rxweb/reactive-form-validators 使用指南

<https://docs.rxweb.io/getting-started>

**显示错误消息的最佳方式**

<https://docs.rxweb.io/how-to/errorMessages/validators>

因为我们主要是在响应式表单上工作，当多个验证器涉及到一个表单控件时，显示错误消息是一项有点笨拙的任务。通常我们在html中放入\*ngIf，并绑定相应的验证器的错误消息。为了克服这个问题，让我们讨论显示错误消息的最合适的方法。

在讨论代码之前，让我们先看看场景:  
我们有用户名和密码字段。用户名字段是必需的字段和密码字段有多个验证，如最小长度应该是5，最大长度应该是10和特殊字符真和数字真。在此基础上，有两种场景，一个验证显示一条错误消息，多个验证显示多条消息。

要在应用程序中全局配置消息，请参考ReactiveFormConfig

**显示一个消息**

<form [formGroup]="userFormGroup">

<div class="form-group">

<label>UserName</label>

<input type="text" formControlName="userName" class="form-control" />

<small class="form-text text-danger">**{{userFormGroup.controls.userName["errorMessage"]}}**<br/></small>

</div>

</form>

**显示多个消息**

<form [formGroup]="userFormGroup">

<div class="form-group">

<label>Password</label>

<input type="text" formControlName="password" class="form-control" />

<small class="form-text text-danger"

**\*ngFor="let errorMessage of userFormGroup.controls.password['errorMessages']"**>

{{errorMessage}}<br/>

</small>

</div>

</form>

**在HTML中使用不带引号的errorMessage**

<form [formGroup]="userFormGroup">

<div class="form-group">

<label>UserName</label>

<input type="text" formControlName="userName" class="form-control" />

<small class="form-text text-danger">**{{userFormGroup.controls.userName.errorMessage}}**<br/></small>

</div>

<button [disabled]="!userFormGroup.valid" class="btn btn-primary">Submit</button>

</form>

**Class Abstraction**

<https://docs.rxweb.io/how-to/classAbstraction/decorators>

在使用基于模型的响应式表单时，当你想要将基类的属性使用到派生类中并隐藏继承类实现的复杂性时，你必须使用类抽象。

假设你有一个Order类，它有OrderId和OrderType属性，你想要从Product类继承它，Product类有productId和productName，你想要在Products类中使用Orders的属性。

**Commit**

**<https://docs.rxweb.io/how-to/commit/validators>**

如果您在表单中使用脏检查特性，并且希望在将值更新到数据库后保持脏状态为false。为此，你必须使用各自formGroup的commit方法

让我们看看它是如何工作的。在这里，我们创建了三个表单控件，分别是Id、Name和名称。一旦显示数据更新，dirty状态应该变为false。

export class CommitDynamicValidatorComponent implements OnInit {

editForm: RxFormGroup

constructor(

private formBuilder: RxFormBuilder,private http: HttpClient ) { }

ngOnInit() {

this.editForm = <RxFormGroup>this.formBuilder.group({

id:[1], name: [''], designation: ['']

})

}

update(){

this.http.get('assets/dynamic.json').subscribe(dynamic => {

this.editForm.patchModelValue(dynamic);

this.editForm.commit();

})

}

}

Next, we need to write html code.

<div>

<form [formGroup]="editForm">

<div class="form-group">

<label>Id</label>

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

</div>

<button (click)="update()" [disabled]="!editForm.**valid**" class="btn btn-primary">Update</button>

</form>

</div>

**About Dirty Check**

<https://docs.rxweb.io/how-to/dirtyCheck/validators>

最适合在编辑表单中使用脏检查的情况。这是最常见的场景，我们需要根据提供的值进行检查，如果值被更改，那么表单就会变脏。

Angular在FormControl中提供了一个dirty字段。但是，如果用户再次根据初始化后的值更改该值，这将不起作用。但是通过isDirty函数，你可以达到同样的效果

**isModified**

isModified属性用于检查formControl的值是否被改变，而不检查整个状态。如果formControl的值被改变了，状态就会改变，否则不会。

<button [disabled]="!editForm.**isModified**" class="btn btn-primary">Is Modified</button>

**全局绑定错误消息策略**

<https://docs.rxweb.io/how-to/errorMessageStrategy/validators>

全局绑定错误消息策略

当我们想要在响应式表单中验证formControls时有条件地绑定错误消息时，我们会通过错误或模型装饰器绑定它，但是当出现在整个应用程序中都有相同的错误消息策略时，在每个模型上声明它就会变得非常笨重且难以管理。一个更好的方法是全局配置错误绑定策略

如果你的表单有嵌套的formGroup或formArray，那么你不需要为每个嵌套的formGroup和formArray设置属性为真，你只需要为根formGroup设置它为真。

在整个应用程序中配置errorMessageBindingStrategy。请参阅ReactiveFormConfig

如果你正在使用error或model decorator，它将覆盖特定formControl或formGroup的全局消息绑定策略

根据你的需要，你可以设置errorMessageBindingStrategy为以下策略:

errorMessageBindingStrategy.OnSubmit

当RxFormGroup属性提交为true时，无效的FormControl错误消息将被绑定。

errorMessageBindingStrategy.OnDirty

一旦FormControl被脏了，在那之后无效的FormControl错误消息将被相应地绑定。FormControl应该用markAsDirty()标记为dirty

errorMessageBindingStrategy.OnTouched

一旦FormControl被触摸，当我们相应地设置无效的FormControl错误消息后，FormControl应该被标记为触摸，使用markAsTouched()

errorMessageBindingStrategyOnDirtyOrTouched

一旦FormControl被Dirty或Touched，无效的FormControl错误消息将被相应地绑定。

errorMessageBindingStrategyOnDirtyOrSubmit

一旦FormControl被Dirty或Submitted，无效的FormControl错误消息将被相应地绑定。

errorMessageBindingStrategyOnTouchedOrSubmit

一旦FormControl被Dirty或Touched，无效的FormControl错误消息将被相应地绑定。

**Example**

为了在整个应用程序中配置errorMessageBindingStrategy，在本例中errorMessageBindingStrategy被设置为OnSubmit，这将相应地工作。

ReactiveFormConfig.set({

reactiveForm: {

errorMessageBindingStrategy: ErrorBindingStrategy.onSubmit

}

})

这是一个errorMessageBindingStrategyOnSubmit的例子，当提交的属性设置为true时，formControl的所有errorMessage都会在提交时被绑定。

@Component({

selector: 'app-errormessagestrategy-onsubmit-validator',

templateUrl: './errormessagestrategy-onsubmit.component.html'

})

export class ErrormessagestrategyOnsubmitValidatorComponent implements OnInit {

userFormGroup: RxFormGroup

constructor(

private formBuilder: RxFormBuilder , private fb : FormBuilder ) { }

ngOnInit() {

this.userFormGroup = <RxFormGroup>this.formBuilder.group({

userName:['',RxwebValidators.required()],

password:['',RxwebValidators.required()],

address:this.formBuilder.group({

city: ['',RxwebValidators.required()]

}),

skills:this.fb.array([

this.getSkillFormGroup()

])

});

}

OnSubmit()

{

this.userFormGroup.submitted = true;

}

addSkill(){

let skillsArray = <FormArray>this.userFormGroup.controls.skills;

skillsArray.push(this.getSkillFormGroup());

}

getSkillFormGroup(){

return this.formBuilder.group({

skillName:['',RxwebValidators.required()]

})

}

}

<div>

<form [formGroup]="userFormGroup">

<div class="form-group">

<label>User Name</label>

<input type="text" formControlName="userName" class="form-control" />

<small class="form-text text-danger"

\*ngIf="userFormGroup.controls.userName.errors">{{userFormGroup.controls.userName.errorMessage}}

<br/></small>

</div>

<div class="form-group">

<label>Password</label>

<input type="text" formControlName="password" class="form-control" />

<small class="form-text text-danger" \*ngIf="userFormGroup.controls.password.errors">

{{userFormGroup.controls.password.errorMessage}}<br/>

</small>

</div>

<div class="form-group" [formGroup]="userFormGroup.controls.address">

<label>City</label>

<input type="text" formControlName="city" class="form-control" />

<small class="form-text text-danger"

\*ngIf="userFormGroup.controls.address.controls.city.errors">

{{userFormGroup.controls.address.controls.city.errorMessage}}<br/></small>

</div>

<div class="form-group" [formGroup]="skillGroup"

\*ngFor="let skillGroup of userFormGroup.controls.skills.controls">

<label>SkillName</label>

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

<small class="form-text text-danger" \*ngIf="skillGroup.controls.skillName.errors">

{{skillGroup.controls.skillName.errorMessage}}<br/></small>

</div>

<button (click)="OnSubmit()" class="btn btn-primary">Submit</button>

</form>

</div>

**按需验证配置**

<https://docs.rxweb.io/how-to/bindConfig/validators>

在开发大型表单时，我们需要验证FormControl。为此，我们将编写大量自定义业务逻辑来管理复杂的场景。

使用它的好处是不必担心这个验证器何时调用，它将被调用您在dynamicConfig函数中使用的任何属性。框架会自动处理这些事情。

下面是一个场景示例，其中有两个字段:购买价格和转售价格，转售价格被验证，转售价格必须至少比购买价格多30%，这可以使用dynamicConfig实现。让我们看看:

export class BindCompleteValidatorComponent implements OnInit {

userInfoFormGroup: FormGroup

ngOnInit() {

this.userInfoFormGroup = this.formBuilder.group({

purchasePrice:[''],

resalePrice: ['', RxwebValidators.minNumber({

dynamicConfig: (x, y) => {

const minimumPrice = (x.purchasePrice \* 1) + (x.purchasePrice \* 30 / 100);

return { value: minimumPrice, message:

`转售价格必须比购买价至少高出30%。最低转售价格应该是 ${minimumPrice}` };

}

})]

});

}

}

<div class="form-group">

<label>Resale Price</label>

<input type="text" formControlName="resalePrice" class="form-control" />

<small class="form-text text-danger" \*ngIf="userInfoFormGroup.controls.resalePrice.errors">

{{userInfoFormGroup.controls.resalePrice.errors.minNumber.message}}<br/></small>

</div>

</div>

**用基本配置合并dynamicConfig返回的对象**

只有那些您想要动态返回的属性需要配置，因为在初始化各自的验证器时已经定义了所有属性。

在本例中，只有acceptValue通过dynamicConfig返回，而allowDecimal和message已经配置。因此，在这种情况下，如果在字段negativeNumber中验证失败，它将显示消息Positive Number

export class BindReturnValidatorComponent implements OnInit {

userInfoFormGroup: FormGroup

ngOnInit() {

this.userInfoFormGroup = this.formBuilder.group({

negativeNumber: ['', RxwebValidators.numeric({

dynamicConfig: (x, y) => {

if (x.typeValue == 1)

return { acceptValue: NumericValueType.PositiveNumber }

else

return { acceptValue: NumericValueType.NegativeNumber }

},

allowDecimal: false,

message: 'Positive Integer'

