# aurelia基础知识--validation

这篇文章介绍了Aurelia的验证插件的基本验证。您将了解如何使用流畅的规则API向您的应用程序添加验证，以及对模板进行最小的更改。

你需要安装aurelia-validation，之后在main.js中配置.plugin('aurelia-validation')。如果你使用了aurelia-cli，在你通过npm安装到包中后，添加以下的配置到你的aurelia.json。

{

"name": "aurelia-validation",

"path": "../node\_modules/aurelia-validation/dist/amd",

"main": "aurelia-validation"

}

如果你不知道该把它放在哪里，搜索一下你的aurelia.json。用于aurelia-templating-resources，并将其放在下面。

## Defining Rules

Aurelia验证的标准规则引擎使用一种流畅的语法来定义一组规则。语法有五个部分：

1. 用‘.ensure’搜索一个属性
2. 用‘.required’‘.matches’关联规则和属性
3. 自定义属性使用‘.withMessage’‘.when’。
4. 使用‘.then’进行规则排序
5. 使用‘.on’应用一个规则集合类或实例。

使用一个ValidationRules类定义一个规则集合。以一个属性作为目标，使用ValidationRules.ensure(...)。ensure方法接收一个代表属性名作为参数。参数可以是字符串或简单的属性访问表达式。

### **displayName**

当你使用ensure去获取目标属性时，你可以使用.displayName(name: string|ValidationDisplayNameAccessor)定义一个属性的display name。这个名字用于验证器的message。指定display name是可选的。如果你不设置，验证引擎将会试图通过通过upper-case 方式分割你的属性名来计算display name。

例如firstName的display name会是First Name。

ValidationRules

.ensure('ssn').displayName('SSN')...

### **Applying Rules**

通过ensure方法获取目标属性，可选地设置他的display name 你可以通过构造规则方法来建立规则和属性的关联。

* required() 非空验证。
* matches(regex) 验证属性匹配指定的正则表达式。
* email() 验证电子邮件地址。
* minLength(length)、maxLength(length) 验证字符串属性的长度。
* minItems(count)、maxItems(count) 验证数组长度。
* equals(expectedValue) 验证是否匹配期望的值。
* satisfies((value: any, object?: any) => boolean|Promise<boolean>) 验证提供的函数返回的 是true或者一个Promise的resolves是true。
* 这个函数将会带有两个参数调用:
  + 属性的当前值
  + 这个属性属于的对象

### **withMessage**

所有的规则都有一个在具体的基础使用中可以使用.withMessage(message)被重写的消息。message参数是一个字符串，它将被解释为一个插值绑定表达式，并在验证错误发生时对已验证的对象进行评估。插值绑定表达式可以访问任何对象的属性以及下面所列的上下文属性：

* $displayName: 属性的display name
* $propertyName: 属性名
* $value: 属性值（当验证规则执行后的）
* $object: 拥有该属性的对象
* $config: 包含规则设置的对象。 例如一个有minLength的规则的配置将会有一个length属性
* $getDisplayName: 传入一个属性名，返回一个display name。用大写字母分开，第一字母确保大写。

例如：

ValidationRules

.ensure('ssn').displayName('SSN')

.required().withMessage(`\${$displayName} cannot be blank.`);

.matches(/\d{3}-\d{2}-\d{4}/).withMessage(`"\${$value}" is not a valid \${$displayName}.`);

### **withMessageKey**

另一个复写消息的方法是使用.withMessageKey(key)。key是一个在validationMessages词典中代表key的字符串。你可以像如下的方式一样添加新的keys：

import {validationMessages} from 'aurelia-validation';

validationMessages['invalidAirportCode'] = `\${$displayName} is not a valid airport code.`;

ValidationRules

.ensure('airportCode')

.matches(/^[A-Z]{3}$/).withMessageKey('invalidAirportCode')...

