使用RxWebValidators创建优雅的响应式表单

<https://indepth.dev/posts/1310/creating-elegant-reactive-forms-with-rxwebvalidators>

在本文中，我们将重点关注如何使复杂的响应式表单验证代码更具可读性、可扩展性和可维护性。  
Angular响应式表单方法最大的好处就是从模板中删除了核心验证逻辑。它允许我们以受控的方式集中在一个地方(组件)进行表单验证。但是当我们处理复杂的表单时，我们的代码变得笨拙或性能下降。

**如何?**

根据Angular的交叉控制验证指南，我们必须在FormGroup级别上设置自定义验证器。如果表单有太多的FormControls，它可能会创建一个性能问题，因为自定义验证器会在每个字段值更改时触发。

响应式表单api很酷，但是在条件验证场景中，要更干净地管理代码是很复杂的。例如，我们希望根据其他FormControl值的变化来要求我们的字段。为了实现这一点，我们必须订阅依赖的FormControl(notification)的值变化，并使用相应条件要求的FormControl(phone)的setValidators方法。这是一种幼稚且有问题的方法，因为在将有条件要求的验证器分配给FormControl时，很有可能会错过最初应用的验证器。

让我们学习RxWebValidators在响应式表单中跨字段验证、条件验证和按需验证的优雅解决方案。  
为了使用RxWebValidators，我们必须在项目中安装这个包(@rxweb/reactive-form-validators)。下面是安装命令:

yarn add @rxweb/reactive-form-validators

安装完成后，在应用程序的根模块中注册RxReactiveFormsModule

import { RxReactiveFormsModule } from "@rxweb/reactive-form-validators"

@NgModule({

imports: [ RxReactiveFormsModule,...],

declarations: [...],

})

export class AppModule { }

让我们来理解一下用例以及标准和RxWebValidators方法之间的区别。

**交叉领域的验证**

用例:我们想要比较两个FormControl的值，如果两个控件的值不一样，就把FormControl标记为无效。

**1 - 标准方法**

const userForm = new FormGroup({

'firstName': new FormControl(),

'password': new FormControl(),

'comparePassword': new FormControl()

},

{ validators: (control: FormGroup): ValidationErrors | null => {

const password = control.get('password');

const comparePassword = control.get('comparePassword');

return password && comparePassword && password.value !== comparePassword.value ?

{ 'compare': true } : null; }

});

根据上面显示的代码，我们已经在FormGroup级别上添加了自定义验证器;验证器也会调用非验证的FormControl，比如firstName FormControl。有一些其他的方法可以解决这个问题:

1 、 根据Deborah K建议的例子，我们可以创建一个单独的FormGroup，它包括'password'和'comparePassword' FormControl。使用这种方法，我们可能必须在将表单值对象传递给服务器时重新构造它。这是应用程序的位代码开销，维护一致性具有挑战性。

1. 另一种方法是将相同的验证器放在两个FormControl级别，而不是FormGroup级别。确认器只在指定的FormControl值变化时触发。这看起来很好，但是如果有多个交叉控件，那么管理代码就很困难。

**2 - 用RxWebValidators来尝试同样的情况**。

let userForm =new FormGroup({

password:new FormControl(),

confirmPassword:new FormControl('', RxwebValidators.compare({fieldName:'password' })),

});

根据上面的代码，我们已经在FormControl的“comparePassword”上使用了比较验证器。当FormControl的'password'或'comparePassword'的值改变时，验证器就会触发。使用这种方法，我们的代码更具可读性。

**有条件的验证**  
用例:一旦通知复选框勾选，“phone”字段是必需的。  
**1 - 标准方法：**

setNotification(ticked: boolean): void {

const phoneControl = this.userForm.get('phone');

if (ticked) {

phoneControl.setValidators(Validators.required);

} else {

phoneControl.clearValidators();

}

phoneControl.updateValueAndValidity();

}

当表单变大并涉及到复杂的验证规则时，可以根据上面显示的代码应用下面的高亮显示。  
1 关注代码可维护性，因为跟踪FormControl上有多少验证规则很有挑战性。  
2 如果我们错过了任何验证器来在运行时重新分配相同的值，就会有不一致行为的公平机会。  
3 通过放置多个if/else子句来覆盖许多条件验证案例，可能会产生“代码味道”。  
让我们参考下面的代码，它从表面上解决了上面提到的所有问题。

let userForm = new FormGroup({

notification:new FormControl(),

phone:new FormControl('',

RxwebValidators.required({conditionalExpression:x => x.notification === true }))

});

通过使用conditionalExpression， 'phone' FormControl只有在通知FormControl的值为'true'时才需要。

**基于价值的按需验证**用例:有三个领域的优质产品收费，购买价格和转售价格。转售价格必须至少比购买价加优质产品收费高出30%:

**1 标准方法**

ngOnInit() {

this.userInfoFormGroup = new FormGroup({

premiumCharge: new FormControl(),

purchasePrice: new FormControl(),

resalePrice: new FormControl()

});

this.userInfoFormGroup.controls.premiumCharge.valueChanges.subscribe(

t => {

this.**setMinValidator**(this.userInfoFormGroup.value);

});

this.userInfoFormGroup.controls.purchasePrice.valueChanges.subscribe(

t => {

this.**setMinValidator**(this.userInfoFormGroup.value);

});

}

**setMinValidator**(formValue: any) {

const minimumPrice =

((parseInt(formValue.purchasePrice) + parseInt(formValue.premiumCharge)) \* 30 / 100);

this.userInfoFormGroup.controls.resalePrice.clearValidators();

this.userInfoFormGroup.controls.resalePrice.setValidators(Validators.min(minimumPrice));

this.userInfoFormGroup.controls.resalePrice.updateValueAndValidity({ onlySelf: true });

}

管理这样的情况就是在组件中编写太多的代码，这使得我们的代码在某些时候扩展性较差。  
让我们用RxWebValidators优雅地转换这些代码。

this.userInfoFormGroup = new FormGroup({

premiumCharge:new FormControl(),

purchasePrice:new FormControl(),

resalePrice: new FormControl('', RxwebValidators.minNumber({

dynamicConfig: (x, y) => {

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

return { value: minimumPrice };

}

}))

});

有了这种方法，我们就有了扩展代码的全面范围。我们不需要订阅每个FormControl的值变化，因为库会自动处理依赖验证业务计算FormControls。