})

]

});

}

}

**管理通用嵌套级别属性**

我们通过类对象创建一个FormGroup。

这里我们讨论类对象属性中非常常见的场景。例如说我们学生模型和我们想要存储的信息的学科和课程的科目是propArray包含通用属性subjectDetails在这个我们需要传递参数T和它应该工作实例的基础上创建的对象。

例子

首先，我们需要创建一个用户类，并在模型中定义属性。

import { prop, propArray} from "@rxweb/reactive-form-validators"

export class Course{

@prop()

courseId : number;

}

export abstract class Student<T> extends Course {

@propArray() SubjectDetails: T[];

}

export class Subjects {

@prop()

enrollmentNumber: number;

}

export class StudentCourse extends Student<Subjects> {

}

export class GenericCompleteComponent implements OnInit {

ngOnInit() {

let studentCourse = new StudentCourse();

studentCourse.SubjectDetails = new Array<Subjects>();

studentCourse.SubjectDetails.push(new Subjects);

this.studentFormGroup = this.formBuilder.formGroup(studentCourse);

}

getFormArray(){

let formarray = this.studentFormGroup.controls.SubjectDetails as FormArray;

return formarray.controls;

}

}

<form [formGroup]="studentFormGroup">

<div [formGroup]="studentGroup" \*ngFor="let studentGroup of getFormArray()">

<div class="form-group">

<label>Enrollment Number</label>

<input type="text" formControlName="enrollmentNumber" class="form-control" />

</div>

<button [disabled]="!studentFormGroup.valid" class="btn btn-primary">Submit</button>

</div>

</form>

**ModifiedValue**

<https://docs.rxweb.io/how-to/modifiedValue/validators>

当使用大型表单时，当用户想要修补值时，RxFormGroup为它提供了modifiedValue属性。

它将更新RxFormGroup的值，只更改列值。

export class ModifiedValueCompleteValidatorComponent implements OnInit {

ngOnInit() {

this.userInfoFormGroup = <RxFormGroup>this.formBuilder.group({

firstName:[''],

address:{ areaName:''},

hobbies:[ { name:''}]

});

}

getFormArray(){

let formarray = this.userInfoFormGroup.controls.hobbies as FormArray;

return formarray.controls;

}

modifiedvalue(){this.userInfoFormGroup.modifiedValue }

}

<div>

<form [formGroup]="userInfoFormGroup">

<div class="form-group">

<label>FirstName</label>

<input type="text" formControlName="firstName" class="form-control" />

</div>

<div class="form-group" [formGroup]="userInfoFormGroup.controls.address">

<label>areaName</label>

<input type="text" formControlName="areaName" class="form-control" />

</div>

<div class="form-group" [formGroup]="hobbyGroup" \*ngFor="let hobbyGroup of getFormArray()">

<label>Name</label>

<input type="text" formControlName="name" class="form-control" />

</div>

<button (click)="modifiedvalue()" class="btn btn-primary">Update</button>

</form>

</div>

**多语种验证消息**

<https://docs.rxweb.io/how-to/multilingualValidationMessage/validators>

为了在angular响应式表单中使用@rxweb/ reactiveform -validators的多语言验证消息，这是通过在根组件中设置ReactiveFormConfig的配置来实现的，组件html中的errorMessage属性会根据所选的语言绑定验证消息。用于设置多语言验证消息。请参考使用ngx-translate-extension验证消息

**Root Module Configuration**

ReactiveFormConfig.i18n = {

validationMessage: () => { return TranslationCore.getText("global.validationMessages") }

};

Object.defineProperty(ReactiveFormConfig.i18n, "language", {

get: () => { return this.rxTranslation.language; }

})

}

让我们看看如何使用响应式表单验证器实现多语言验证消息

在下面的例子中，firstName应用了所需的验证器，而userName字段应用了alpha验证器。验证消息从资产的i18n文件夹解析，并使用errorMessage属性绑定。

export class MultilingualValidationMessageCompleteValidatorComponent implements OnInit {

ngOnInit() {

this.userInfoFormGroup = this.formBuilder.group({

firstName:['',RxwebValidators.required()],

userName:['',RxwebValidators.alpha()]

**Patch Model Value**

https://docs.rxweb.io/how-to/patchModelValue/validators

在使用响应式表单时，如果表单控件已经创建，并且从服务器接收到数据，则可能需要更新特定表单控件的值。

patchModelValue将根据提供的服务器JSON对象或模型对象更新FormGroup中的FormControl的值。它将从FormGroup中更新特定的FormControl的值。

例子

export class PatchCompleteValidatorComponent implements OnInit {

ngOnInit() {

this.userInfoFormGroup = <RxFormGroup>this.formBuilder.group({

firstName:[''],

address:{ areaName:'' },

hobbies:[ { name:''}]

});

}

getFormArray(){

let formarray = this.userInfoFormGroup.controls.hobbies as FormArray;

return formarray.controls;

}

patchvalue() {

this.userInfoFormGroup.patchModelValue( {firstName:'John',

address: { areaName: "Victoria" },

hobbies: [{ name: "Travelling" }] });

}

}

**Post as FormData**

https://docs.rxweb.io/how-to/formData/validators

由于我们正在使用大型企业面向服务的应用程序，可能需要以FormData的形式发布数据对象，而不是JSON。

现在，这可以很容易地用RxFormBuilder完成。这提供了一个'toFormData()'方法，它将FormGroup值转换为FormData。

让我们创建一个FormGroup，并在发布用户数据时使用'toFormData'方法。

**发送formData到服务器**

要将formData发送到服务器，你必须使用RxFormBuilder服务在组件中创建FormGroup。

export class FormadataCompleteValidatorComponent implements OnInit {

userFormGroup: RxFormGroup

api:string = 'api/User'

constructor(private formBuilder: RxFormBuilder, private http: HttpClient ){ }

ngOnInit() {

this.userFormGroup = <RxFormGroup>this.formBuilder.group({

firstName:[''], lastName :[''],userName:[''], password : ['']

});

}

addUser(){

let formdata = this.userFormGroup.toFormData()

this.http.post(this.api, formdata); // This is fake uri, This is just for your reference.

}

}

<div>

<form [formGroup]="userFormGroup">

<div class="form-group">

<label>Password</label>

<input type="text" formControlName="password" class="form-control" />

</div>

<button [disabled]="!userFormGroup.valid" (click)="addUser()" class="btn btn-primary">Submit</button>

</form>

</div>

**Post Image through formData**

https://docs.rxweb.io/how-to/fileObject/validators

首先，我们必须通过RxFormBuilder服务创建一个FormGroup，该服务提供了一个toFormData()方法，我们可以用这个方法将JSON值转换为FormData。但我们需要的是传递图像数据。因此，传递图像数据时，我们必须在htmlputfileelement上设置writeFile="true"属性。

就是这样。其他事情可以由框架本身管理。让我们看看下面的代码示例。

ngOnInit() {

this.userInfoFormGroup = <RxFormGroup>this.formBuilder.group({

profilePhoto:[''],

});

}

addUser(){

let formdata = this.userInfoFormGroup.toFormData()

this.http.post(this.api, formdata); // This is fake uri, This is just for your reference.

}

<div class="form-group">

<label>Profile Photo</label>

<input type="file" [writeFile]="true" formControlName="profilePhoto" class="form-control" **multiple** />

</div>

<button [disabled]="!userInfoFormGroup.valid" class="btn btn-primary">Submit</button>

**允许未索引属性属性**

使用不带fileList值的索引格式的属性名。

使用formDataConfig中的excludeImageIndex属性来使用不带索引格式的fileList值。

addUser() {

let formdata = this.userInfoFormGroup.toFormData({

**excludeImageIndex**: true

});

this.http.post(this.api, formdata); // This is fake uri, This is just for your reference.

}

<div class="form-group">

<label>Profile Photo</label>

<input type="file" [writeFile]="true" formControlName="profilePhoto" class="form-control" **multiple** />

</div>

**About Reset Form**

在开发企业级应用程序时，有很多场景需要使用最初定义的值来更新表单。

Reset form提供以下功能:

**ResetFormType**

**1 ControlsOnly**

它将只重置formControls。

resetForm(){ this.userFormGroup.resetForm({resetType:ResetFormType.ControlsOnly}); }

**2 FormGroupsOnly**

它将只重置formGroups

resetForm(){this.userFormGroup.resetForm({resetType:ResetFormType.FormGroupsOnly}); }

**3 FormArraysOnly**

它只会重置formArrays

resetForm(){this.userFormGroup.resetForm({resetType:ResetFormType.FormArraysOnly});}

**4 ControlsAndFormGroupsOnly**

它将重置formGroups和formGroups

resetForm(){this.userFormGroup.resetForm({resetType:ResetFormType.ControlsAndFormGroupsOnly})

**5 DefinedPropsOnly**

它将重置已定义的 props

resetForm(){

this.userFormGroup.resetForm({resetType:ResetFormType.DefinedPropsOnly,

value:{ firstName: 'John' }}); }

**6 All**

它将重置所有的，包括formGroups,formGroups和formarray

resetForm(){ this.userFormGroup.resetForm({resetType:ResetFormType.All}); }

**7 with**

Type : string[]

如果你想要一些特定的formControls或formArray来恢复，你需要设置和指定formControl或formArray的名称。

resetForm(){this.userFormGroup.resetForm({ with: ["firstName","address.areaName"] }); }

**8 value**

如果要重置特定属性，可以使用值属性。

resetForm(){ this.userFormGroup.resetForm({ value: { firstName: 'John' } }); }

**设置和清除后端消息**

https://docs.rxweb.io/how-to/backEndMessages/validators

推后端验证消息并清除它们，这是在HTTP回调之后根据需要在FormControl和FormGroup级别完成的。通常我们会在app组件中全局配置验证消息，或者使用message或messageKey参数进行设置

此概念用于需要设置和清除来自服务器的验证消息的表单。下面是一个用例，其中有一个用户信息表单，其中有像firstName、lastName这样的字段和一个嵌套的formGtrup地址，其中包含areaName字段。这涉及到两种不同的情况，即后台验证消息是使用RxFormControl和RxFormGroup设置的。

**FormControl Level**

在FormControl级别设置后端验证消息是通过使用RxFormControl来完成的，它是由FormControl类扩展的。使用setBackEndErrors设置验证器，clearBackEndErrors清除验证消息

**getValidators**

此方法用于在手动重新加载验证器时获取formControl级别的验证器。在下面的例子中，getFirstNamerrors被用来获得formControl的firstName验证器，而getAreaNameErrors被用来获得嵌套的formGroup的areaName验证器

export class BackEndMessagesCompleteValidatorComponent implements OnInit {

userInfoFormGroup: RxFormGroup

firstNameCount:any=''

areaNameCount : any = ''

constructor(

private formBuilder: RxFormBuilder) { }

ngOnInit() {

this.userInfoFormGroup = <RxFormGroup>this.formBuilder.group({

firstName: ['', RxwebValidators.required()],

address: <RxFormGroup>this.formBuilder.group({

areaName: ['', RxwebValidators.required()]

