Valit Documentation

Release 0.1.0

Dariusz Pawlukiewicz, Daniel Barwikowski, Arkadiusz Bal

Oct 07, 2018

Contents

Valit is **dead simple validation** for .NET Core. No more if-statements all around your code. Write nice and clean fluent validators instead!

CHAPTER 1

Getting started

This part will guide you through very basic concepts of Valit, including instalation proccess and creating simple validator for domain object.

1.1 Installation

Valit is available on NuGet.

1.1.1 Package manager

```
Install-Package Valit -Version 2.0.0
```

1.1.2 .NET CLI

```
dotnet add package Valit --version 2.0.0
```

1.2 Creating basic validator

In order to create a validator you need to go through few steps. It's worth mentioning that not all of them are mandatory. The steps are:

- creating new instance of validator using `Create()` static method.
- choosing validation *strategy* using `WithStrategy()` method **(not required)**.
- selecting property using `Ensure()` method and defining rules for it.
- Extending rules with *custom errors* (such as messages or error codes), *tags* and *conditions*. **(not required)**.

- applying created rules to an object using `For()` method.

Having the validator created, simply invoke `Validate()` method which will produce the result with all the data.

Let's try it out with very practical example. Imagine that you're task is to validate model sent from registration page of your app. The example object might look as follows:

```
public class RegisterModel
{
    public string Email { get; set; }
    public string Password { get; set; }
    public ushort Age { get; set ;}
}
```

These are the validation criteria:

- `Email` is required and needs to be a valid email address.
- `Password` is required and needs to be at least 10 characters long.
- `Age` must be greater than 16.

This is how you can handle such scenario using Valit:

```
void ValidateModel(RegisterModel model)
{
    var result = ValitRules<RegisterModel>
        .Create()
        .Ensure(m => m.Email, _=>_
            .Required()
            .Email())
        .Ensure(m => m.Password, _=>_
            .Required()
            .MinLength(10))
        .Ensure(m => m.Age, _=>_
            .IsGreaterThan(16))
        .For(model)
        .Validate();

    if(result.Succeeded)
    {
        // do something on success
    }
    else
    {
        // do something on failure
    }
}
```

Pretty cool, right? Of course, the above example was fairly simple but trust us. From now on, even complicated validation criterias won't scare you anymore ;)

1.3 Need help?

If you need some help, feel free to look at more examples on GitHub. Also don't hesitate to open new issues if something could be done better!

CHAPTER 2

Validation rules

2.1 Supported data types

Valit provides validation rules for plenty different data types **with full support for** `Nullable<T>`. The supported data types are:

- `Boolean`
- `Byte`
- `DateTime`
- `DateTimeOffset`
- `Decimal`
- `Double`
- `Float`
- `Guid`
- `IEnumerable<T>`
- `Int16`
- `Int32`
- `Int64`
- `SByte`
- `String`
- `TimeSpan`
- `UInt16`
- `UInt32`
- `UInt64`

2.2 Available validation rules

Because of differences between particular data types, each one contains its own set of available validation rules. However, there are two general extensions which can be applied on any type you want:

- `Satisifies<T>(Predicate<T>)` - checks whether given predicate returns `true`.
- `Required<T>()` - checks whether given **reference type** is not `null`.

Below you can find the full list of available validation rules for supported data types.

Note: `null` either on validated property or given value always results rule returning `false`

2.2.1 Boolean

- `IsTrue(bool|bool?)` - checks whether `bool|bool?` property is equal to `true`.
- `IsFalse(bool|bool?)` - checks whether `bool|bool?` property is equal to `false`.
- `IsEqualTo(bool|bool?)` - checks whether `bool|bool?` property is equal to given value.
- `Required()` - checks whether `bool?` property is not `null`.

2.2.2 Byte

- `IsEqualTo(byte|byte?)` - checks whether `byte|byte?` property is equal to given value.
- `IsGreaterThan(byte|byte?)` - checks whether `byte|byte?` property is greater than given value.
- `IsGreaterThanOrEqualTo(byte|byte?)` - checks whether `byte|byte?` property is greater than or equal to given value.
- `IsLessThan(byte|byte?)` - checks whether `byte|byte?` property is less than given value.
- `IsLessThanOrEqualTo(byte|byte?)` - checks whether `byte|byte?` property is less than or equal to given value.
- `IsNonZero()` - checks whether `byte|byte?` property is not a zero.
- `Required()` - checks whether `byte?` property is not `null`.