### **Conditional Validation**

你可能会遇到这样的情况你只需要在满足特定条件的情况下对规则进行评估。你可以使用.when(condition: (object) => boolean)方法去定义规则在评估之前必须满足的条件。when接收一个参数，一个包含了该对象的方法，并返回一个指示规则是否需要被评估的值（true表示规则需要被评估）。

ValidationRules

.ensure('email')

.email()

.required()

.when(order => order.shipmentNotifications)

.withMessage('Email is required when shipment notifications have been requested.');

### **Sequencing Rule Evaluation**

规则评估是并行的。我们可以使用.then()方法去延时一个规则评估直到在ensure之前的规则被评估之后。

ValidationRules

.ensure('email')

.email()

.required()

.then()

.satisfiesRule('emailNotAlreadyRegistered')

.ensure('username')

.required()

.minLength(3)

.maxLength(50)

.then()

.satisfiesRule('usernameNotInUse');

上面的例子中，'emailNotAlreadyRegistered'自定义规则只有当email属性通过required和email验证后，才会被评估。同样的，usernameNotInUse将会在required、minLength、maxLength验证通过后才会被评估。

### **Tagging Rules**

使用.tag(tag: string)方法通过名称去标记一个指定属性的规则。你可以用一个指定的标签检索规则，通过使用

let someRules = ValidationRules.taggedRules(rules, tag)

为了得到没有标签的规则，你可以使用

let someRules = ValidationRules.untaggedRules(rules)

当您想要执行特定的规则或规则子集时，这一点很方便。

下面的ValidationController的文档展示了如何验证指定的对象\属性\规则。你还可以使用验证器API的规则子集。

一旦你的规则集被定义了，你可以通过使用.on方法把他们应用到一个类上。这将确保验证引擎在评估特定对象时能够定位规则。

export class Person {

firstName = '';

lastName = '';

}

ValidationRules

.ensure('firstName').required()

.ensure('lastName').required()

.on(Person);

.on也可以把规则应用到一个简单的js对象上。

let patient = {

firstName: '',

lastName: ''

};

ValidationRules

.ensure('firstName').required()

.ensure('lastName').required()

.on(patient);

我们并没有必要将规则直接应用到对象或类中，你可以使用.rule来捕获变量或属性中的规则集。

const personRules = ValidationRules

.ensure('firstName').required()

.ensure('lastName').required()

.rules;

## Customizing Messages

前面的部分展示了如何定制一个单独的属性规则的消息。您可以通过在validationMessages字典中替换规则的缺省消息来覆盖整个系统。

import {validationMessages} from 'aurelia-validation';

validationMessages['required'] = `\${$displayName} is missing!`;

你可以复写ValidationMessageProvider的

getMessage(key: string): Expression方法来使用更加动态的消息逻辑。

import {ValidationMessageProvider} from 'aurelia-validation';

ValidationMessageProvider.prototype.standardGetMessage = ValidationMessageProvider.prototype.getMessage;

ValidationMessageProvider.prototype.getMessage = function(key) {

const translation = i18next.t(key);

return this.parser.parse(translation);

};

你也可以复写

ValidationMessageProvider的

getDisplayName(propertyName: string, displayName: string): string方法

import {ValidationMessageProvider} from 'aurelia-validation';

ValidationMessageProvider.prototype.getDisplayName = function(propertyName, displayName) {

if (displayName !== null && displayName !== undefined) {

return displayName;

}

return i18next.t(propertyName);

};

## Validation Controller

ValidationController通过对各种触发器的响应来协调验证属性的UI过程，并通过渲染器显示验证错误。通常，每个‘form’的视图模型都有一个验证控制器的实例。你可能有多个这取决于你的用例。

Creating a Controller

你可以使用NewInstance来创建验证控制器。

import {inject, NewInstance} from 'aurelia-dependency-injection';

import {ValidationController} from 'aurelia-validation';

@inject(NewInstance.of(ValidationController))

export class RegistrationForm {

controller = null;

constructor(controller) {

this.controller = controller;

}

}

或者使用ValidationControllerFactory

import {ValidationControllerFactory} from 'aurelia-validation';

@inject(ValidationControllerFactory)

export class RegistrationForm {

controller = null;

constructor(controllerFactory) {

this.controller = controllerFactory.createForCurrentScope();

}

}

这两种技术都创建了一个ValidationController的新实例，并将实例注册到组件的容器中，使验证库中的其他组件能够访问approriate控制器实例，而不需要大量的样板代码或标记。

如果你想显式地用视图模型和绑定连接控制器，或者你想在你的控件里使用多个控制器，你需要使用Factory解析器或者ValidationControllerFactory的create方法来创建控制器实例。使用这些方法将不会自动在容器中注册控制器实例，这将阻止带有绑定和渲染器的控制器的自动连接，并强制您在绑定中指定控制器实例，并手动将渲染器添加到控制器。