})

});

}

setFirstNameBackEndErrors() {

(<RxFormControl>this.userInfoFormGroup.controls.firstName)

.setBackEndErrors({ invalidValue: '输入值为“无效”' });

}

setAreaNameBackEndErrors() {

(<RxFormControl>this.userInfoFormGroup.get('address')['controls'].areaName)

.setBackEndErrors({ invalidValue: '输入值为“无效”' });

}

clearFirstNameBackEndErrors() {

(<RxFormControl>this.userInfoFormGroup.controls.firstName)

.clearBackEndErrors();

}

clearAreaNameBackEndErrors() {

(<RxFormControl>this.userInfoFormGroup.get('address')['controls'].areaName)

.clearBackEndErrors();

}

getAreaNameErrors() {

let validators =

(<RxFormControl>this.userInfoFormGroup.get('address')['controls'].areaName).getValidators();

validators.push(RxwebValidators.alpha());

this.userInfoFormGroup.get('address')['controls'].areaName.setValidators(validators);

this.areaNameCount = (<RxFormControl>this.userInfoFormGroup.get('address')['controls'].areaName)

.getValidators().length;

}

getFirstNamerrors() {

let validators = (<RxFormControl>this.userInfoFormGroup.controls.firstName).getValidators();

validators.push(RxwebValidators.alpha());

this.userInfoFormGroup.controls.firstName.setValidators(validators);

this.firstNameCount = (<RxFormControl>this.userInfoFormGroup.controls.firstName)

.getValidators().length;

}

}

Next, we need to write html code.

<div>

<form [formGroup]="userInfoFormGroup">

<div class="form-group">

<label>FirstName</label>

<input type="text" formControlName="firstName" class="form-control" />

<small class="form-text text-danger"

\*ngIf="userInfoFormGroup.controls.firstName.errors">

{{userInfoFormGroup.controls.firstName.errorMessage}}<br /></small>

<small> FirstName Validators Count : {{firstNameCount}}</small>

</div>

<div class="form-group" [formGroup]="userInfoFormGroup.controls.address">

<label>areaName</label>

<input type="text" formControlName="areaName" class="form-control" />

<small class="form-text text-danger"

\*ngIf="userInfoFormGroup.get('address')['controls'].areaName.errors">

{{userInfoFormGroup.get('address')['controls'].areaName.errorMessage}}<br /></small>

<small> FirstName Validators Count : {{areaNameCount}}</small>

</div>

</form>

</div>

**FormGroup Level**

在FormGroup级别，它使用setBackEndErrors来设置验证器，clearBackEndErrors通过创建一个RxFormGroup类型的对象来清除验证消息，该对象扩展了FormGroup类，提供了上面提到的方法

export class BackEndMessagesAddValidatorComponent implements OnInit {

ngOnInit() {

this.userInfoFormGroup = <RxFormGroup>this.formBuilder.group({

firstName: ['', RxwebValidators.required()],

address: <RxFormGroup>this.formBuilder.group({

areaName: ['', RxwebValidators.required()]

})

});

}

setEndErrors() {

this.userInfoFormGroup.setBackEndErrors(

{

address: { areaName: { 'required': 'Invalid City' } },

firstName: { 'required': 'The entered value is Invalid' }

});

}

clearBackEndErrors() {

this.userInfoFormGroup.clearBackEndErrors();

}

}

Next, we need to write html code.

<div>

<form [formGroup]="userInfoFormGroup">

<div class="form-group">

<label>FirstName</label>

<input type="text" formControlName="firstName" class="form-control" />

<small class="form-text text-danger"

\*ngIf="userInfoFormGroup.controls.firstName.errors">

{{userInfoFormGroup.controls.firstName.errorMessage}}<br /></small>

</div>

<div class="form-group" [formGroup]="userInfoFormGroup.controls.address">

<label>areaName</label>

<input type="text" formControlName="areaName" class="form-control" />

<small class="form-text text-danger"

\*ngIf="userInfoFormGroup.get('address')['controls'].areaName.errors">

{{userInfoFormGroup.get('address')['controls'].areaName.errorMessage}}<br /></small>

</div>

<div class="btn btn-group">

<button (click)="setEndErrors()" class="btn btn-sm btn-primary">Set Errors</button>&nbsp;

<button (click)="clearBackEndErrors()" class="btn btn-sm btn-primary">Clear Errors</button>

</div>

</form>

</div>

强类型反应式表单

https://docs.rxweb.io/how-to/stronglyType/decorators

在开发复杂表单时，相应的表单控件应该与模型一起强类型化。这可以通过“IFormGroup”接口实现。

让我们看看它是如何工作的。

首先，我们创建一个User模型并定义fullName属性。

export class User{

@prop()

fullName:string;

}

现在我们创建一个FormGroup，但是我们的FormGroup对象类型是IFormGroup。IFormGroup是一个接口，它与FormGroup类一起继承，这样你就可以访问FormGroup的特权以及RxFormGroup的其他方法。

export class StronglyTypeAddComponent implements OnInit {

userFormGroup: IFormGroup<User>

constructor( private formBuilder: RxFormBuilder ) { }

ngOnInit() {

this.userFormGroup = this.formBuilder.formGroup(User) as IFormGroup<User>;

let user:User = this.userFormGroup.modelInstance;

}

}

Next, we need to write html code.

<div>

<form [formGroup]="userFormGroup">

<div class="form-group">

<label>Full Name</label>

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

</div>

<span> {{userFormGroup.modelInstance|json}}</span>

</form>

</div>

Full Name

{}

根据定义的类型维护formControl值

当值来自任何第三方UI组件或HTML控件时，强模型类型将失败。就像下面的例子，我们绑定一个下拉选项，值是字符串形式，但我们的模型属性类型是布尔。为了克服这个问题，我们必须使用toBoolean杀毒器将字符串的值转换为布尔值并存储。

export class User{

@prop()

fullName:string;

@prop()

age: number;

@prop()

@toBoolean()

accepted: boolean;

}

export class StronglyTypeDefinedComponent implements OnInit {

acceptCookies =[{"text":"Yes",value:"True"},{"text":"No",value:"false"}];

userFormGroup: IFormGroup<User>

constructor(private formBuilder: RxFormBuilder) { }

ngOnInit() {

this.userFormGroup = this.formBuilder.formGroup(User) as IFormGroup<User>;

}

}

Next, we need to write html code.

<div>

<form [formGroup]="userFormGroup">

<div class="form-group">

<label for="accepted">Do you accept cookies?</label>

<select id="accepted" formControlName="accepted" class="form-control">

<option [value]="item.value" \*ngFor="let item of acceptCookies">{{item.text}}</option>

</select>

</div>

</form>

{{this.userFormGroup.modelInstance | json}}

</div>

**逻辑验证器**

<https://docs.rxweb.io/how-to/logicalOperators/validators>

在开发复杂表单时，逻辑验证在其中扮演着重要的角色。尝试简单地设置逻辑条件，您可以根据需要使用三个逻辑验证器。逻辑验证器如下所示:

**1 or operator Example**

这是可以使用的，如果我们想要验证控件基于任何验证器将被传递，例如有一个电子邮件字段，你想要包含验证或endsWith验证被传递。

this.userInfoFormGroup = this.formBuilder.group({

email :['',RxwebValidators.**or**(

{validation:{contains:{value:'@gmail.com'},endsWith:{value:'.com'}}})]

});

**2 and operator Example**

如果我们想验证基于所有验证器传递的控件，例如，在你的用户表单中有一个Flat Address字段，你想验证它，这样字母数字和maxLength都应该被传递。

this.userFormGroup = this.formBuilder.group({

flatAddress:['', RxwebValidators.**and**(

{validation:{alphaNumeric:{allowWhiteSpace:true},maxLength:{value:50}}})]

});

**3 not operator Example**

当您希望验证执行其逆时，使用Not运算符。假设有一个年龄字段，您想要验证它，使年龄应该在9到100之间，使用minNumber和maxNumber验证，不使用操作符。

this.userFormGroup = this.formBuilder.group({

age:['', RxwebValidators.**not**(

{validation:{maxNumber:{value:9},minNumber:{value:100}}})],

});

**使用验证器有条件地禁用控件**

对于要求根据其他输入值禁用formControls的响应式表单，这里有一个属性，可以根据应用的条件禁用输入。属性disableExpression用于与应用于实现相同目的的验证一起使用

让我们考虑这样一个场景:有一个下订单表单，它需要当前地址详细信息，如果用户在用户配置文件中保存了地址，那么他可以使用相同的地址下订单，并添加新地址。在上述情况下，如果“same as permanent address”的复选框设置为true，则currentAddress字段将被禁用。

export class ConditionalControlDisableCompleteValidatorComponent implements OnInit {

constructor( private formBuilder: FormBuilder ) { }

ngOnInit() {

this.userFormGroup = this.formBuilder.group({

permanentAddress:['F-16, Victoria Lake',RxwebValidators.required()],

IsSameAddress:[''],

currentAddress:['', RxwebValidators.required(

{ **disableExpression: (x, y) => { return x.IsSameAddress == true }})**],

});

}

}

**Auto Push FormGroup into Array using Model**

在基于模型的反应式表单中，每当propArray在FormGroup中创建RxFormArray时，每当用户想要在特定的FormGroup中添加一个新的索引时，一个新的FormGroup就会被推入。但在某种形式下，需要完成多个操作，并基于向模型类中添加实例将formgroup创建到数组中。

这种场景的一个例子是Cart模型，在这个模型中，用户可以向列表中添加产品数量，这个Cart模型创建RxFormGroup对象，其中包含propArray。现在，在这种情况下，需要自动创建FormGroup，并在Cart模型类中添加新项目时添加到FormArray中。让我们借助代码来理解这一点

第一步是在应用组件的全局级别上设置autoInstancePush属性为true进入翻译页面

export class Cart {

@unique()

cartItem: string;

}

export class User {

@propArray(Cart)

carts: Cart[];

}

export class AutoPushFormGroupCompleteComponent implements OnInit {

userFormGroup: FormGroup

user: User;

constructor(private formBuilder: RxFormBuilder) { }

ngOnInit() {

this.user = new User();

this.user.carts = new Array<Cart>();

this.user.carts.push(new Cart());

this.userFormGroup = this.formBuilder.formGroup(this.user);

}

addCartItem() {

**this.user.carts.push( new Cart());**

}

**GreaterThan/LessThan validation in FormArray**

就像数组有很多验证用例一样，下面是@rxweb响应式表单涉及的另一个验证场景，即将greaterThan和lessThan验证到数组中。使用场景是这样的:在greaterThan验证器中，前一个索引大于下一个索引，在小于验证器中，前一个索引小于下一个索引。以下是描述的案例:进入翻译页面

数组中的大于验证

GreaterThan验证将以这样一种方式验证索引，即前一个索引必须大于下一个推入的索引。下面是数组中GreaterThan验证的有效例子。进入翻译页面

**管理翻译后的验证消息**

当使用像ngx-translate这样的翻译库时，通过它可以使用messageKey显示翻译后的验证消息。会有很多validationMessage道具的映射。在这种情况下，如果你将isAddMessageKey属性标记为true，它将向refValues对象添加messageKey。

export class NgxTranslateValidationMessageCompleteValidatorComponent implements OnInit {

constructor(private formBuilder: FormBuilder ){ }

ngOnInit() {

this.userFormGroup = this.formBuilder.group({

fullName:['', RxwebValidators.required({messageKey:'requiredMessageKey',isAddMessageKey:true })],

});

}

}

Next, we need to write html code.

