عنوان: **FluentValidation #1**
نویسنده: محمّد زارع
تاریخ: ۱۳۹۱/۰۸/۲۰ ۱۰:۵
آدرس: www.dotnettips.info
برچسب‌ها: Validation, FluentValidation

FluentValidation یک پروژه سورس باز برای اعتبارسنجی Business Objectها با استفاده از Fluent Interface و Lambda Expressions می‌باشد.

جهت نصب این کتابخانه دستور زیر را در Package Manager Console وارد نمایید:

```
PM> Install-Package FluentValidation
```

ایجاد یک Validator

برای تعریف مجموعه قوانین اعتبارسنجی برای یک موجودیت ابتدا بایستی یک کلاس ایجاد کرد که از AbstractValidator<T> مشتق می‌شود که T در اینجا برابر موجودیتی است که می‌خواهیم اعتبارسنجی کنیم. به عنوان مثال کلاس مشتری به صورت زیر را در نظر بگیرید:

```
public class Customer
{
      public int Id { get; set; }
      public string Surname { get; set; }
      public string Forename { get; set; }
      public decimal Discount { get; set; }
      public string Address { get; set; }
}
```

مجموعه قوانین اعتبارسنجی با استفاده از متد RuleFor و داخل متد سازنده کلاس Validator تعریف می‌شوند. به عنوان مثال برای اطمینان از اینکه مقدار خاصیت Surname برابر Null نباشد باید به صورت زیر عمل کرد:

```
using FluentValidation;

public class CustomerValidator : AbstractValidator<Customer>
{
      public CustomerValidator
      {
           RuleFor(customer => customer.Surname).NotNull();
      }
}
```

اعتبارسنجی زنجیره ای برای یک خاصیت

برای اعتبارسنجی یک خاصیت، می‌توان از چندین Validator باهم نیز استفاده کرد:

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

در اینجا خاصیت Surname نباید Null باشد و همچنین مقدار آن نباید برابر "Foo" باشد.
برای اجراکردن اعتبارسنجی، ابتدا یک نمونه از کلاس Validator مان را ساخته و شیء ای را که می‌خواهیم اعتبارسنجی کنیم به متد Validate آن میفرستیم:

```
Customer customer = new Customer();
CustomerValidator validator = new CustomerValidator();
ValidationResult results = validator.Validate(customer);
```

خروجی متد Validate، یک ValidationResult است که شامل دو خاصیت زیر می‌باشد:

IsValid: از نوع bool برای تعیین اینکه اعتبارسنجی موفقیت آمیز بوده یا خیر.
Errors: یک مجموعه از ValidationFailure که جزئیات تمام اعتبارسنجی‌های ناموفق را شامل می‌شود.

به عنوان مثال قطعه کد زیر، جزئیات اعتبارسنجی‌های ناموفق را نمایش می‌دهد:

```
Customer customer = new Customer();
CustomerValidator validator = new CustomerValidator();

ValidationResult results = validator.Validate(customer);

if(! results.IsValid)
{
     foreach(var failure in results.Errors)
     {
          Console.WriteLine("Property " + failure.PropertyName + " failed validation. Error was: " +
failure.ErrorMessage);
     }
}
```

پرتاب استثناها (Throwing Exceptions)

به جای برگرداندن ValidationResult شما میتوانید با کمک متد ValidateAndThrow به FluentValidation بگویید که هنگام اعتبارسنجی ناموفق یک استثنا پرتاب کند:

```
Customer customer = new Customer();
CustomerValidator validator = new CustomerValidator();
validator.ValidateAndThrow(customer);
```

در این صورت Validator یک ValidationException را پرتاب خواهد کرد که دربردارنده‌ی پیام‌های خطا در خاصیت Errors خود می‌باشد.

