عنوان: **FluentValidation #2**
نویسنده: محمّد زارع
تاریخ: ۱۳۹۱/۰۸/۲۰ ۱۳:۵
آدرس: www.dotnettips.info
برچسب‌ها: Validation, FluentValidation

کتابخانه FluentValidation به صورت پیش فرض دارای تعدادی Validation می‌باشد که برای اکثر کارهای ابتدایی کافی می‌باشد.

NotNull	اطمینان از اینکه خاصیت مورد نظر Null نباشد
NotEmpty	اطمینان از اینکه خاصیت مورد نظر Null یا رشته خالی نباشد (یا مقدار پیش فرض نباشد، مثلا 0 برای int)
NotEqual	اطمینان از اینکه خاصیت مورد نظر برابر مقدار تعیین شده نباشد (یا برابر مقدار خاصیت دیگری نباشد)
Equal	اطمینان از اینکه خاصیت مور نظر برابر مقدار تعیین شده باشد (یا برابر مقدار خاصیت دیگری باشد)
Length	اطمینان از اینکه طول رشته‌ی خاصیت مورد نظر در محدوده خاصی باشد
LessThan	اطمینان از اینکه مقدار خاصیت مورد نظر کوچکتر از مقدار تعیین شده باشد (یا کوچکتر از خاصیت دیگری)
LessThanOrEqual	اطمینان از اینکه مقدار خاصیت مورد نظر کوچکتر یا مساوی مقدار تعیین شده باشد (یا کوچکتر مساوی مقدار خاصیت دیگری)
GreaterThan	اطمینان از اینکه مقدار خاصیت مورد نظر بزرگتر از مقدار تعیین شده باشد (یا بزرگتر از مقدار خاصیت دیگری)
GreaterThanOrEqual	اطمینان از اینکه مقدار خاصیت مورد نظر بزرگتر مساوی مقدار تعیین شده باشد (یا بزرگتر مساوی مقدار خاصیت دیگری)
Matches	اطمینان از اینکه مقدار خاصیت مورد نظر با عبارت باقائده (Regular Expression) تنظیم شده مطابقت داشته باشد
Must	اعتبارسنجی یک predicate با استفاده از Lambda Expressions. اگر عبارت Lambda مقدار true برگرداند اعتبارسنجی با موفقیت انجام شده و اگر false برگرداند، اعتبارسنجی با شکست مواجه شده است.
Email	اطمینان از اینکه مقدار خاصیت مورد نظر یک آدرس ایمیل معتبر باشد
CreditCard	اطمینان از اینکه مقدار خاصیت مورد نظر یک Credit Card باشد

همان طور که در جدول بالا ملاحظه می‌کنید بعضی از اعتبارسنجی‌ها را می‌توان با استفاده از مقدار خاصیت‌های دیگر انجام داد. برای درک این موضوع مثال زیر را در نظر بگیرید:

```
RuleFor(customer => customer.Surname).NotEqual(customer => customer.Forename);
```

در مثال بالا مقدار خاصیت Surname نباید برابر مقدار خاصیت Forename باشد.

برای تعیین اینکه در هنگام اعتبارسنجی چه پیامی به کاربر نمایش داده شود نیز می‌توان از متد WithMessage استفاده کرد:

```
RuleFor(customer => customer.Surname).NotNull().WithMessage("Please ensure that you have entered your
Surname");
```

اعتبارسنجی تنها در مواقع خاص

با استفاده از شرط‌های When و Unless می‌توان تعیین کرد که اعتبارسنجی فقط در مواقعی خاص انجام شود. به عنوان مثال در قطعه کد زیر با استفاده از متد When، تعیین می‌کنیم که اعتبارسنجی روی خاصیت CustomerDiscount تنها زمانی اتفاق بیفتد که خاصیت IsPreferredCustomer برابر true باشد.

```
RuleFor(customer => customer.CustomerDiscount).GreaterThan(0).When(customer =>
customer.IsPreferredCustomer);
```

متد Unless نیز برعکس متد When می‌باشد.
اگر نیاز به تعیین یک شرط یکسان برای چند خاصیت باشد، میتوان به جای تکرار شرط برای هرکدام از خاصیت‌ها به صورت زیر عمل کرد:

```
When(customer => customer.IsPreferred, () => {
   RuleFor(customer => customer.CustomerDiscount).GreaterThan(0);
   RuleFor(customer => customer.CreditCardNumber).NotNull();
});
```

تعیین نحوه برخورد با اعتبارسنجی‌های زنجیره ای

در قطعه کد زیر ملاحظه می‌کنید که از دو Validator برای یک خاصیت استفاده شده است. (*NotNull* و *NotEqual*)

```
RuleFor(x => x.Surname).NotNull().NotEqual("foo");
```

قطعه کد بالا بررسی می‌کند که مقدار خاصیت Surname، ابتدا برابر Null نباشد و پس از آن برابر رشته "Foo" نیز نباشد. در این حالت (حالت پیش فرض) اگر اعتبارسنجی اول (NotNull) با شکست مواجه شود، اعتبارسنجی دوم (NotEqual) نیز انجام خواهد شد. برای جلوگیری از این حالت می‌توان از CascadeMode به صورت زیر استفاده کرد:

```
RuleFor(x => x.Surname).Cascade(CascadeMode.StopOnFirstFailure).NotNull().NotEqual("foo");
```