<div>

<form [formGroup]="userFormGroup">

<div class="form-group">

<label>Full Name</label>

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

<small class="form-text text-muted">显示基于消息键的自定义验证消息<br/></small>

<small class="form-text text-danger" \*ngIf="userFormGroup.controls.fullName.errors">

{{userFormGroup.controls.fullName.errors.required.message}}<br/></small>

</div>

<button [disabled]="!userFormGroup.valid" class="btn btn-primary">Submit</button>

</form>

</div>

**FormBuilderconfiguration**

FormBuilderconfiguration用于动态验证FormControl以及其他一些活动，比如在运行时创建嵌套对象的实例。

这为在使用基于Decorator的验证方法时添加新属性或排除属性提供了灵活性。

当在响应式表单中使用嵌套的formGroup时，如果你的嵌套对象属性是未定义的，那么formbuilder将不会创建嵌套的formGroup，在这种情况下，你需要通过FormBuilderConfiguration传递动态配置。

**创建一个对象实例**

如果你在组件中嵌套了formGroup，而你的组件中的模型类中可能没有那个特定的属性，装饰器就不能配置属性。

在这种情况下，对象属性实例可以使用objectPropInstanceConfig来创建，它在AutoInstanceConfig接口下，你需要设置对象的propertyName到它。

例如，你有一个用户表单，在编辑过程中，你想要一个名为state的属性，这个属性并不存在于你的模型类中，所以你可以使用它来创建它

var formBuilderConfig = new FormBuilderConfiguration();

formBuilderConfig.autoInstanceConfig = {

objectPropInstanceConfig: [{ propertyName: 'address', }]

}

**创建一个数组实例**

当访问formGroup控件时，如果你的数组对象没有定义，并且你想在运行时创建数组对象实例，你可以使用arrayPropInstanceConfig来创建它，你可以在其中设置你想要的属性名和行项的数量。

var formBuilderConfig = new FormBuilderConfiguration();

formBuilderConfig.autoInstanceConfig = {

arrayPropInstanceConfig: [{

propertyName: 'skills',

rowItems: 2

}]

}

this.userInfoFormGroup = <RxFormGroup>this.formBuilder.formGroup(user,formBuilderConfig);

**包含和排除属性**

尽管formGroup访问数据,基于你要执行的操作,如果你想包括或排除属性从你formGroup已经定义的模型类,它可以通过使用includeProps excludeProps只是设置propetyNames要包括或排除。

formBuilderConfig.includeProps = ["userName","password","confirmPassword"]

**对所有props应用验证**

当需要对所有formControls应用验证时使用。可以使用allProps属性来完成

export class FormbuilderconfigurationAllpropsComponent implements OnInit {

userInfoFormGroup: RxFormGroup

constructor(

private formBuilder: RxFormBuilder) { }

ngOnInit() {

var formBuilderConfig = new FormBuilderConfiguration();

formBuilderConfig.applyAllProps = [RxwebValidators.required()]

this.userInfoFormGroup = <RxFormGroup>this.formBuilder.group({

firstName: [''],

lastName: ['']

}, formBuilderConfig)

}

}

**动态设置默认值**

这主要用于通过组件应用默认值，而不是定义到装饰器（decorator）中。

export class FormbuilderconfigurationDynamicdefaultComponent implements OnInit {

userFormGroup: FormGroup

constructor(private formBuilder: RxFormBuilder) { }

ngOnInit() {

let user = new User();

var formBuilderConfig = new FormBuilderConfiguration();

formBuilderConfig.**propsConfig** = {'emailAddress':{defaultValue:"bharat@gmail.com"}}

this.userFormGroup = this.formBuilder.formGroup(user,formBuilderConfig);

}}

**通用的实体 Generic Entities**

使用genericEntities属性通过向抽象类传递实体来设置需要在formGroup中访问的泛型实体。

export class FormbuilderconfigurationGenericentitiesComponent implements OnInit {

studentFormGroup: FormGroup

constructor(private formBuilder: RxFormBuilder) { }

ngOnInit() {

let studentCourse = new StudentCourse();

studentCourse.SubjectDetails = new Array<Subjects>();

studentCourse.SubjectDetails.push(new Subjects);

var formBuilderConfig = new FormBuilderConfiguration();

formBuilderConfig.genericEntities ={ 'SubjectDetails': Subjects}

this.studentFormGroup = this.formBuilder.formGroup(studentCourse,formBuilderConfig);

}

getFormArray(){

let formarray = this.studentFormGroup.controls.SubjectDetails as FormArray;

return formarray.controls;

}

}

**ignoreUndefinedProps**

它用于使用formBuilderConfiguration动态忽略未定义的属性。

export class FormbuilderconfigurationIgnoreundefinedComponent implements OnInit {

userInfoFormGroup: RxFormGroup

constructor(private formBuilder: RxFormBuilder) { }

ngOnInit() {

let user = new User();

user.address = new Address();

var formBuilderConfig = new FormBuilderConfiguration();

formBuilderConfig.ignoreUndefinedProps = ["address"];

this.userInfoFormGroup = <RxFormGroup>this.formBuilder.formGroup(user,formBuilderConfig);

}}

**abstractControlOptions**

控件要更新的事件名称的选项，称为abstractControlOptions。他们是blur, change,submit。

export class FormbuilderconfigurationAbstractControlOptionsComponent implements OnInit {

userInfoFormGroup: FormGroup

constructor(private formBuilder: RxFormBuilder ){ }

ngOnInit() {

this.userInfoFormGroup = this.formBuilder.group(

{ firstName:['', RxwebValidators.required()], },

{ abstractControlOptions: { firstName: 'blur' } }

);

}

}

**ReactiveFormConfig**

ReactiveFormConfig 用于在应用程序开始时全局配置日期格式、验证消息。

**About ReactiveFormConfig**

ReactiveFormConfig用于全局设置日期格式，每当应用程序启动时，验证消息将自动初始化。我们必须设置“validationMessage”来设置全局验证消息，“baseConfig”用来设置全局dateformat，“internationalization”用来设置全球应用程序的dateformat。

**baseConfig**

要在整个应用程序中设置日期格式，进行日期验证检查，ReactiveFormConfig的baseConfig是一个配置，如果你在'minDate'和'maxDate'验证中设置了value参数'value'，否则它将采用'mm/dd/yyyy'的默认格式。假设你的国家日期格式是“mm/dd/yy”，设置日期格式为“mdy”，如果是“dd/mm/yy”，设置日期格式为“dmy”，格式将被全局设置。

ReactiveFormConfig.set({"baseConfig": { "dateFormat": "mdy","seperator": "/" } });

**internationalization**

假设您希望应用程序在全球范围内工作，并且希望根据您的地区验证日期格式，在这种情况下，如果您的系统支持国际化，您可以动态地更改它。

ReactiveFormConfig.set({

"baseConfig": {

"dateFormat": "mdy",

"seperator": "/",

"internationalization": {

"dateFormat": "mdy",

"seperator": "/"

}

});

**Validation Messages**

在整个应用程序中应用全局验证消息，然后全局配置验证消息。您必须为使用的每个验证器设置消息。

ReactiveFormConfig.set({

"validationMessage": {

"required": "this field is required.",

"alpha": "Only alphabets are allowed.",

......

}

});

**Password validation Messages**

您可以在ReactiveFormConfig中全局设置多个密码验证消息，如下所示。

ReactiveFormConfig.set({

"validationMessage": {

"password": {

minLength: 'Minimum Character length should be 5.',

maxLength: 'MaxLength allowed is 5',

password: "Invalid Password"

},

}

});

**ErrorMessageBindingStrategy**

在整个应用程序中显示错误消息的策略相同时，全局配置错误消息策略。

默认情况下errorMessageBindingStrategy为errorMessageBindingStrategy.none。

根据你的需要，你可以将errorMessageBindingStrategy设置为以下条件:

ErrorMessageBindingStrategy.none :

没有条件来绑定错误消息，一旦FormControl无效，错误消息就会被绑定。

ErrorMessageBindingStrategy.onSubmit :

当RxFormGroup属性提交为true时，无效的FormControl错误消息将被绑定。

ErrorMessageBindingStrategy.onDirty:

一旦FormControl被脏了，在那之后无效的FormControl错误消息将被相应地绑定。

ErrorMessageBindingStrategy.onTouched:

一旦触摸了FormControl，无效的FormControl错误消息就会相应地被绑定。

ErrorMessageBindingStrategy.onDirtyOrTouched:

一旦FormControl被Dirty或Touched，无效的FormControl错误消息将被相应地绑定。

ReactiveFormConfig.set({

reactiveForm: {

errorMessageBindingStrategy: ErrorMessageBindingStrategy.onSubmit

}

})

**allowDecimalSymbol**

假设您想要在整个应用程序中设置相同的十进制符号，那么您必须在app.component.ts中的ReactiveFormConfig中全局配置该十进制符号。例如，如果你想设置"，"作为分隔符，而不是"."，那么你可以使用allowDecimalSymbol来配置它。

ReactiveFormConfig.set({ "allowDecimalSymbol": ","})

**defaultValidationLocale**

这将允许在alpha和字母数字验证中基于指定的验证字母区域设置的多语言字符。它可以设置为西班牙语，丹麦语，法语，德语，俄语。在这里，它被全局设置为西班牙语，用于alpha和字母数字验证。

ReactiveFormConfig.set({

defaultValidationLocale: {

alpha: ValidationAlphabetLocale.spanish,

alphaNumeric: ValidationAlphabetLocale.spanish

}

});

**RxFormBuilder**  **group**

<https://docs.rxweb.io/api/RxFormBuilder/decorators#group>

您可以通过此服务创建FormGroup。这提供了一个group方法和formGroup方法。它们都用于创建FormGroup。两者都可以在不同的情况下使用。

让我们看看在哪种情况下我们可以使用哪种方法。

**group**

它提供了Angular FormBuilder组方法的所有特性，以及其他一些在创建复杂表单时很重要的特性，比如如果我们想动态创建一个FormGroup，或者我们不想预先配置验证器。验证规则来自服务器，并在相应的FormControl上绑定验证规则运行时。