### **Setting the Validate Trigger**

当你设置了一个控制器，可以设置他的validationTrigger到blur,change,changeOrBlur或者manual。默认是blur，这意味着验证器控制器将会在绑定的元素失去焦点的时候，验证该属性。如果触发器是change绑定对模型属性的每一个更改都将触发该属性的验证。

使用throttle、debounce、updateTrigger绑定行为结合change验证触发器来自定义行为。

使用manual触发器来指示控制器不应该自动验证绑定中使用的属性。错误只会在当调用控制器的validate方法时显示出来，在调用控制器的reset方法时，清除。

import {ValidationControllerFactory, validateTrigger} from 'aurelia-validation';

@inject(ValidationControllerFactory)

export class RegistrationForm {

controller = null;

constructor(validationControllerFactory) {

this.controller = validationControllerFactory.createForCurrentScope();

this.controller.validateTrigger = validateTrigger.manual;

}

}

### **validate & reset**

你可以使用validate()方法强制验证器控制器进行验证。验证将会进行验证，渲染结果，返回一个使用ControllerValidateResult 实例解析的Promise。

只有当出现意外的应用程序错误时，Promise才会reject。一定要捕捉这些reject，就想你会遇到意外的应用程序错误时一样。

使用无参数调用验证方法将验证与控制器注册的所有绑定和对象。你可以提供一个验证指令，以将验证限制为特定对象、属性和规则集。

controller.validate();

controller.validate({ object: person });

controller.validate({ object: person, rules: myRules });

controller.validate({ object: person, propertyName: 'firstName' });

controller.validate({ object: person, propertyName: 'firstName', rules: myRules });

controller.validate()

.then(result => {

if (result.valid) {

// validation succeeded

} else {

// validation failed

}

});

大多数时候，理会使用ControllerValidateResult 的valid属性来确定验证是通过还是失败。使用resultes属性来访问被controller.validate(...)评估的规则的ValidateResult。每个ValidateResult有他自己的rule和valid属性，他们会高数你特定的规则是通过还是失败，并包含着message，object和propertyName属性。

与validate相对的是reset方法。调用不带参数地调用reset方法将会取消之前的渲染的验证结果。你可以提供一个重置指令来将重置限制为一个特定的对象或属性。

controller.reset();

controller.reset({ object: person });

controller.reset({ object: person, propertyName: 'firstName' });

### **addError & removeError**

你也许会需要呈现从其他来源的验证错误。也许在尝试保存更改时，服务器返回了一个业务规则错误。你可以使用控制器的addError(message: string, object: any, propertyName?: string):ValidateResult方法来显示服务错误。这个方法返回一个实例，这个实例可以通过使用removeError(result: ValidateResult)取消验证的错误。

### **addRenderer & removeRenderer**

验证控制器渲染错误通过发送他们到ValidationRenderer接口的实现。该库使用内置的渲染器，将错误“呈现”给数组属性以进行数据绑定/模板化。这在下面的displaying errors 中将会呈现。你可以创建自定义的渲染器，通过addRenderer(renderer)把它添加到控制器的集合中去。

### **Events**

验证控制器有一个subscribe(callback: (event: ValidateEvent) => void)方法。你可以使用它去订阅验证和重置事件。当控制器的验证和重置方法被调用时，回调将会被传递一个通过一个，包含可以用来确定整体有效性状态的属性，以及验证或重置调用的结果的ValidateEvent实例。

## Validator

validator是一个ValidationController后台验证对象和属性的一个接口。aurelia-validation插件包提供了一个该接口的实现StandardValidator。他可以知道如何去评估由aurelia-validation的API创建的评估规则。当你使用Validator直接去验证一个特定的对象或属性，他不会有UI上的副作用，验证结果不会被验证器渲染器去呈现。

import {inject} from 'aurelia-dependency-injection';

import {Validator} from 'aurelia-validation';

@inject(Validator)

export class RegistrationForm {

validator = null;

constructor(validator) {

this.validator = validator;

}

}

使用Validator 实例的validateObject和validateProperty方法去验证不会有任何渲染的副作用。这些方法会返回一个带有一个ValidationResults数组的Promise。