اکنون اگر اعتبارسنجی NotNull با شکست مواجه شود، دیگر اعتبارسنجی دوم انجام نخواهد شد. این ویژگی در مواردی کاربرد دارد که یک زنجیره پیچیده از اعتبارسنجی‌ها داریم که شرط انجام هرکدام از آنها موفقیت در اعتبارسنجی‌های قبلی است.
اگر نیاز بود تا CascadeMode را برای تمام خاصیت‌های یک کلاس Validator تعیین کنیم می‌توان به صورت خلاصه از روش زیر استفاده کرد:

```
public class PersonValidator : AbstractValidator<Person> {
  public PersonValidator() {
    // First set the cascade mode
    CascadeMode = CascadeMode.StopOnFirstFailure;
```

```
    // Rule definitions follow
    RuleFor(...)
    RuleFor(...)
   }
}
```

سفارشی سازی اعتبارسنجی برای ایجاد اعتبارسنجی سفارشی دو راه وجود دارد:
راه اول ایجاد یک کلاس که از PropertyValidator مشتق می‌شود. برای توضیح نحوه استفاده از این راه، تصور کنید که میخواهیم یک اعتبارسنج سفارشی درست کنیم تا چک کند که یک لیست حتماً کمتر از 10 آیتم داخل خود داشته باشد. در این صورت کدی که بایستی نوشته شود به صورت زیر خواهد بود:

```
using System.Collections.Generic;
using FluentValidation.Validators;

public class ListMustContainFewerThanTenItemsValidator<T> : PropertyValidator {

public ListMustContainFewerThanTenItemsValidator()
: base("Property {PropertyName} contains more than 10 items!") {

}

protected override bool IsValid(PropertyValidatorContext context) {
var list = context.PropertyValue as IList<T>;

if(list != null && list.Count >= 10) {
return false;
}

return true;
}
}
```

کلاسی که از PropertyValidator مشتق می‌شود بایستی متد IsValid آن را override کند. متد IsValid یک PropertyValidatorContext را به عنوان ورودی می‌گیرد و یک boolean را که مشخص کننده نتیجه اعتبارسنجی است، بر می‌گرداند. همان طور که در مثال بالا ملاحظه می‌کنید پیغام خطا نیز در constructor مشخص شده است.
برای استفاده از این Validator سفارشی نیز می‌توان از متد SetValidator به صورت زیر استفاده نمود:

```
public class PersonValidator : AbstractValidator<Person> {
    public PersonValidator() {
       RuleFor(person => person.Pets).SetValidator(new
ListMustContainFewerThanTenItemsValidator<Pet>());
    }
}
```

راه دیگر استفاده از آن تعریف یک Extension Method می‌باشد که در این صورت می‌توان از آن به صورت زنجیره ای مانند دیگر Validatorها استفاده نمود:

```
public static class MyValidatorExtensions {
   public static IRuleBuilderOptions<T, IList<TElement>> MustContainFewerThanTenItems<T, TElement>(this
IRuleBuilder<T, IList<TElement>> ruleBuilder) {
      return ruleBuilder.SetValidator(new ListMustContainFewerThanTenItemsValidator<TElement>());
   }
}
```

اکنون برای استفاده از Extension Method می‌توان به راحتی مانند زیر عمل کرد:

```
public class PersonValidator : AbstractValidator<Person> {
    public PersonValidator() {
```

```
        RuleFor(person => person.Pets).MustContainFewerThanTenItems();
    }
}
```

راه دوم استفاده از متد Custom می‌باشد. برای توضیح نحوه استفاه از این متد مثال قبل (چک کردن تعداد آیتم‌های لیست) را به صورت زیر بازنویسی می‌کنیم:

```
public class PersonValidator : AbstractValidator<Person> {
   public PersonValidator() {
       Custom(person => {
           return person.Pets.Count >= 10
              ? new ValidationFailure("More than 10 pets is not allowed.")
              : null;
       });
   }
}
```

توجه داشته باشید که متد Custom تنها برای اعتبارسنجی‌های خیلی پیچیده طراحی شده است و در اکثر مواقع می‌توان خیلی راحت‌تر و تمیزتر از (Must PredicateValidator) برای اعتبارسنجی سفارشی مان استفاده کرد، مانند مثال زیر:

```
public class PersonValidator : AbstractValidator<Person> {
   public PersonValidator() {
      RuleFor(person => person.Pets).Must(HaveFewerThanTenPets).WithMessage("More than 9 pets is not
allowed");
   }

   private bool HaveFewerThanTenPets(IList<Pet> pets) {
      return pets.Count < 10;
   }
}
```

پ.ن.
در این دو مقاله سعی شد تا ویژگی‌های FluentValidation به صورت انتزاعی توضیح داده شود. در قسمت بعد نحوه استفاده از این کتابخانه در یک برنامه ASP.NET MVC نشان داده خواهد شد.

نظرات خوانندگان

نویسنده: alireza
تاریخ: ۱۳۹۲/۰۶/۳۰ ۱۴:۴۲

با سلام میشه مقایسه ای با validation تو کار ماکروسافت داشته باشید؟
با تشکر

نویسنده: محمّد زارع
تاریخ: ۱۳۹۲/۰۷/۰۲ ۹:۴۴

کنترل بهتر روی قوانین اعتبارسنجی.

جداسازی قوانین اعتبارسنجی از کلاس‌های ViewModel یا Model.

پشتیبانی خوب از Client Side Validation.
UnitTesting ساده‌تر نسبت به DataAnnotations.
ساده‌تر بودن نوشتن Custom Validator برای موارد خاص.
اعمال اعتبارسنجی شرطی با FluentValidation راحت‌تر است.
امکان اعتبارسنجی گروهی و ...