استفاده از Validatorها برای Complex Properties

جهت درک این ویژگی تصور کنید که کلاس‌های مشتری و آدرس و همچنین کلاس‌های مربوط به اعتبارسنجی آن‌ها را به صورت زیر داریم:

```
public class Customer
{
     public string Name { get; set; }
     public Address Address { get; set; }
}

public class Address
{
     public string Line1 { get; set; }
     public string Line2 { get; set; }
     public string Town { get; set; }
     public string County { get; set; }
     public string Postcode { get; set; }
}

public class AddressValidator : AbstractValidator<Address>
{
     public AddressValidator()
     {
          RuleFor(address => address.Postcode).NotNull();
          //etc
     }
}

public class CustomerValidator : AbstractValidator<Customer>
{
```

```
     public CustomerValidator()
     {
          RuleFor(customer => customer.Name).NotNull();
          RuleFor(customer => customer.Address).SetValidator(new AddressValidator())
      }
}
```

در این صورت وقتی متد Validate کلاس CustomerValidator را فراخوانی نمایید AddressValidator نیز فراخوانی خواهد شد و نتیجه این اعتبارسنجی به صورت یکجا در یک ValidationResult برگشت داده خواهد شد.

استفاده از Validatorها برای مجموعه‌ها (Collections)

Validatorها همچنین می‌توانند بر روی خاصیت هایی که شامل مجموعه ای از یک شیء دیگر هستند نیز استفاده شوند. به عنوان مثال یک مشتری که دارای لیستی از سفارشات است را در نظر بگیرید:

```
public class Customer
{
     public IList<Order> Orders { get; set; }
}

public class Order
{
     public string ProductName { get; set; }
     public decimal? Cost { get; set; }
}

var customer = new Customer();
customer.Orders = new List<Order>
{
     new Order { ProductName = "Foo" },
     new Order { Cost = 5 }
};
```

کلاس OrderValidator نیز به صورت زیر خواهد بود:

```
public class OrderValidator : AbstractValidator<Order>
{
     public OrderValidator()
     {
          RuleFor(x => x.ProductName).NotNull();
          RuleFor(x => x.Cost).GreaterThan(0);
     }
}
```

این Validator می‌تواند داخل CustomerValidator مورد استفاده قرار بگیرد (با استفاده از متد SetCollectionValidator):

```
public class CustomerValidator : AbstractValidator<Customer>
{
     public CustomerValidator()
     {
          RuleFor(x => x.Orders).SetCollectionValidator(new OrderValidator());
     }
}
```

می توان با استفاده از متد Where یا Unless روی اعتبارسنجی شرط گذاشت:

```
RuleFor(x => x.Orders).SetCollectionValidator(new OrderValidator()).Where(x => x.Cost != null);
```

گروه بندی قوانین اعتبارسنجی

RuleSetها به شما این امکان را می‌دهند تا بعضی از قوانین اعتبارسنجی را داخل یک گروه قرار دهید تا با یکدیگر اجرا شوند. در حالی که دیگر قوانین نادیده گرفته می‌شوند.
برای مثال تصور کنید شما سه خاصیت در کلاس Person دارید که شامل (Id, Surname, Forename) می‌باشند و همچنین یک قانون برای هرکدام از آن ها. میتوان قوانین مربوط به Surname و Forename را در یک RuleSet مجزا به نام Names قرار داد:

```
public class PersonValidator : AbstractValidator<Person>
{
     public PersonValidator()
     {
          RuleSet("Names", () =>
          {
               RuleFor(x => x.Surname).NotNull();
               RuleFor(x => x.Forename).NotNull();
          });

          RuleFor(x => x.Id).NotEqual(0);
      }
}
```

در اینجا دو خاصیت Surname و Forename با یکدیگر داخل یک RuleSet به نام Names گروه شده اند. برای اعتبارسنجی جداگانه این گروه نیز به صورت زیر می‌توان عمل کرد:

```
var validator = new PersonValidator();
var person = new Person();
var result = validator.Validate(person, ruleSet: "Names");
```

این ویژگی به شما این امکان را می‌دهد تا یک Validator پیچیده را به چندین قسمت کوچکتر تقسیم کرده و توانایی اعتبارسنجی این قسمت‌ها را به صورت جداگانه داشته باشید.

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 ...