## Validate Binding Behavior

validate绑定行为支持双向数据绑定的快速和简单的验证。该行为向控制器注册绑定实例，使系统能够在验证触发器发生时验证绑定的相关属性。绑定行为能够识别出在各种绑定表达式中验证的对象和属性名。

<input type="text" value.bind="firstName & validate">

<input type="text" value.bind="person.firstName & validate">

<input type="text" value.bind="person['firstName'] | upperCase & validate">

<input type="text" value.bind="currentEntity[p] & debounce & validate">

validate接受一些可选的参数使你能显式地指定规则和控制器实例。

<input type="text" value.bind="firstName & validate:personController">

<input type="text" value.bind="firstName & validate:personRules">

<input type="text" value.bind="firstName & validate:personController:personRules">

validate绑定行为遵从关联控制器的validateTrigger。如果您想在特定的绑定中使用不同的validate触发器，请使用以下绑定行为之一:

* & validateOnBlur: DOM的blur事件触发验证。
* & validateOnChange: 数据输入导致的model的改变触发验证。
* & validateOnChangeOrBlur: DOM的blur事件或数据输入触发验证。
* & validateManually: 关联的元素被用户失去焦点或改变时绑定不会自动校验。

## Displaying Errors

控制器暴露出了UI通过标准aurelia模板技术创建error的属性。

* results: 当前ValidateResult实例的数组。这些是验证个人规则的结果
* errors: 当前valid属性时false的ValidateResult实例的数组。
* validating: 一个布尔值，表示控制器是否正在执行验证。

假设你的视图模型有一个控制器属性，你可以使用一个重复的方法向你的表单添加一个简单的错误总结:

<form>

<ul if.bind="controller.errors">

<li repeat.for="error of controller.errors">

${error.message}

</li>

</ul>

...

...

</form>

为了构建更复杂的错误ui，您可能需要一个特定绑定或绑定集合的错误列表。validation-errors自定义属性创建一个包含所有validation-errors出现的元素以及他的派生元素的相关的验证错误数组。

<form>

<div class="form-group" validation-errors.bind="firstNameErrors"

class.bind="firstNameErrors.length ? 'has-error' : ''">

<label for="firstName">First Name</label>

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

placeholder="First Name"

value.bind="firstName & validate">

<span class="help-block" repeat.for="errorInfo of firstNameErrors">

${errorInfo.error.message}

</span>

</div>

<div class="form-group" validation-errors.bind="lastNameErrors"

class.bind="lastNameErrors.length ? 'has-error' : ''">

<label for="lastName">Last Name</label>

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

placeholder="Last Name"

value.bind="lastName & validate">

<span class="help-block" repeat.for="errorInfo of lastNameErrors">

${errorInfo.error.message}

</span>

</div>

</form>

第一个表单组div使用validation-errors自定义属性创建firstNameError属性。当数组中有项时，bootstrap的has-error样式被应用到表单组div中。每个错误都会在help-block的span下显示。lastName字段使用同样的方法。

validation-errors自定义属性实现了ValidationRenderer接口。与其直接进行DOM操作来显示错误，它还“将”错误“呈现”给数组属性，以启用上面所示的数据绑定和模板场景。当他的bind生命周期事件调用的时候，他会使用addRenderer自动地把自己加到控制器里，当他的unbind生命周期调用的时候，他会使用removeRenderer方法把自己移除出控制器。

## Custom Renderers

上一节中描述的模板方法可能需要比您希望在模板中包含的更多的标记。如果您希望使用直接DOM操作来呈现验证错误，那么您可以实现一个自定义的渲染器。

自定义渲染器实现一个方法的接口：

render(instruction: RenderInstruction)，RenderInstruction是一个有两个参数的对象:呈现的结果合未呈现的结果。

import {ValidationRenderer,RenderInstruction,ValidateResult} from 'aurelia-validation';

export class BootstrapFormRenderer {

render(instruction: RenderInstruction) {

for (let { result, elements } of instruction.unrender) {

for (let element of elements) {

this.remove(element, result);

}

}

for (let { result, elements } of instruction.render) {

for (let element of elements) {

this.add(element, result);

}

}

}

add(element: Element, result: ValidateResult) {

if (result.valid) {

return;

}

const formGroup = element.closest('.form-group');

if (!formGroup) {

return;

}

// add the has-error class to the enclosing form-group div

formGroup.classList.add('has-error');