在这里，我们讨论了三个可以通过小组方法实现的案例

在深入代码之前，让我们先了解组方法的定义。

group(groupObject: { [key: string]: any; }, validatorConfig?: FormBuilderConfiguration): FormGroup

group方法接受json对象，该对象将根据指定的值转换为FormControl, FormGroup或FormArray，第二个参数用于动态验证，我们将在本文档结束时讨论。

Create the FormGroup

我们会讨论在Angular FormBuilder中使用的类似方法。但唯一的区别是，不需要在不调用group方法的情况下创建嵌套的FormGroup。

这里，我们创建了一个FormControl，嵌套的FormGroup和嵌套的FormArray在FormGroup中。参见下面的代码

export class UserComponent implements OnInit {

userInfoFormGroup: FormGroup

constructor( private formBuilder: RxFormBuilder ) { }

ngOnInit() {

this.userInfoFormGroup = this.formBuilder.group({

userName:['',Validators.required],

password:['',[Validators.required,Validators.min(5)]],

confirmPassword:['',RxwebValidators.compare({fieldName:'password'})],

address:{ areaName:'' },

hobbies:[{ name:'' }]

});

}

}

**动态创建一个带有动态验证规则的FormGroup**

在这里，我们动态创建一个FormGroup，并应用相应的FormControl的验证规则。为了更好地理解，我们首先看到来自服务器的JSON数据，为了更好地理解，让我们给用户对象一个名称。这个用户对象有一些验证规则，这些规则也是来自服务器。

让我们看看下面的用户对象:

{

"firstName":"John",

"lastName":"Cena"

}

参见下面的动态验证规则:

{

"firstName":{ "required":true },

"lastName":{ "required":true }

}

现在，对于动态服务器验证规则，我们可以调用服务器或放置验证规则的JSON对象。

：这里我们将静态JSON对象的验证规则，见下面的代码:

this.userInfoFormGroup = this.formBuilder.group(

{firstName:'', lastName:'' },

{dynamicValidation:{"firstName":{"required":true},

"lastName":{"required":true}

}

});

**formGroup**

如果你正在使用基于装饰器的验证方法，那么你的FormGroup将通过这个方法创建。创建FormGroup的方法有很多。我们将逐一讨论所有的方法。

在基于装饰器的验证中，最重要的是你必须先定义一个模型，然后才能创建FormGroup。

让我们创建一个User模型:

export class User {

@required()

firstName:string;

@required()

lastName:string;

}

创建一个FormGroup通过模型没有实例

此场景可用于添加表单。

this.userInfoFormGroup = this.formBuilder.formGroup(User);

通过模型实例创建一个FormGroup

this.user = new User();

this.userInforFormGroup = this.formBuilder.formGroup(this.user);

用服务器数据创建一个FormGroup，服务器值应该映射到模型属性

这可以在编辑表单中使用，当值来自服务器时，我们必须创建表单，并且验证规则可以基于配置在User Model属性上的验证装饰器应用。

let jObject = {"firstName":"John","lastName":"Cena"};

this.userInfoFormGroup = this.formBuilder.formGroup(User,jObject);

根据上面的代码，你必须传递两个参数，第一个参数是用户模型，第二个是服务器json数据。服务器数据JSON对象属性应该与用户模型属性相同。

**control**

使用RxFormBuilder的控件方法创建一个新的formControl。

export class ControlCompleteComponent implements OnInit {

countryFormGroup: FormGroup

constructor(private formBuilder: RxFormBuilder ) { }

ngOnInit() {

let countryName = this.formBuilder.control('', [RxwebValidators.required()]);

this.countryFormGroup = this.formBuilder.group({

countryName

});

}

}

**RxFormGroup**

**modelInstance**

https://docs.rxweb.io/api/RxFormGroup/decorators#bindingerrormessagesatonce

**使用modelInstance获取模型对象值**

如果你想访问你的formGroup的模型对象的值，你可以使用RxFormGroup的modelInstance属性。

注意:如果你想访问模型对象的值而不影响根模型，使用modelInstanceValue。

export class RxFormGroupModelInstanceComponent implements OnInit {

userFormGroup: RxFormGroup

modelInstance :{};

constructor( private formBuilder: RxFormBuilder ) { }

ngOnInit() {

let user = new User();

this.userFormGroup = <RxFormGroup>this.formBuilder.formGroup(user);

this.modelInstance = this.userFormGroup.modelInstance;

}

}

**立即绑定errorMessages**

使用getErrorSummary方法可以一次获得所有errorMessages的摘要。

export class RxFormGroupErrorMessageSummaryComponent implements OnInit {

userFormGroup: RxFormGroup

errorMessagesOnSubmit:{}

constructor(private formBuilder: RxFormBuilder ){ }

ngOnInit() {

this.userFormGroup = <RxFormGroup>this.formBuilder.group({

userName:['', RxwebValidators.alpha()],

email:['',RxwebValidators.required()]

});

}

errorSummary(){

this.errorMessagesOnSubmit = this.userFormGroup.getErrorSummary(true);

}

}

**About Dirty Check**

https://docs.rxweb.io/how-to/dirtyCheck/validators

最适合在编辑表单中使用脏检查的情况。这是最常见的场景，我们需要根据提供的值进行检查，如果值被更改，那么表单就会变脏。

Angular在FormControl中提供了一个dirty字段。但是，如果用户再次根据初始化后的值更改该值，这将不起作用。但是通过isDirty函数，你可以达到同样的效果。

**isModified**

isModified属性用于检查formControl的值是否被改变，而不检查整个状态。如果formControl的值被改变了，状态就会改变，否则不会。

export class DirtyModifiedValidatorComponent implements OnInit {

editForm:FormGroup;

constructor(private formBuilder:RxFormBuilder){}

ngOnInit(){

this.editForm = this.formBuilder.group({

id:[1],

name: ["Bharat" ],

designation: ["Software Engg."]

});

}

}

Next, we need to write html code.

<div>

<form [formGroup]="editForm">

<button [disabled]="!editForm.**isModified**" class="btn btn-primary">Is Modified</button>

</form>

</div>

patchModelValue

https://docs.rxweb.io/how-to/patchModelValue/validators

在使用响应式表单时，如果表单控件已经创建，并且从服务器接收到数据，则可能需要更新特定表单控件的值。

patchModelValue将根据提供的服务器JSON对象或模型对象更新FormGroup中的FormControl的值。它将从FormGroup中更新特定的FormControl的值。

export class PatchCompleteValidatorComponent implements OnInit {

userInfoFormGroup: RxFormGroup

constructor( private formBuilder: RxFormBuilder ) { }

ngOnInit() {

this.userInfoFormGroup = <RxFormGroup>this.formBuilder.group({

firstName:[''],

address:{ areaName:'' },

hobbies:[{ name:''}]

});

}

getFormArray(){

let formarray = this.userInfoFormGroup.controls.hobbies as FormArray;

return formarray.controls;

}

patchvalue()

{

this.userInfoFormGroup.patchModelValue( {firstName:'John',

address: { areaName: "Victoria" },

hobbies: [{ name: "Travelling" }] });

}

}

export class Address {

@prop() areaName: string;

}

export class Hobby {

@prop() name: string;

}

export class User {

@prop() firstName: string;

@propObject(Address) address: Address;

@propArray(Hobby) hobbies: Hobby[];

}

**disable**

https://docs.rxweb.io/decorators/disable/decorators

**禁用控制条件**

它用于有条件地禁用控件

**什么时候使用**

我们使用验证器检查了conditionalExpression，但现在出现了一种情况，根据条件，我们需要启用或禁用控件。作为一个标准的角度接近，我们将[禁用]和它的条件。一旦条件成功，它将禁用控件。

这里，我们将通过@disable装饰器管理被禁用的控件。我们必须设置条件表达式。

让我们看看如何有条件地禁用控件。

**Basic disable decorator**

场景:在响应式表单中，我们有两个字段modeType和fullName。当用户点击此时的Change Mode按钮时，它将禁用或启用基于modeType = view或modeType = edit的控件。

首先，我们需要创建一个用户模型，并在模型中定义一个modeType和fullName属性，以实现功能需求。

export class User{

@prop() modeType:string;

@disable({conditionalExpression:function(control:AbstractControl){return this.modeType == "View"; }})

@prop() fullName:string;

}

通过Angular的FormBuilder服务，我们在组件中创建了FormGroup。

export class DisableAddComponent implements OnInit {

currentMode: string = "View";

userFormGroup: FormGroup;

constructor(private formBuilder: RxFormBuilder) { }

ngOnInit() {

let user = new User();

user.modeType = this.currentMode;

this.userFormGroup = this.formBuilder.formGroup(user);

}

changeMode() {

this.currentMode = this.currentMode == "View" ? "Edit" : "View";

this.userFormGroup.controls.modeType.setValue(this.currentMode);

}

}

接下来，我们需要编写html代码。用于基于按钮单击和启用或禁用控件的切换条件

<form \*ngIf="userFormGroup" [formGroup]="userFormGroup">

<div class="form-group">

<button class="btn btn-primary" \*ngIf="userFormGroup.value.modeType == 'View'" (click)="changeMode()">

Change Mode Edit</button>

<button class="btn btn-primary" \*ngIf="userFormGroup.value.modeType == 'Edit'" (click)="changeMode()">

Change Mode View</button>

</div>

<div class="form-group">

<label>Full Name</label>

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

</div>

</form>

**elementClass**

https://docs.rxweb.io/decorators/elementClass/decorators

有条件地将类应用于表单控件元素

您可以通过TS代码有条件地设置元素类

**Basic elementClass decorator**

让我们创建一个用户模型，并在模型中定义密码属性。

export class User {

@prop()

@elementClass({

conditionalExpression: function (c) {

return this.password == "complex" && c.dirty == true ? 'ng-invalid' : '';

}})

userName: string;

@prop()

password: string;

}

你必须使用RxFormBuilder服务在组件中创建FormGroup。

**error**

https://docs.rxweb.io/decorators/error/decorators

**什么时候使用**

在UI上有条件地显示错误消息有点笨拙。作为一种标准的angular方法，我们使用\*ngIf条件并写入错误消息。一旦条件成功，它将显示错误消息。

这里，我们将通过@error装饰器管理错误消息。我们必须设置条件表达式。

让我们看看如何有条件地显示错误消息。

**Basic error decorator**

场景:在响应式表单中，有两个字段firstName和userName。主要的需求是提交没有被禁用，当用户点击提交时，它将显示无效表单字段的错误消息。

要在应用程序中全局配置消息，请参考ReactiveFormConfig

让我们创建一个用户模型，并在模型中定义freeText的属性。

export class FormField{

@prop() action:string;

@error({conditionalExpression:function(control:AbstractControl){ return this.action === "submit"}})

@required()

firstName:string;

@error({conditionalExpression:function(control:AbstractControl){ return this.action == "submit"}})

@required()

userName:string;