عنوان: **استفاده از FluentValidation در ASP.NET MVC**
نویسنده: محمّد زارع
تاریخ: ۱۳۹۱/۰۸/۲۰ ۱۶:۵
آدرس: www.dotnettips.info
برچسب‌ها: MVC, Validation, FluentValidation

برای هماهنگی این کتابخانه با ASP.NET MVC نیاز به نصب FluentValidation.Mvc3 یا FluentValidation.Mvc4 از طریق Nuget یا دانلود کتابخانه از سایت CodePlex می‌باشد. بعد از نصب کتابخانه، نیاز به تنظیم FluentValidationModelValidatorProvider داخل متد Application_Start (فایل Global.asax) داریم:

```
protected void Application_Start() {
    AreaRegistration.RegisterAllAreas();

    RegisterGlobalFilters(GlobalFilters.Filters);
    RegisterRoutes(RouteTable.Routes);

    FluentValidationModelValidatorProvider.Configure();
}
```

تصور کنید دو کلاس Person و PersonValidator را به صورت زیر داریم:

```
[Validator(typeof(PersonValidator))]
    public class Person {
    public int Id { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
    public int Age { get; set; }
}

public class PersonValidator : AbstractValidator<Person> {
    public PersonValidator() {
        RuleFor(x => x.Id).NotNull();
        RuleFor(x => x.Name).Length(0, 10);
        RuleFor(x => x.Email).EmailAddress();
        RuleFor(x => x.Age).InclusiveBetween(18, 60);
    }
}
```

همان طور که ملاحظه می‌کنید، در بالای تعریف کلاس Person با استفاده از ValidatorAttribute مشخص کرده ایم که از PersonValidator جهت اعتبارسنجی استفاده کند.
در آخر می‌توانیم Controller و View ی برنامه مان را درست کنیم:

```
public class PeopleController : Controller {
    public ActionResult Create() {
        return View();
    }

    [HttpPost]
    public ActionResult Create(Person person) {

        if(! ModelState.IsValid) { // re-render the view when validation failed.
            return View("Create", person);
        }

        TempData["notice"] = "Person successfully created";
        return RedirectToAction("Index");
    }
}
```

```
@Html.ValidationSummary()

@using (Html.BeginForm()) {
Id: @Html.TextBoxFor(x => x.Id) @Html.ValidationMessageFor(x => x.Id)
<br />
Name: @Html.TextBoxFor(x => x.Name) @Html.ValidationMessageFor(x => x.Name)
<br />
Email: @Html.TextBoxFor(x => x.Email) @Html.ValidationMessageFor(x => x.Email)
<br />
Age: @Html.TextBoxFor(x => x.Age) @Html.ValidationMessageFor(x => x.Age)
```

```
<input type="submit" value="submit" />
}
```

اکنون DefaultModelBinder موجود در MVC برای اعتبارسنجی شیء Person از FluentValidationModelValidatorProvider استفاده خواهد کرد.
توجه داشته باشید که FluentValidation با اعتبارسنجی سمت کاربر ASP.NET MVC به خوبی کار خواهد کرد منتها نه برای تمامی اعتبارسنجی ها. به عنوان مثال تمام قوانینی که از شرط‌های When/Unless استفاده کرده اند، Validatorهای سفارشی، و قوانینی که در آن‌ها از Must استفاده شده باشد، اعتبارسنجی سمت کاربر نخواهند داشت. در زیر لیست Validator هایی که با اعتبارسنجی سمت کاربر به خوبی کار خواهند کرد آمده است:

NotNull/NotEmpty

Matches

InclusiveBetween

CreditCard

Email

EqualTo

Length

صفت CustomizeValidator

با استفاده از CustomizeValidatorAttribute می‌توان نحوه اجرای Validator را تنظیم کرد. به عنوان مثال اگر میخواهید که Validator تنها برای یک RuleSet مخصوص انجام شود می‌توانید مانند زیر عمل کنید:

```
public ActionResult Save([CustomizeValidator(RuleSet="MyRuleset")] Customer cust) {
  // ...
}
```

مواردی که تا اینجا گفته شد برای استفاده در یک برنامه‌ی ساده MVC کافی به نظر می‌رسد، اما از اینجا به بعد مربوط به مواقعی است که نخواهیم از Attributeها استفاده کنیم و کار را به IoC بسپاریم.