2.2.3 DateTime

- `IsAfter(DateTime|DateTime?)` - checks whether `DateTime|DateTime?` property is after (greater than) given value.
- `IsAfterOrSameAs(DateTime|DateTime?)` - checks whether `DateTime|DateTime?` property is after or same as (greater than or equal to) given value.
- `IsAfterNow()` - checks whether `DateTime|DateTime?` property is after (greater than) `DateTime.Now`.
- `IsAfterUtcNow()` - checks whether `DateTime|DateTime?` property is after (greater than) `DateTime.UtcNow`.
- `IsBefore(DateTime|DateTime?)` - checks whether `DateTime|DateTime?` property is before (less than) given value.

- `IsBeforeOrSameAs(DateTime|DateTime?)` - checks whether `DateTime|DateTime?` property is before or same as (less than or equal to) given value.
- `IsBeforeNow()` - checks whether `DateTime|DateTime?` property is before (less than) `DateTime.Now`.
- `IsBeforeUtcNow()` - checks whether `DateTime|DateTime?` property is before (less than) `DateTime.UtcNow`.
- `IsSameAs(DateTime|DateTime?)` - checks whether `DateTime|DateTime?` property is same as (equal to) given value.
- `Required()` - checks whether `DateTime?` property is not `null`.

2.2.4 DateTimeOffset

- `IsAfter(DateTimeOffset|DateTimeOffset?)` - checks whether `DateTimeOffset|DateTimeOffset?` property is after (greater than) given value.
- `IsAfterOrSameAs(DateTimeOffset|DateTimeOffset?)` - checks whether `DateTimeOffset|DateTimeOffset?` property is after or same as (greater than or equal to) given value.
- `IsAfterNow()` - checks whether `DateTimeOffset|DateTimeOffset?` property is after (greater than) `DateTime.Now`.
- `IsAfterUtcNow()` - checks whether `DateTimeOffset|DateTimeOffset?` property is after (greater than) `DateTime.UtcNow`.
- `IsBefore(DateTimeOffset|DateTimeOffset?)` - checks whether `DateTimeOffset|DateTimeOffset?` property is before (less than) given value.
- `IsBeforeOrSameAs(DateTimeOffset|DateTimeOffset?)` - checks whether `DateTimeOffset|DateTimeOffset?` property is before or same as (less than or equal to) given value.
- `IsBeforeNow()` - checks whether `DateTimeOffset|DateTimeOffset?` property is before (less than) `DateTime.Now`.
- `IsBeforeUtcNow()` - checks whether `DateTimeOffset|DateTimeOffset?` property is before (less than) `DateTime.UtcNow`.
- `IsSameAs(DateTimeOffset|DateTimeOffset?)` - checks whether `DateTimeOffset|DateTimeOffset?` property is same as (equal to) given value.
- `Required()` - checks whether `DateTimeOffset?` property is not `null`.

2.2.5 Decimal

- `IsEqualTo(decimal|decimal?)` - checks whether `decimal|decimal?` property is equal to given value.
- `IsGreaterThan(decimal|decimal?)` - checks whether `decimal|decimal?` property is greater than given value.
- `IsGreaterThanOrEqualTo(decimal|decimal?)` - checks whether `decimal|decimal?` property is greater than or equal to given value.
- `IsLessThan(decimal|decimal?)` - checks whether `decimal|decimal?` property is less than given value.

- `IsLessThanOrEqualTo(decimal|decimal?)` - checks whether `decimal|decimal?` property is less than or equal to given value.
- `IsNonZero()` - checks whether `decimal|decimal?` property is not a zero.
- `IsNegative()` - checks whether `decimal|decimal?` property is negative **(less than zero)**.
- `IsPositive()` - checks whether `decimal|decimal?` property is positive **(greater than zero)**.
- `Required()` - checks whether `decimal?` property is not `null`.