// add help-block

const message = document.createElement('span');

message.className = 'help-block validation-message';

message.textContent = result.message;

message.id = `validation-message-${result.id}`;

formGroup.appendChild(message);

}

remove(element: Element, result: ValidateResult) {

if (result.valid) {

return;

}

const formGroup = element.closest('.form-group');

if (!formGroup) {

return;

}

// remove help-block

const message = formGroup.querySelector(`#validation-message-${result.id}`);

if (message) {

formGroup.removeChild(message);

// remove the has-error class from the enclosing form-group div

if (formGroup.querySelectorAll('.help-block.validation-message').length === 0) {

formGroup.classList.remove('has-error');

}

}

}

}

为了使用自定义渲染器，你需要实例化他，并且通过addRenderer方法把它传递给你的控制器。控制器的任何现有错误都会立即被渲染。你可以通过使用removeRenderer方法移除一个渲染器。删除渲染器将不会渲染渲染器先前渲染的任何错误。如果你在activate或者bind方法中使用addRenderer，你要确保在deactivate或unbind方法中调用removeRenderer。

## Entity Validation

到目前为止的示例中展示了控制器使用 & validate绑定来验证指定的属性。控制器还可以验证整个实体，即使实体的某些属性并没有在数据绑定中使用。我们使用控制器的addObject(object, rules?)方法来选择实体的样式验证。调用addObject会添加一个指定的对象到控制器的validate方法调用时会验证的对象集合中。rules参数是可选的。当对象的规则没有使用.on方法指定时使用它。你可以使用removeObject(object)将控制器要被验证的对象列表中移除对象。

您可能有与单个属性没有关联的规则。在规则语法中，你可以使用ensureObject()去定义整个对象的规则。

export class Shipment {

length = 0;

width = 0;

height = 0;

}

ValidationRules

.ensureObject()

.satisfies(obj => obj.length \* obj.width \* obj.height <= 50)

.withMessage('Volume cannot be greater than 50 cubic centemeters.')

.on(Shipment);

## Custom Rules

API中的satisfies方法可以快速地自定义规则。如果您有一个需要多次使用的自定义规则，您可以使用customRule方法来定义它。一旦定义了，你就可以使用satisfiesRule应用这个规则。

ValidationRules.customRule(

'date',

(value, obj) => value === null || value === undefined || value instanceof Date,

`\${$displayName} must be a Date.`

);

ValidationRules

.ensure('startDate')

.required()

.satisfiesRule('date');

你通常需要将参数传递给您的自定义规则。下面是一个“整数范围”规则的示例，该规则接受“min”和“max”参数。注意到customRule方法的最后一个参数将预期的参数打包成一个“配置”对象。当出现错误时，将使用配置对象来计算验证消息，使消息表达式能够访问规则的配置。

ValidationRules.customRule(

'integerRange',

(value, obj, min, max) => value === null || value === undefined

|| Number.isInteger(value) && value >= min && value <= max,

`\${$displayName} must be an integer between \${$config.min} and \${$config.max}.`,

(min, max) => ({ min, max })

);

ValidationRules

.ensure('volume')

.required()

.satisfiesRule('integerRange', 1, 5000);

你可能已经注意到上面的自定义规则会考虑null和undefined是否为有效。规则应该遵循单一责任原则和验证唯一约束。.require()规则的工作是验证数据是否被填充，你不应该添加required检查逻辑到你的自定义规则中，这有两个原因：

规则重用——如果我们的“integerRange”规则也做“required”的检查，我们不能在可选的字段上使用它。

消息相关性——如果我们的“integerRange”规则也进行“required”检查，用户将得到“range”错误消息，当我们应该得到“需要”的错误消息时。

当您编写一个自定义规则时，当规则是“satisfied”/“valid”时返回true，当规则被“broken”/“invalid”时，函数应该返回false。

你还可以一个可以解析为true或false的Promise。这个Promise除非遇到意外的异常，否则不应该被reject。Promise的reject不被用于控制流或表示‘invalid’状态。

一个常用的应用自定义规则是确认两个密码实体的输入是否匹配。下面有个例子：

ValidationRules.customRule(

'matchesProperty',

(value, obj, otherPropertyName) =>

value === null

|| value === undefined

|| value === ''

|| obj[otherPropertyName] === null

|| obj[otherPropertyName] === undefined

|| obj[otherPropertyName] === ''

|| value === obj[otherPropertyName],

'${$displayName} must match ${$getDisplayName($config.otherPropertyName)}', otherPropertyName => ({ otherPropertyName })