استفاده از Validator Factory با استفاده از یک IoC Container

Validator Factory چیست؟ Validator Factory یک کلاس می‌باشد که وظیفه ساخت نمونه از Validatorها را بر عهده دارد. اینترفیس IValidatorFactory به صورت زیر می‌باشد:

```
public interface IValidatorFactory {
   IValidator<T> GetValidator<T>();
   IValidator GetValidator(Type type);
}
```

ساخت Validator Factory سفارشی:

برای ساخت یک Validator Factory شما می‌توانید به طور مستقیم IValidatorFactory را پیاده سازی نمایید یا از کلاس ValidatorFactoryBase به عنوان کلاس پایه استفاده کنید (که مقداری از کارها را برای شما انجام داده است). در این مثال نحوه ایجاد یک Validator Factory که از StructureMap استفاده می‌کند را بررسی خواهیم کرد. ابتدا نیاز به ثبت Validatorها در

StructureMap داریم:

```
ObjectFactory.Configure(cfg => cfg.AddRegistry(new MyRegistry()));

public class MyRegistry : Registry {
    public MyRegistry() {
        For<IValidator<Person>>()
    .Singleton()
    .Use<PersonValidator>();
    }
}
```

در اینجا StructureMap را طوری تنظیم کرده ایم که از یک Registry سفارشی استفاده کند. در داخل این Registry به StructureMap میگوییم که زمانی که خواسته شد تا یک نمونه از IValidator<Person> ایجاد کند، PersonValidator را بر گرداند. متد CreateInstance نوع مناسب را نمونه سازی می‌کند (CustomerValidator) و آن را بازمی گرداند (یا Null بر می‌گرداند اگر نوع مناسبی وجود نداشته باشد)

استفاده از AssemblyScanner

FluentValidation دارای یک AssemblyScanner می‌باشد که کار ثبت Validatorها داخل یک اسمبلی را راحت‌تر می‌سازد. با استفاده از AssemblyScanner کلاس MyRegistery ما شبیه قطعه کد زیر خواهد شد:

```
public class MyRegistry : Registry {
   public MyRegistry() {

     AssemblyScanner.FindValidatorsInAssemblyContaining<MyValidator>()
       .ForEach(result => {
            For(result.InterfaceType)
               .Singleton()
               .Use(result.ValidatorType);
       });
   }
}
```

حالا زمان استفاده از factory ساخته شده در متد Application_Start برنامه می‌باشد:

```
protected void Application_Start() {
    RegisterRoutes(RouteTable.Routes);

    //Configure structuremap
    ObjectFactory.Configure(cfg => cfg.AddRegistry(new MyRegistry()));
    ControllerBuilder.Current.SetControllerFactory(new StructureMapControllerFactory());

    //Configure FV to use StructureMap
    var factory = new StructureMapValidatorFactory();

    //Tell MVC to use FV for validation
    ModelValidatorProviders.Providers.Add(new FluentValidationModelValidatorProvider(factory));
    DataAnnotationsModelValidatorProvider.AddImplicitRequiredAttributeForValueTypes = false;
}
```

اکنون FluentValidation از StructureMap برای نمونه سازی Validatorها استفاده خواهد کرد و کار اعتبارسنجی مدل‌ها به FluentValidaion سپرده شده است.

نظرات خوانندگان

نویسنده: محمد صاحب
تاریخ: ۱۳۹۱/۰۸/۲۱ ۸:۵۱

با تشکر مجدد از شما...
برای مواردی که سمت کلاینت ساپورت نمیشن راه حل ی وجود داره؟
بهتره این اعتبارسنجی‌ها تو کدوم لایه نوشته بشن؟

نویسنده: میثم زارع
تاریخ: ۱۳۹۱/۰۸/۲۱ ۹:۴۷

در مورد سوال اول : Custom validators with client side و Enable custom validator client side in FV
در مورد سوال دوم هم، اکثر مواقع روی ViewModel یا بهتر بگم InputModel انجام میشه، هر چند اگر نیاز بود میشه روی خود کلاس Entity مورد نظر هم ایجاد کرد.
اینجا خود سازنده کتابخانه توضیح داده که چطور ازش استفاده میکنه:
http://fluentvalidation.codeplex.com/discussions/355068

نویسنده: نارینه
تاریخ: ۱۳۹۱/۱۱/۰۱ ۱۳:۲۶

StructureMapValidatorFactory که در Application_Start() استفاده شده است، د رکجا و به چه شکلی تعریف گردیده؟
امکان دارد نمونه کد کامل را جهت استفاده قرار دهید؟

نویسنده: محمد صاحب
تاریخ: ۱۳۹۱/۱۱/۰۱ ۱۴:۳۷

```
public class StructureMapValidatorFactory : ValidatorFactoryBase {
public override IValidator CreateInstance(Type validatorType) {
return ObjectFactory.TryGetInstance(validatorType) as IValidator;
}
}
```

اطلاعات بیشتر