2.2.6 Double

- `IsEqualTo(double|double?)` - checks whether `double|double?` property is equal to given value.
- `IsGreaterThan(double|double?)` - checks whether `double|double?` property is greater than given value.
- `IsGreaterThanOrEqualTo(double|double?)` - checks whether `double|double?` property is greater than or equal to given value.
- `IsLessThan(double|double?)` - checks whether `double|double?` property is less than given value.
- `IsLessThanOrEqualTo(double|double?)` - checks whether `double|double?` property is less than or equal to given value.
- `IsNonZero()` - checks whether `double|double?` property is not a zero.
- `IsNegative()` - checks whether `double|double?` property is negative **(less than zero)**.
- `IsPositive()` - checks whether `double|double?` property is positive **(greater than zero)**.
- `IsNaN()` - checks whether `double|double?` property is `double.NaN`.
- `IsNumber()` - checks whether `double|double?` property is not `double.NaN`.
- `Required()` - checks whether `double?` property is not `null`.

2.2.7 Float

- `IsEqualTo(float|float?)` - checks whether `float|float?` property is equal to given value.
- `IsGreaterThan(float|float?)` - checks whether `float|float?` property is greater than given value.
- `IsGreaterThanOrEqualTo(float|float?)` - checks whether `float|float?` property is greater than or equal to given value.
- `IsLessThan(float|float?)` - checks whether `float|float?` property is less than given value.
- `IsLessThanOrEqualTo(float|float?)` - checks whether `float|float?` property is less than or equal to given value.
- `IsNonZero()` - checks whether `float|float?` property is not a zero.
- `IsNegative()` - checks whether `float|float?` property is negative **(less than zero)**.
- `IsPositive()` - checks whether `float|float?` property is positive **(greater than zero)**.
- `IsNaN()` - checks whether `float|float?` property is `float.NaN`.
- `IsNumber()` - checks whether `float|float?` property is not `float.NaN`.
- `Required()` - checks whether `float?` property is not `null`.

2.2.8 Guid

- `IsEqualTo(Guid|Guid?)` - checks whether `Guid|Guid?` property is equal to given value.
- `IsNotEmpty()` - checks whether `Guid|Guid?` property is not `Guid.Empty`.
- `Required()` - checks whether `Guid?` property is not `null`.

2.2.9 IEnumerable<T>

- `MinItems(int)` - checks whether collection contains at least given number of elements.
- `MaxItems(int)` - checks whether collection contains at most given number of elements.

2.2.10 Int16

- `IsEqualTo(short|short?)` - checks whether `short|short?` property is equal to given value.
- `IsGreaterThan(short|short?)` - checks whether `short|short?` property is greater than given value.
- `IsGreaterThanOrEqualTo(short|short?)` - checks whether `short|short?` property is greater than or equal to given value.
- `IsLessThan(short|short?)` - checks whether `short|short?` property is less than given value.
- `IsLessThanOrEqualTo(short|short?)` - checks whether `short|short?` property is less than or equal to given value.
- `IsNonZero()` - checks whether `short|short?` property is not a zero.
- `IsNegative()` - checks whether `short|short?` property is negative **(less than zero)**.
- `IsPositive()` - checks whether `short|short?` property is positive **(greater than zero)**.
- `Required()` - checks whether `short?` property is not `null`.

2.2.11 Int32

- `IsEqualTo(int|int?)` - checks whether `int|int?` property is equal to given value.
- `IsGreaterThan(int|int?)` - checks whether `int|int?` property is greater than given value.
- `IsGreaterThanOrEqualTo(int|int?)` - checks whether `int|int?` property is greater than or equal to given value.
- `IsLessThan(int|int?)` - checks whether `int|int?` property is less than given value.
- `IsLessThanOrEqualTo(int|int?)` - checks whether `int|int?` property is less than or equal to given value.
- `IsNonZero()` - checks whether `int|int?` property is not a zero.
- `IsNegative()` - checks whether `int|int?` property is negative **(less than zero)**.
- `IsPositive()` - checks whether `int|int?` property is positive **(greater than zero)**.
- `Required()` - checks whether `int?` property is not `null`.