);

ValidationRules

.ensure(a => a.password)

.required()

.ensure(a => a.confirmPassword)

.required()

.satisfiesRule('matchesProperty', 'password');

<div class="form-group">

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

<input type="password" class="form-control" id="password" placeholder="Password"

value.bind="password & validate"

change.delegate="confirmPassword = ''">

</div>

<div class="form-group">

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

<input type="password" class="form-control" id="confirmPassword" placeholder="Confirm Password"

value.bind="confirmPassword & validate">

</div>

## Multiple Validation Controllers

如果有两种需要独立验证的表单，当然建议在单独的组件中实现它们。但是，从技术上讲，通过创建多个验证控制器，可以在同一个组件中执行两个或多个独立验证。

假设你正在为一家意大利餐厅制作一份订单。顾客可以点披萨或意大利面。如果顾客在订购披萨时犯了错误，他们应该可以点一份意大利面。因此你的视图模型应该如下：

import {inject, NewInstance} from 'aurelia-framework';

import {ValidationController, ValidationRules} from 'aurelia-validation';

@inject(NewInstance.of(ValidationController), NewInstance.of(ValidationController))

export class ItalianRestaurant {

pizza;

pasta;

pizzaValidationController;

pastaValidationController

constructor(pizzaValidationController, pastaValidationController) {

this.pizzaValidationController = pizzaValidationController;

this.pastaValidationController = pastaValidationController;

const pizzaRules = ValidationRules

.ensure((res: ItalianRestaurant) => res.pizza).required()

.rules;

this.pizzaValidationController.addObject(this, pizzaRules);

const pastaRules = ValidationRules

.ensure((res: ItalianRestaurant) => res.pasta).required()

.rules;

this.pastaValidationController.addObject(this, pastaRules);

}

orderPizza() {

this.pizzaValidationController.validate()

.then(result => {

if (result.valid) {

alert("Ordering this pizza: " + this.pizza);

}

});

}

orderPasta() {

this.pastaValidationController.validate()

.then(result => {

if (result.valid) {

alert("Ordering this pasta: " + this.pasta);

}

});

}

}

<template>

<form validation-errors="errors.bind: pizzaErrors; controller.bind: pizzaValidationController"

submit.delegate="orderPizza()">

<label for="pizza">Choose a pizza:</label>

<input id="pizza" value.bind="pizza & validateManually:pizzaValidationController"/>

<span class="help-block" repeat.for="errorInfo of pizzaErrors">

${errorInfo.error.message}

</span>

<input type="submit" value="Order pizza!"/>

</form>

<form validation-errors="errors.bind: pastaErrors; controller.bind: pastaValidationController"

submit.delegate="orderPasta()">

<label for="pasta">Choose a pasta:</label>

<input id="pasta" value.bind="pasta & validateManually:pastaValidationController"/>

<span class="help-block" repeat.for="errorInfo of pastaErrors">

${errorInfo.error.message}

</span>

<input type="submit" value="Order pasta!"/>

</form>

</template>

上面的表单中，你可以看出，每个validation-errors参数以及validateManually绑定行为都被绑定到了合适的验证器控制器上。这需要每次都指定，因为默认的参数和绑定行为需要请求容器获取一个ValidationController实例，不知道去获取哪一个。

## Integration With Other Libraries

在aurelia-validation中，对象和属性验证工作是由一个Validator接口的实现类StandardValidator类来完成。StandardValidator是负责使用aurelia-validation语法创建的规则应用。如果您有自己的自定义验证引擎，或者您正在使用像Breeze那样的客户端数据管理库，那么您可能不需要这些机器，因为它有自己的验证逻辑。你可以在插件被安装好的时候替代他。

import {ValidateResult} from 'aurelia-validation';

export class BreezeValidator {

validateObject(object) {

if (object.entityAspect.validateEntity()) {

return [];

}

return object.entityAspect

.getValidationErrors()

.map(({ errorMessage, propertyName, key }) => new ValidateResult(key, object, propertyName, false, errorMessage));

}

validateProperty(object, propertyName) {

if (object.entityAspect.validateProperty(propertyName)) {

return [];

}

return object.entityAspect

.getValidationErrors(propertyName)

.map(({ errorMessage, propertyName, key }) => new ValidateResult(key, object, propertyName, false, errorMessage));

}

}

import {BreezeValidator} from './breeze-validator';

export function configure(aurelia) {

aurelia.use

.standardConfiguration()

.plugin('aurelia-validation', config => config.customValidator(BreezeValidator))

...