}

你必须使用RxFormBuilder服务在组件中创建FormGroup。

export class ErrorAddComponent implements OnInit {

errorObject = {}

userInfoFormGroup: FormGroup

constructor(

private formBuilder: RxFormBuilder

) { }

ngOnInit() {

var formField = new FormField();

this.userInfoFormGroup =<RxFormGroup>this.formBuilder.formGroup(formField);

}

submit(){

if(this.userInfoFormGroup.invalid)

this.userInfoFormGroup.controls.action.setValue("submit");

}

}

接下来，我们需要编写html代码。为了显示错误信息，你必须使用“errorMessage”属性。

<form [formGroup]="userInfoFormGroup">

<div class="form-group">

<label>First Name</label>

<input type="text" formControlName="firstName" class="form-control" />

<small class="form-text text-danger" >{{userInfoFormGroup.controls.firstName["errorMessage"]}}</small>

</div>

<div class="form-group">

<label>User Name</label>

<input type="text" formControlName="userName" class="form-control" />

<small class="form-text text-danger" >{{userInfoFormGroup.controls.userName["errorMessage"]}}</small>

</div>

<button (click)="submit()" class="btn btn-primary">Submit</button>

</form>

**model**

https://docs.rxweb.io/decorators/model/decorators

**@model**

这对于同样的验证以及有条件地显示错误消息很有帮助。

**Example**

让我们创建一个用户模型，并在模型中定义密码属性。

@model([{ propNames: [":all:"], validationConfig: { required: true } }])

export class User{

@prop() userName:string;

@prop() password:string;

@prop() cityName:string;

@prop() countryName:string;

@prop() areaName:string;

}

你必须使用RxFormBuilder服务在组件中创建FormGroup。

**elementClass**

当您希望有条件地对多个属性应用elementClass时。

@model([{

propNames: [":all:"],

elementClass: {

conditionalExpression: function (c) {

return this.password == "complex" && c.dirty == true ? 'dirty' : '';

}}

}])

export class User{}

**error**

当你想在两个以上的属性上设置error属性时，在模型装饰器的propsConfig中设置error属性会更容易。

@model([{

propNames: [":all:"],

validationConfig: { required: true },

error: {conditionalExpression:function(control:AbstractControl){ return this.userName == "john"} }

}, ])

export class User{}

**排除属性动态**

当你有两个以上的属性时，在模型装饰器的propsConfig中设置ignore属性会更容易。

@model([{

propNames: ["userName"],

ignore:(x)=>x.type == "Admin"}])

export class User{}

**prop**

https://docs.rxweb.io/decorators/prop/decorators

它用于定义模型类中字段的属性。

假设您想要创建一个用户表单，其中包含像emailAddress这样的字段，并且您想要在该字段上使用道prop饰器。

**Basic prop decorator**

首先，我们需要创建一个用户模型，并在模型中定义一个emailAddress属性，以实现功能需求。

export class User {

@prop() emailAddress: string;

}

通过Angular的FormBuilder服务，我们在组件中创建了FormGroup。

export class PropAddComponent implements OnInit {

userFormGroup: FormGroup

constructor( private formBuilder: RxFormBuilder) { }

ngOnInit() {

let user = new User();

this.userFormGroup = this.formBuilder.formGroup(user);

}

}

**Binding服务器端属性名**

如果客户端属性名和服务器属性名不同。然后将服务器属性名绑定到客户端。

使用prop绑定服务器端名称

export class User {

@prop({name:'email\_Address'})

emailAddress: string;

}

export class PropServerComponent implements OnInit {

userFormGroup: FormGroup

constructor(private formBuilder: RxFormBuilder) { }

ngOnInit() {

let user = new User();

let formBuilderConfiguration = new FormBuilderConfiguration();

let userData = {

email\_Address:'bharat.patel@gmail.com',

};

formBuilderConfiguration.dynamicValidation = JSON.parse(JSON.stringify(userData));

this.userFormGroup = this.formBuilder.formGroup(user,formBuilderConfiguration);

}

}

**设置默认值**

当你想设置属性的默认值时，你必须在创建FormGroup的多个组件中设置默认值，而不是使用一个有效的方法来设置默认值，它应该通过decorator在属性上应用。

export class User {

@prop({defaultValue:'bharat.patel@gmail.com'})

emailAddress:string

}

**排除条件属性**

当需要根据特定条件排除属性时，必须在propConfig中添加ignore属性。

export class User {

@prop()

type:string;

@prop({ignore:function(){ return this.type == "Personal";}})

emailAddress:string;

}

**isPrimaryKey**

它被用来定义模型的主键，你需要设置isPrimaryKey属性为true。

export class User {

@prop({ isPrimaryKey: true})

userId: number;

@prop()

firstName: string;

}

**propArray**

https://docs.rxweb.io/decorators/propArray/decorators

propArray用于定义一个类的实例数组。

假设您想要创建一个User表单，其中包含User类中的CourseName等字段，

这是一个类Course的实例数组，如果您想添加@required验证，那么可以使用@propArray()。

**Basic PropArray Decorator**

首先，我们需要创建一个Course类，并在其中创建一个类型为字符串的属性courseName。创建一个User类，并使用@propArray()装饰器在其中创建一个Array类型的课程属性。

export class Address {

@prop() city: string;

@prop() country: string;

}

export class User {

@prop()

emailAddress: string;

@propArray(Address)

addresses: Address[];

}

通过Angular的FormBuilder服务，我们在组件中创建了FormGroup。

export class PropArrayAddComponent implements OnInit {

ngOnInit() {

let user = new User();

user.addresses = new Array<Address>();

let address = new Address();

user.addresses.push(address);

this.userFormGroup = this.formBuilder.formGroup(user);

}

addAddress(){

let addresses = this.userFormGroup.controls.addresses as FormArray;

addresses.push(this.formBuilder.formGroup(Address));

}

onSubmit(){

}

}

**PropArray based on Entity Instance**

如果您有条件地想要propArray实体的实体。

如果contactType属性值等于address，那么address模型属性应该包含在contact的嵌套表单组中，或者属性值包含telephone，那么telephone模型属性应该包含在嵌套表单组中。

export class Address {

@prop() city: string;

@prop() country: string;

}

export class Telephone {

@prop() mobileNumber: number;

}

export class User {

@prop()

contactType: string;

@propArray(undefined, { entityProvider: function () { return this.contactType == "Address" ? Address : Telephone } })

contacts: Telephone[] | Address[]

}

export class PropArrayEntityComponent implements OnInit {

contactType: string = "Address";

userFormGroup: FormGroup;

constructor(private formBuilder: RxFormBuilder) { }

ngOnInit() {

let user = new User();

user.contactType = "Address";

user.contacts = new Array<Address>();

user.contacts = new Array<Telephone>();

this.userFormGroup = this.formBuilder.formGroup(user);

}

addAddress() {

let addresses = this.userFormGroup.controls.contacts as FormArray;

addresses.push(this.formBuilder.formGroup(Address));

}

addTelephone() {

let telephones = this.userFormGroup.controls.contacts as FormArray;

telephones.push(this.formBuilder.formGroup(Telephone));

}

changeValue(value: string) {

this.userFormGroup.reset();

let user = new User();

this.userFormGroup.controls.contactType.setValue(value);

user.contactType = value;

this.contactType = value;

if (this.userFormGroup.controls.contactType.value == "Address") {

user.contacts = new Array<Address>();

let address = new Address();

user.contacts.push(address);

this.userFormGroup = this.formBuilder.formGroup(user);

}

else {

user.contacts = new Array<Telephone>();

let contact = new Telephone();

user.contacts.push(contact);

this.userFormGroup = this.formBuilder.formGroup(user);

}

}

}

**绑定propArray上的服务器端属性名**

如果客户端属性名和服务器属性名不同。然后将propArray的服务器属性名绑定到客户端。

export class Address {

@prop() city: string;

@prop() country: string;

}

export class User {

@prop({name:'email\_address'})

emailAddress: string;

@propArray(Address,{name:"user\_address"})

addresses: Address[];

}

**PropArray基于allowMaxIndex**

当您想要验证在proprArray中添加的索引数时，可以使用它。您需要在allowMaxIndex属性中指定它。

export class Address {

@prop() city: string;

@prop() country: string;

}

export class User {

@prop() emailAddress: string;

@propArray(Address ,{ allowMaxIndex: 2 })

addresses: Address[];

}

**创建BlankFormArray**

它用于创建一个初始化的formArray。您需要指定createBlank值为true。

export class Address {

@prop() city: string;

@prop() country: string;

}

export class User {

@prop() emailAddress: string;

@propArray(Address,{createBlank:true})

addresses: Address[];

}

**propObject**

https://docs.rxweb.io/decorators/propObject/decorators

它用于定义模型类中字段属性的对象。

假设您想要创建一个用户表单，其中包含诸如emailAddress、城市和国家等字段。您需要创建地址类的一个对象，并在用户模型类中访问它的属性。

**Basic propObject decorator**

首先，我们需要创建一个用户模型并在用户类中定义一个emailAddress属性，在用户模型中创建Address类并在其中定义城市和国家属性。在用户类中创建地址类的propoject，实现功能需求。

export class Address {

@prop() city: string;

@prop() country: string;

}

export class User {

@prop() emailAddress: string;

@propObject(Address)

address: Address;

}

通过Angular的FormBuilder服务，我们在组件中创建了FormGroup。

export class PropObjectAddComponent implements OnInit {

userFormGroup: FormGroup

constructor( private formBuilder: RxFormBuilder ) { }

ngOnInit() {

let user = new User();

user.address = new Address();

this.userFormGroup = this.formBuilder.formGroup(user);

}

}

**allOfAsync**

https://docs.rxweb.io/async-validators/allOfAsync/validators  
allOfAsync验证验证器将检查用户是否已输入给定字段的所有值或非异步。

**什么时候使用**假设您想要创建一个雇员表单，其中包含部门、projectDomains、资格、技能和爱好等字段。您希望用户输入特定字段的所有值。在这里，该字段以数组的形式获取，并根据此选择通过异步提供matchValues应用于属性。在这里，根据需求可能会出现这些场景。

在projectDomains字段上应用allOfAsync验证，您希望用户选择matchValues中提到的所有项目域。  
在兴趣爱好字段上显示自定义验证消息。  
使用messageKey属性显示语言字段上的自定义消息。  
基于服务器规则动态应用allOfAsync验证。

让我们看看allOfAsync验证器是如何满足需求的。

**基本allOfAsync验证**通过Angular的FormBuilder服务，我们在组件中创建了FormGroup。  
这里我们介绍了添加表单操作。

@Component({

selector: 'app-allOfAsync-add-validator',

templateUrl: './all-of-async-add.component.html'