2.2.12 Int64

- `IsEqualTo(long|long?)` - checks whether `long|long?` property is equal to given value.
- `IsGreaterThan(long|long?)` - checks whether `long|long?` property is greater than given value.
- `IsGreaterThanOrEqualTo(long|long?)` - checks whether `long|long?` property is greater than or equal to given value.
- `IsLessThan(long|long?)` - checks whether `long|long?` property is less than given value.
- `IsLessThanOrEqualTo(long|long?)` - checks whether `long|long?` property is less than or equal to given value.
- `IsNonZero()` - checks whether `long|long?` property is not a zero.
- `IsNegative()` - checks whether `long|long?` property is negative **(less than zero)**.
- `IsPositive()` - checks whether `long|long?` property is positive **(greater than zero)**.
- `Required()` - checks whether `long?` property is not `null`.

2.2.13 SByte

- `IsEqualTo(sbyte|sbyte?)` - checks whether `sbyte|sbyte?` property is equal to given value.
- `IsGreaterThan(sbyte|sbyte?)` - checks whether `sbyte|sbyte?` property is greater than given value.
- `IsGreaterThanOrEqualTo(sbyte|sbyte?)` - checks whether `sbyte|sbyte?` property is greater than or equal to given value.
- `IsLessThan(sbyte|sbyte?)` - checks whether `sbyte|sbyte?` property is less than given value.
- `IsLessThanOrEqualTo(sbyte|sbyte?)` - checks whether `sbyte|sbyte?` property is less than or equal to given value.
- `IsNonZero()` - checks whether `sbyte|sbyte?` property is not a zero.
- `IsNegative()` - checks whether `sbyte|sbyte?` property is negative **(less than zero)**.
- `IsPositive()` - checks whether `sbyte|sbyte?` property is positive **(greater than zero)**.
- `Required()` - checks whether `sbyte?` property is not `null`.

2.2.14 String

- `Email()` - checks whether `string` property is a correct email address.
- `IsEqualTo(string)` - checks whether `string` property is equal to given value.
- `Matches(string)` - checks whether `string` property matches given **regex**.
- `MinLength(int)` - checks whether `string` property has at least given number of characters.
- `MaxLength(int)` - checks whether `string` property has at most given number of characters.
- `Required()` - checks whether `string` property is not `null`.

2.2.15 TimeSpan

- `IsEqualTo(TimeSpan|TimeSpan?)` - checks whether `TimeSpan|TimeSpan?` property is equal to given value.
- `IsGreaterThan(TimeSpan|TimeSpan?)` - checks whether `TimeSpan|TimeSpan?` property is greater than given value.
- `IsGreaterThanOrEqualTo(TimeSpan|TimeSpan?)` - checks whether `TimeSpan|TimeSpan?` property is greater than or equal to given value.
- `IsLessThan(TimeSpan|TimeSpan?)` - checks whether `TimeSpan|TimeSpan?` property is less than given value.
- `IsLessThanOrEqualTo(TimeSpan|TimeSpan?)` - checks whether `TimeSpan|TimeSpan?` property is less than or equal to given value.
- `IsNonZero()` - checks whether `TimeSpan|TimeSpan?` property is not a zero.
- `Required()` - checks whether `TimeSpan?` property is not `null`.

2.2.16 UInt16

- `IsEqualTo(ushort|ushort?)` - checks whether `ushort|ushort?` property is equal to given value.
- `IsGreaterThan(ushort|ushort?)` - checks whether `ushort|ushort?` property is greater than given value.
- `IsGreaterThanOrEqualTo(ushort|ushort?)` - checks whether `ushort|ushort?` property is greater than or equal to given value.
- `IsLessThan(ushort|ushort?)` - checks whether `ushort|ushort?` property is less than given value.
- `IsLessThanOrEqualTo(ushort|ushort?)` - checks whether `ushort|ushort?` property is less than or equal to given value.
- `IsNonZero()` - checks whether `ushort|ushort?` property is not a zero.
- `IsNegative()` - checks whether `ushort|ushort?` property is negative **(less than zero)**.
- `IsPositive()` - checks whether `ushort|ushort?` property is positive **(greater than zero)**.
- `Required()` - checks whether `ushort?` property is not `null`.

