**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[];

}