})

export class AllOfAsyncAddValidatorComponent implements OnInit {

employeeInfoFormGroup: FormGroup

projectDomains: string[] = [];

projectDomainsArray: string[] = ["ECommerce", "Banking", "Educational", "Gaming"];

constructor(

private formBuilder: FormBuilder, private http: HttpClient) { }

ngOnInit() {

this.employeeInfoFormGroup = this.formBuilder.group({

department: [''],

projectDomains: ['',[], RxwebValidators.allOfAsync({validatorConfig:of({ matchValues: ["ECommerce", "Banking"] })})],

});

}

addProjectDomain(element: any, index: number) {

var indexOf = this.projectDomains.indexOf(element.value);

element.checked ? this.projectDomains.push(element.value) : this.projectDomains.splice(indexOf, 1);

this.employeeInfoFormGroup.controls.projectDomains.setValue(this.projectDomains);

}

}

**ValidatorConfig**

类型:Observable  
Validator配置用于根据服务器规则使用关键字的动态解析属性的值，如下所示  
下面的选项在RxwebValidators.allOfAsync(({validatorConfig:of({<property\_name>:<value>})}))验证器中不是强制使用的。如果需要，那么使用下面的选项。

Option Description

matchValues： matchValues是一个数组，用于验证值的匹配。根据它，数组中的所有值都应该匹配

message： 覆盖全局配置消息并在各自的FormControl上设置自定义错误消息

messageKey： 当用户希望在其某些字段上显示不同的自定义验证消息时，可以使用BaseConfig的messageKey属性。用户可以通过在ReactiveFormConfig中定义一个全局自定义messageKey，并在验证的message属性中设置它。

**matchValues**

Type : any[]

RxwebValidators.allOfAsync({validatorConfig:of({ matchValues: ["ECommerce", "Banking"] })

**message**

Type : string

RxwebValidators.allOfAsync(

{validatorConfig:

of({matchValues:["Drawing", "Singing","Dancing","Travelling"],message: "请选择所有的爱好"}

)

**messageKey**

Type : string

RxwebValidators.allOfAsync(

{validatorConfig:of({ matchValues: ["English", "Spanish", "French"], messageKey: 'allOfAsyncMessageKey' })

**完整的allOfAsync示例**这个完整的allOfAsync示例包含了所有的ArrayConfig属性，它将满足场景1、2、3和4的要求。

ngOnInit() {

this.employeeInfoFormGroup = this.formBuilder.group({

department: [''],

projectDomains: ['', [], RxwebValidators.allOfAsync({ validatorConfig: of({ matchValues: ['ECommerce', 'Banking', 'Educational', 'Gaming'] }) })],

qualifications: ['', [], RxwebValidators.allOfAsync({

validatorConfig: of({

matchValues: ['Secondary', 'Senior Secondary', 'B.Tech.', 'M.Tech.', 'B.C.A.', 'M.C.A.'],

conditionalExpression: (x, y) => x.department == 'DotNet'

})

})],

skills: ['', [], RxwebValidators.allOfAsync({

validatorConfig: of({

matchValues: ['MVC', 'AngularJS', 'Angular 5', 'C#', 'Web Api', 'SQL Server'],

conditionalExpression: 'x => x.department ==\'DotNet\''

})

})],

hobbies: ['', [], RxwebValidators.allOfAsync({

validatorConfig: of({

matchValues: ['Drawing', 'Singing', 'Dancing', 'Travelling', 'Sports'],

message: 'Please select all hobbies'

})

})],

languages: ['', [], RxwebValidators.allOfAsync({

validatorConfig: of({

matchValues: ['English', 'Spanish', 'French'],

messageKey: 'allOfAsyncMessageKey'

})

})]

});

}

**choiceAsync**

<https://docs.rxweb.io/async-validators/choiceAsync/validators>

choiceAsync验证验证器将检查用户输入的值是否与属性中定义的范围异步匹配

**什么时候使用**

假设您想要创建一个雇员表单，其中包含部门、projectDomains、资格、技能和爱好等字段。您希望用户为特定的choiceAsync选择特定范围的值。这里以数组的形式获取字段，并根据此情况，通过异步应用minlength和maxLength将choiceAsync应用于属性。在这里，根据需求可能会出现这些场景。

projectDomains字段，您希望用户在其中输入最少三个域。  
您希望用户最多输入4个条件的条件字段。  
在爱好字段中添加自定义消息。  
使用messageKey属性显示certification字段上的自定义消息。  
基于服务器规则动态应用choiceAsync验证。

languages: ['', [], RxwebValidators.choiceAsync({

validatorConfig: of({

minLength: 2,

conditionalExpression: (x, y) => x.department == 'DotNet'

})

})],

skills: ['', [], RxwebValidators.choiceAsync({

validatorConfig: of({

minLength: 2,

conditionalExpression: 'x => x.department ==\'DotNet\''

})

})],

hobbies: ['', [], RxwebValidators.choiceAsync({

validatorConfig: of({

maxLength: 4,

message: 'Please select upto 4 hobby'

})

})],

certifications: ['', [], RxwebValidators.choiceAsync({

validatorConfig: of({

minLength: 2,

messageKey: 'choiceMessageKey'

})

})],

noneOfAsync  
noneOfAsync验证验证器将检查是否没有从给定输入中异步选择任何值。

<https://docs.rxweb.io/async-validators/noneOfAsync/validators>

**什么时候使用**让我们假设您正在创建一个雇员表单，您希望雇员在其中不输入任何包含部门、爱好和技能等字段的特定值。在这里，该字段以数组的形式获取，并通过异步应用matchvalues将选择应用于属性。在这里，根据需求可能会出现这些场景。

您希望用户根据匹配值在其中输入技能的技能字段。  
应用noneOfAsync验证基于匹配条件的表单，如如果部门是' dotnet '，那么技能值应该基于匹配值。  
爱好上的自定义消息字段。  
使用messageKey属性显示语言字段上的自定义消息。  
基于服务器规则动态应用noneOfAsync验证。

this.employeeInfoFormGroup = this.formBuilder.group({

department: [''],

projectDomains: ['', [], RxwebValidators.noneOfAsync({ validatorConfig: of({ matchValues: ['ECommerce', 'Banking', 'Educational', 'Gaming'] }) })],

qualifications: ['', [], RxwebValidators.noneOfAsync({

validatorConfig: of({

matchValues: ['Secondary', 'Senior Secondary', 'B.Tech', 'M.Tech', 'B.C.A.', 'M.C.A.'],

conditionalExpression: (x, y) => x.department == 'DotNet'

})

})],

skills: ['', [], RxwebValidators.noneOfAsync({

validatorConfig: of({

matchValues: ['MVC', 'AngularJS', 'Angular 5', 'C#', 'Web Api', 'SQL Server'],

conditionalExpression: 'x => x.department ==\'DotNet\''

})

})],

hobbies: ['', [], RxwebValidators.noneOfAsync({

validatorConfig: of({

matchValues: ['Drawing', 'Singing', 'Dancing', 'Travelling', 'Sports'],

message: 'Please do not select any hobby'

})

})],

languages: ['', [], RxwebValidators.noneOfAsync({

validatorConfig: of({

matchValues: ['English', 'Spanish', 'French'],

messageKey: 'noneOfMessageKey'

})

})]

});

**oneOfAsync**  
oneOfAsync验证验证器将检查用户是否异步输入了任何一个给定的输入。

<https://docs.rxweb.io/async-validators/oneOfAsync/validators>

**什么时候使用**让我们假设您正在创建一个雇员表单，您希望雇员在其中输入某个特定值的任何一个值，该值包含部门、爱好和技能等字段。在这里，该字段以数组的形式获取，并根据此，oneOfAsync通过应用matchvalues异步应用于属性。在这里，根据需求可能会出现这些场景。

您希望用户根据匹配值在其中输入技能的技能字段。  
根据表单中的匹配条件应用oneOfAsync验证，比如如果部门是“dotnet”，那么技能值应该基于匹配值。  
爱好上的自定义消息字段。  
使用messageKey属性显示语言字段上的自定义消息。  
基于服务器规则动态应用oneOfAsync验证。

this.employeeInfoFormGroup = this.formBuilder.group({

department: [''],

projectDomains: ['',[],

RxwebValidators.oneOfAsync({ matchValues: ["ECommerce", "Banking", "Educational", "Gaming"] })})],

qualifications: ['',[],

RxwebValidators.oneOfAsync({ matchValues:

["Secondary", "Senior Secondary", "B.Tech", "M.Tech", "B.C.A.", "M.C.A."],

conditionalExpression: (x, y) => x.department == 'DotNet' })})

],

skills: ['',[],

RxwebValidators.oneOfAsync({ matchValues:

["MVC", "AngularJS", "Angular 5", "C#", "Web Api", "SQL Server"],

conditionalExpression: "x => x.department =='DotNet'" })})],

hobbies: ['',[],

RxwebValidators.oneOfAsync({ matchValues:

["Drawing", "Singing", "Dancing", "Travelling", "Sports"],

message: "Please select atleast 1 hobby" })})],

languages: ['',[],

RxwebValidators.oneOfAsync({ matchValues:

["English", "Spanish", "French"],

messageKey: 'oneOfAsyncMessageKey' })})]

});

**patternAsync**patternAsync验证验证器将允许用户异步输入匹配预定义的patternAsync值参数的输入。

<https://docs.rxweb.io/async-validators/patternAsync/validators>

允许输入只包含字母的用户名。  
根据表单中匹配的条件应用patternAsync验证，例如，如果Username是John，那么只有patternAsync验证必须应用到Age值(即:Age)。，年龄字段只能是数字)。  
在Zipcode字段上添加自定义消息。  
使用messageKey属性显示Pin Code字段上的自定义消息。  
基于服务器规则动态应用patternAsync验证。

this.userFormGroup = this.formBuilder.group({

userName: ['', [], RxwebValidators.patternAsync({ validatorConfig: of({ expression: { 'onlyAlpha': /^[A-Za-z]+$/ } }) })],

zipCode: ['', [], RxwebValidators.patternAsync({

validatorConfig: of({

expression: { 'zipCode': /^[0-9]{5}(?:-[0-9]{4})?$/ },

message: 'Zip code should match 12345 or 12345-6789'

})

})],

pinCode: ['', [], RxwebValidators.patternAsync({

validatorConfig: of({

expression: { 'pinCode': /^[1-9][0-9]{5}$/ },

messageKey: 'patternMessageKey'

})

})]

});

通过验证器、验证装饰器或模板表单在表单上应用验证的三种方法

**alpha**

Alpha验证只允许输入字母。它不允许任何数字或特殊字符。

this.countryFormGroup = this.formBuilder.group({

countryName:['', RxwebValidators.alpha()],

});

**alphaNumeric**

alphaNumeric验证只允许输入字母和数字，不允许输入任何特殊字符。

ASCII

ASCII验证装饰器允许用户输入正确的ASCII格式的输入。

## **choice**

选择验证将检查输入的值是否与定义的属性匹配。

this.employeeInfoFormGroup = this.formBuilder.group({

skills:[RxwebValidators.choice({minLength:5})],

});