...

}

## Integration With Other Libraries

aurelia-i18n是aurelia的官方I18N包。将国际化插件和验证器结合是很容易的。所有标准的验证器消息都提供了一个ValidationMessageProvider类。通过翻译消息，复写getMessage(key)和getDisplayName(propertyName，displayName)方法带有国际化实现去获取消息和属性名翻译的版本。

import {I18N} from 'aurelia-i18n';

import {ValidationMessageProvider} from 'aurelia-validation';

export function configure(aurelia) {

...

...

ValidationMessageProvider.prototype.getMessage = function(key) {

const i18n = aurelia.container.get(I18N);

const translation = i18n.tr(`errorMessages.${key}`);

return this.parser.parse(translation);

};

ValidationMessageProvider.prototype.getDisplayName = function(propertyName, displayName) {

if (displayName !== null && displayName !== undefined) {

return displayName;

}

const i18n = aurelia.container.get(I18N);

return i18n.tr(propertyName);

};

...

...

}

### **Creating the view and view-model**

一旦你重载ValidationMessageProvider你准备必要的方法创建一个视图和视图模型。这里是一个简单的多语言表单的视图，它具有第一个和最后一个名称字段。验证发生在表单提交上，而更换语言按钮演示了i18n的功能。

<template>

<form>

<label>${'firstName' & t}: <br />

<input type="text" value.bind="firstName & validate" />

</label>

<label>${'lastName' & t}: <br />

<input type="text" value.bind="lastName & validate" />

</label>

<button click.delegate="submit()">${'submit' & t}</button>

</form>

<div>

<h3>${'latestValidationResult' & t}: ${message}</h3>

<ul if.bind="controller.errors.length">

<li repeat.for="error of controller.errors">

${error.message}

</li>

</ul>

</div>

<button click.trigger="switchLanguage()">${'switchLanguage' & t}</button>

<template>

import {inject, NewInstance} from 'aurelia-framework';

import {I18N} from 'aurelia-i18n';

import {ValidationRules, ValidationController} from 'aurelia-validation';

@inject(NewInstance.of(ValidationController), I18N)

export class App {

firstName;

lastName;

controller;

i18n;

message = '';

constructor(controller, i18n) {

this.controller = controller;

this.i18n = i18n;

ValidationRules

.ensure((m: App) => m.firstName).required()

.ensure((m: App) => m.lastName).required()

.on(this);

}

submit() {

this.executeValidation();

}

switchLanguage() {

const currentLocale = this.i18n.getLocale();

this.i18n.setLocale(currentLocale === 'en' ? 'de' : 'en')

.then(() => this.executeValidation());

}

executeValidation() {

this.controller.validate()

.then(errors => {

if (errors.length === 0) {

this.message = this.i18n.tr('allGood');

} else {

this.message = this.i18n.tr('youHaveErrors');

}

});

}

}

### **Translation Files**

最后但并非最不重要的是，创建翻译文件，其中包括每个propertyName和每个验证消息键的翻译。下面是上述示例的德语和英语文件。注意，errorMessages部分有对required规则的翻译。在实践中，您需要对所使用的每条规则进行翻译。

{

"firstName": "Vorname",

"lastName": "Nachname",

"submit": "Abschicken",

"switchLanguage": "Sprache wechseln",

"youHaveErrors": "Es gibt Fehler!",

"allGood": "Alles in Ordnung!",

"latestValidationResult": "Aktuelles Validierungsergebnis",

"errorMessages": {

"required": "${$displayName} fehlt!"

}

}

// file: locales/en/translation.json

{

"firstName": "First name",

"lastName": "Last name",

"submit": "Submit",

"switchLanguage": "Switch language",

"youHaveErrors": "You have errors!",

"allGood": "All is good!",

"latestValidationResult": "Latest validation result",

"errorMessages": {

"required": "${$displayName} is missing!"

}

}

## Server-Side Validation

规则api和验证器api可以使用在服务端是nodejs的应用。