORD field
* Finally we check if the PASSWORD and CONFIRM PASSWORD filed values are equal. If they are equal, we return NULL indication validation succeeded otherwise we return an object with key=notEqual and value=true.
* In the HTML we can use this key (notEqual) to display the relevant validation error message

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

    const controlToCompare = control.parent.get(this.appConfirmEqualValidator);

    if (controlToCompare && controlToCompare.value !== control.value) {

        return { 'notEqual': true };

    }

    return null;

}

**Import the custom directive in a module where you want to use it.**  
  
At the moment we only have one module - Root module. So in app.module.ts file include the following import statement  
import { ConfirmEqualValidatorDirective } from './shared/confirm-equal-validator.directive';  
  
Also include ConfirmEqualValidatorDirective in the declarations array of the NgModule() decorator  
  
**Using the custom validator**   
  
Include the following HTML for Password and Confirm Password fields in create-employee.component.html file as shown below.

<div class="form-group"

     [class.has-error]="password.touched && password.invalid">

  <label for="password" class="control-label">Password</label>

  <input id="password" required type="text" class="form-control"

         name="password" [(ngModel)]="employee.password"

         #password="ngModel">

  <span class="help-block"

        \*ngIf="password.touched && password.errors?.required">

    Password is required

  </span>

</div>

<div class="form-group"

     [class.has-error]="confirmPassword.touched && confirmPassword.invalid">

  <label for="confirmPassword" class="control-label">Confirm Password</label>

  <input name="confirmPassword" appConfirmEqualValidator="password" required

         id="confirmPassword" type="text" class="form-control"

         [(ngModel)]="employee.confirmPassword" #confirmPassword="ngModel">

  <span class="help-block"

        \*ngIf="confirmPassword.touched && confirmPassword.errors?.required">

    Confirm Password is required

  </span>

  <span class="help-block"

        \*ngIf="confirmPassword.touched && confirmPassword.errors?.notEqual &&

          !confirmPassword.errors?.required">

    Password and Confirm Password does not match

  </span>

</div>

Notice on the CONFIRM PASSWORD field we are using our custom directive and passing it the name of the control (PASSWORD) that we want to compare with.  
<input name="confirmPassword" appConfirmEqualValidator="password"  
  
Notice the expression of \*ngIf structural directive. We are using the key (notEqual) set by our custom validator to display the validation error message.

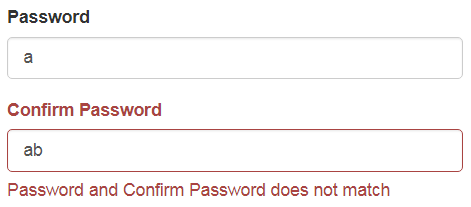
<span class="help-block"

      \*ngIf="confirmPassword.touched && confirmPassword.errors?.notEqual &&

        !confirmPassword.errors?.required">

  Password and Confirm Password does not match

</span>

**At the moment there are 2 problems with our custom validator**  
When the validation fails only the CONFIRM PASSWORD field is styled with red border and not the PASSWORD field. We want the password field also to have the red border.   
  
   
  
If you first change PASSWORD field and then the CONFIRM PASSWORD field, the validation works as expected. Now if you go back and change the PASSWORD field, the validation will not be triggered and you will not see the validation error even if the passwords do not match.   
  
**We will discuss why this is happening and how to fix it in our next video.**

### 

### Lecture 27-> Angular trigger validation manually

**Suggested Videos**  
[Part 25 - Angular custom validator example template driven forms](https://www.youtube.com/watch?v=2AAUf32pKy8) | [Text](http://csharp-video-tutorials.blogspot.com/2018/02/angular-custom-validator-example_27.html) | [Slides](http://csharp-video-tutorials.blogspot.com/2018/02/angular-custom-validator-example_81.html)  
[Part 26 - Angular select list required custom validator](https://www.youtube.com/watch?v=BjsaaUNw4lk) | [Text](http://csharp-video-tutorials.blogspot.com/2018/02/angular-select-list-required-custom.html) | [Slides](http://csharp-video-tutorials.blogspot.com/2018/02/angular-select-list-required-custom_28.html)   
[Part 27 - Angular password and confirm password validation](https://www.youtube.com/watch?v=YhazkQd59Hk) | [Text](http://csharp-video-tutorials.blogspot.com/2018/03/angular-password-and-confirm-password.html) | [Slides](http://csharp-video-tutorials.blogspot.com/2018/03/angular-password-and-confirm-password_6.html)  
  
**In this video we will discuss**

1. How to add or remove validation styles to a group of elements in Angular
2. How to trigger validation manually in Angular using the updateValueAndValidity() function

This is continuation to [Part 27](https://www.youtube.com/watch?v=YhazkQd59Hk). Please watch [Part 27](https://www.youtube.com/watch?v=YhazkQd59Hk) from [Angular CRUD tutorial](https://www.youtube.com/playlist?list=PL6n9fhu94yhXwcl3a6rIfAI7QmGYIkfK5) before proceeding.   
  
**How to add and remove validation styles to a group of elements in Angular :** Use the **ngModelGroup**directive and group the elements. Now we can add or remove validation styles from the group. This in turn adds or removes the validation styles from all the elements in that group.   
  
  
In our case, we want to **group password and confirm password fields** to be able to control their validation styles. Notice in the example below, both password and confirm password fields are grouped using the **ngModelGroup**directive. The bootstrap validation class **has-error** is conditionally added or removed from the group.

<div ngModelGroup="passwordGroup"

     [class.has-error]="confirmPassword.touched && confirmPassword.invalid">

  <div "passwordFieldDiv"> ...

  </div>

  <div "confirmPasswordFieldDiv"> ...

  </div>

</div>

**Use of updateValueAndValidity() function :** At the moment we have a problem with confirm password field validation. It does not work in one of the scenarios. Here is the scenario.   
  
If you first change PASSWORD field and then the CONFIRM PASSWORD field, the validation works as expected. Now if you go back and change the PASSWORD field, the validation will not be triggered and you will not see the validation error even if the passwords do not match.   
  
This is because our custom validation directive is applied on the confirm password filed but not on the password. So our custom validation is triggered only when the confirm password field is changed and not when the password field is changed. To make this work, even when the password field is changed, we have to tell confirm password field to run it's validation when password field is changed.   
  
So the obvious question that comes to our mind is, **how to tell the confirm password field to run it's validation?**  
Well updateValueAndValidity() function comes to the rescue. When this method is called on a control, that control's validation logic is executed again. Notice the event binding in the example below. The **change**event of the password field triggers a call to confirm password field's updateValueAndValidity() function. This in turn runs the confirm password field validation.

<input name="password"

      (change)="confirmPassword.control.updateValueAndValidity()" …>

The **change**event is fired only after the form control has lost focus. The **input**event is fired as the user changes the value. So if you want the validation to trigger as the user is changing the value, use the **input**event instead of **change**event.

### Lecture 27->Angular password and confirm password validation