**compare**

比较验证将比较两个输入，无论它们是否相同。

this.userFormGroup = this.formBuilder.group({

password:['',],

confirmPassword:['', RxwebValidators.compare({fieldName:'password' })],

});

**compose**

组合验证修饰符用于对特定字段应用多个验证。

this.userForm = this.formBuilder.group({

Username:['',[ RxwebValidators.compose({

validators:[

RxwebValidators.required(),

RxwebValidators.alpha()

],]]

});

对比：

userName:['',[RxwebValidators.required(),RxwebValidators.minLength({value:5}),RxwebValidators.maxLength({value:10})]]

})

## **contains**

包含验证将检查输入中的值是否存在，它将不允许输入不包含预定义值的输入。

this.userFormGroup = this.formBuilder.group({

emailAddress:['', RxwebValidators.contains({value:'@gmail.com' })],

});

## **creditCard**

creditCard验证将检查属性值是否为creditCard类型，它不允许输入信用卡格式以外的任何值。

this.userFormGroup = this.formBuilder.group({

cardType:['',],

creditCardNumber:['', RxwebValidators.creditCard({fieldName:'cardType' })],

});

## **dataUri**

dataUri验证允许用户输入一个有效的数据Uri。

this.userFormGroup = this.formBuilder.group({

htmlDataUri:['', RxwebValidators.dataUri()],

});

## **different**

不同的验证将检查两个输入是否不同。它只是比较装饰者的对立面。

this.accountInfoFormGroup = this.formBuilder.group({

firstName:['',],

lastName:['', RxwebValidators.different({fieldName:"firstName" })],

});

## **digit**

数字验证只允许输入数字，不允许输入任何字母或特殊字符。

## **email**

电子邮件验证只允许用户输入正确的电子邮件格式。

## **endsWith**

endsWith验证允许用户输入以特定值结束的输入。

## **even**

甚至验证将检查用户输入的值是否为偶数。

## **extension**

扩展验证允许用户输入正确的扩展格式的输入。

this.userFormGroup = this.formBuilder.group({

Image :['', RxwebValidators.extension({extensions:'png','jpg'})],

});

## **factor**

因子验证将允许用户输入一个称为红利的数字因子。

this.userFormGroup = this.formBuilder.group({

firstNumber:['', RxwebValidators.factor({dividend:50 })],

});

## **file**

文件验证验证器允许用户验证是否可以上传多少文件。它依赖于maxFiles和minFiles。

this.userInfoFormGroup = this.formBuilder.group({

totalDocumentFiles:['', RxwebValidators.file({minFiles:5 })],

});

fileSize

fileSize验证允许用户输入适当的文件大小格式的输入。

greaterThanEqualTo  
大于等于验证装饰器将检查输入属性是否大于或等于相关字段的输入。

this.userFormGroup = this.formBuilder.group({

age:['',],

voterAge:['', RxwebValidators.greaterThanEqualTo({fieldName:'age' })],

});

greaterThan  
大于验证将检查输入属性是否大于相关字段输入。

hexColor  
十六进制颜色验证将允许用户只输入正确的十六进制颜色格式。

IP

IP校验器用于校验设备的IP地址。

## **image**

图像验证验证器允许用户验证像高度，宽度等图像。

Json

验证将允许用户仅以适当的json格式输入输入。

leapYear  
LeapYear验证将检查输入的值是否为闰年。

lessThanEqualTo  
小于等于验证将允许用户只输入小于oe等于预定义字段中的值。

## **lessThan**

小于验证将允许用户只输入小于预定义字段中的值的值。

## **lowercase**

允许用户只输入小写字母。

MAC

MAC验证将检查输入的值是否为有效的MAC地址。

maxDate  
MaxDate验证将允许用户输入小于MaxDate值参数的日期。

this.userFormGroup = this.formBuilder.group({

registrationDate:['', RxwebValidators.maxDate({value:new Date(2018,7,30) })],

});

MaxLength

验证将允许用户输入最大长度值参数的输入。

maxNumber  
MaxNumber验证将允许用户输入最大的数字值参数。

minDate  
最小日期验证将允许用户输入日期大于最小日期值参数。

this.userFormGroup = this.formBuilder.group({

registrationDate:['', RxwebValidators.minDate({value:new Date(2018,7,30) })],

});

MinLength

验证允许用户输入与最小长度值参数匹配的输入长度。

this.contactFormGroup = this.formBuilder.group({

countryName:['', RxwebValidators.minLength({value:8 })],

});

minNumber  
MinNumber验证将允许用户输入大于最小值的输入值参数。

this.resultInfoFormGroup = this.formBuilder.group({

maths:['', RxwebValidators.minNumber({value:35 })],

});

noneOf

noneOf验证将检查用户是否从给定输入中没有选择任何值。

numeric

数值验证将检查输入的值是否为有效的数值。验证可根据需要设置，可以是小数、负数或正数。

Odd

奇数验证将检查输入的值是否为奇数。

oneOf

oneOf验证将检查用户是否输入了任何一个给定的输入。

Password

密码验证将允许用户只输入输入根据正确的密码验证格式。

Pattern

模式验证将允许用户输入匹配预定义的模式值参数的输入

port

端口验证允许用户输入有效的端口号。

primeNumber

验证允许用户只输入质数

Range

范围验证将检查输入的值是否在指定的范围内

Required

验证将检查用户是否在属性中输入了值。

startsWith

startsWith验证允许用户输入以特定值开始的输入。

time

时间验证将允许用户仅以正确的时间格式输入输入。

Unique

唯一验证验证器用于验证基于formArray的唯一输入。

this.employeeFormGroup = this.formBuilder.group({

fullName:[''],

skills:this.formBuilder.array([

this.getSkillFormGroup()

])

});

addSkill(){

let skillsArray = <FormArray>this.employeeFormGroup.controls.skills;

skillsArray.push(this.getSkillFormGroup());

}

getSkillFormGroup(){

return this.formBuilder.group({

skillName:['',RxwebValidators.unique()]

})

}

UpperCase

大写验证将允许用户仅以大写格式输入字母。

url  
Url验证将检查属性中输入的值是否为正确的Url格式。

## **Dynamic Validation**

当验证规则来自服务器端时，使用动态验证，这意味着客户端没有静态代码来管理响应式表单上的验证。场景:First Name字段应该接受字母，并且在服务器端配置。下面是firstName字段的验证规则json。

dynamic-validation.json

{

"firstName":{

"alpha":true

}

}

@Component({

selector: 'app-user-info-add',

templateUrl: './user-info-add.component.html'

})

export class UserInfoAddComponent implements OnInit {

userInfoFormGroup: FormGroup

constructor(

private formBuilder: RxFormBuilder, private http: HttpClient

) { }

ngOnInit() {

this.http.get('assets/dynamic-validation.json').subscribe(

(dynamicValidationConfiguration:any) => {

this.userInfoFormGroup = this.formBuilder.group({

firstName:['John']

},

new FormBuilderConfiguration( {dynamicValidation: dynamicValidationConfiguration}

));

})

}

}

## **Conditional Validation**

根据其他字段的值，有些字段是必需的，比如firstName字段包含'John'的值，那么lastName字段是必需的。参见stackblitz上的工作代码示例

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

import { FormGroup } from "@angular/forms"

import { RxFormBuilder, RxwebValidators } from '@rxweb/reactive-form-validators';

@Component({

selector: 'app-user-info-add',

templateUrl: './user-info-add.component.html'

})

export class UserInfoAddComponent implements OnInit {

userInfoFormGroup: FormGroup

constructor(

private formBuilder: RxFormBuilder

) { }

ngOnInit() {

this.userInfoFormGroup = this.formBuilder.group({

firstName:['',[RxwebValidators.required()]],

lastName:['',[RxwebValidators.required({conditionalExpression:(x)=> x.firstName == 'John'})]]

});

}

}

## **Dirty Check**

检查表单是否脏。在组件端，你必须通过RxFormBuilder创建一个FormGroup对象，然后你可以从FormGroup对象中使用isDirty方法。

<h3 class="bd-title" id="content">required Validator once for all properties</h3>

<br/>

<form [formGroup]="userInfoFormGroup">

<div class="form-group">

<label>First Name</label>

<input type="text" formControlName="firstName" class="form-control" />

<small class="form-text text-danger" >{{userInfoFormGroup.controls.firstName.errorMessage}}<br/></small>

</div>

<div class="form-group">

<label>Last Name</label>

<input type="text" formControlName="lastName" class="form-control" />

<small class="form-text text-danger" >{{userInfoFormGroup.controls.lastName.errorMessage}}<br/></small>

</div>

<button [disabled]="!userInfoFormGroup.isDirty()" class="btn btn-primary">Submit</button>

</form>

## **Post as Form Data of Reactive Form value**

这提供了一个toFormData()方法，它将FormGroup值转换为FormData。下面是一个Post作为响应式表单值的FormData的例子。

this.userFormGroup = this.formBuilder.group({

firstName:[''],

lastName :[''],

userName:[''],

password : ['']

});

addUser(){

let formdata = this.userFormGroup.toFormData()

this.http.post(this.api, formdata); // This is fake uri, This is just for your reference.

}

## **Post Image through form Data**

要从输入创建fileObject，我们必须在HTMLInputFileElement上设置writeFile="true"属性。下面是一个通过formData Post Image的例子

this.userInfoFormGroup = this.formBuilder.group({

profilePhoto:[''],

});

addUser(){

let formdata = (<FormGroupExtension>this.userInfoFormGroup).toFormData()

this.http.post(this.api, formdata); // This is fake uri, This is just for your reference.

}

<input type="file" [writeFile]="true" formControlName="profilePhoto" class="form-control" multiple />

## **Reset Form**

RxFormBuilder为角响应式表单对象提供了一个重置表单的解决方案。如果你想在创建formControls实例时按照初始化的值重置表单，你可以使用FormGroupExtension的resetForm()方法。

this.userFormGroup = this.formBuilder.group({

firstName:[''],

lastName:[''],

userName:[''],

password:['']

});

resetForm(){

(<FormGroupExtension>this.userFormGroup).resetForm();

}

## **Compare Password**

比较验证用于检查两个formControls的值是否相同。下面是一个比较密码的示例

this.userFormGroup = this.formBuilder.group({

password:['',],

confirmPassword:['', RxwebValidators.compare({fieldName:'password' })],

});

## **Single Error Message**

你可以为formControl设置一个默认消息，并在没有多重条件stackblitz的情况下将其显示在单个错误消息中

<input type="text" formControlName="userName" class="form-control" />

{{userFormGroup.controls.userName["errorMessage"]}}

在ReactiveFormConfig中设置全局验证消息

"validationMessage":{

"required":"This field is required",

"minLength":"minimum length is {{0}}",

"maxLength":"allowed max length is {{0}}"

}

});

this.userFormGroup = this.formBuilder.group({

userName:['',[RxwebValidators.required(),RxwebValidators.minLength({value:5}),RxwebValidators.maxLength({value:10})]]

})