2.2.17 UInt32

- `IsEqualTo(uint|uint?)` - checks whether `uint|uint?` property is equal to given value.
- `IsGreaterThan(uint|uint?)` - checks whether `uint|uint?` property is greater than given value.
- `IsGreaterThanOrEqualTo(uint|uint?)` - checks whether `uint|uint?` property is greater than or equal to given value.
- `IsLessThan(uint|uint?)` - checks whether `uint|uint?` property is less than given value.
- `IsLessThanOrEqualTo(uint|uint?)` - checks whether `uint|uint?` property is less than or equal to given value.
- `IsNonZero()` - checks whether `uint|uint?` property is not a zero.
- `IsNegative()` - checks whether `uint|uint?` property is negative **(less than zero)**.
- `IsPositive()` - checks whether `uint|uint?` property is positive **(greater than zero)**.

- `Required()` - checks whether `uint?` property is not `null`.

2.2.18 UInt64

- `IsEqualTo(ulong|ulong?)` - checks whether `ulong|ulong?` property is equal to given value.
- `IsGreaterThan(ulong|ulong?)` - checks whether `ulong|ulong?` property is greater than given value.
- `IsGreaterThanOrEqualTo(ulong|ulong?)` - checks whether `ulong|ulong?` property is greater than or equal to given value.
- `IsLessThan(ulong|ulong?)` - checks whether `ulong|ulong?` property is less than given value.
- `IsLessThanOrEqualTo(ulong|ulong?)` - checks whether `ulong|ulong?` property is less than or equal to given value.
- `IsNonZero()` - checks whether `ulong|ulong?` property is not a zero.
- `IsNegative()` - checks whether `ulong|ulong?` property is negative **(less than zero)**.
- `IsPositive()` - checks whether `ulong|ulong?` property is positive **(greater than zero)**.
- `Required()` - checks whether `ulong?` property is not `null`.

2.3 Validating nested objects

Sometimes there could be need to validate nested objects. You can do this by creating `IValitator` (more details *here*). Let's say we have the following model:

```
public class OrderModel
{
    public AddressModel Address { get; set; }
}

public class AddressModel
{
    public string City { get; set; }
    public string Street { get; set; }
}
```

We'd like to be sure that `City` and `Street` in our `OrderModel` are not `null`. To do that we have to create the following valitator:

```
public class AdressModelValitator : IValitator<AddressModel>
{
    public IValitResult Validate(AddressModel @object, IValitStrategy strategy = null)
        => ValitRules<AddressModel>
            .Create()
            .Ensure(am => am.City, _ => _
                .Required())
            .Ensure(am => am.City, _ => _
                .Required())
            .For(@object)
            .Validate();
}
```

Then we can validate our `OrderModel` using the created valitator:

```
void ValidateModel(OrderModel model)
{
    var result = ValitRules<OrderModel>
            .Create()
            .Ensure(m => m.Address, new AdressModelValitator())
            .For(model)
            .Validate();
}
```

2.4 Validating collections

Using Valit you can also easily validate collections. Let's assume that you have following model:

```
public class ContactModel
{
    public IEnumerable<string> Emails { get; set; }
}
```

We'd like to check whether each item in `Emails` list is neither `null` nor `string.Empty` and is also valid email address. To validate collections use `EnsureFor`:

```
void Validate(ContactModel model)
{
    var result = ValitRules<ContactModel>
            .Create()
        .WithStrategy(picker => picker.FailFast)
        .EnsureFor(m => m.Emails, _=>_
            .Required()
            .WithMessage("Email list is empty")
            .Email()
            .WithMessage("Email list contains incorrect addresses"))
        .For(model)
        .Validate();
}
```

2.5 Conditional rules

In some cases there might be need to apply certain validation rules only if specific conditions are fulfilled. Valit allows you to do this using `When()` extension which can be applied on each rule. Let's say we have the following model:

```
public class RegisterModel
{
    public string Email { get; set; }
    public string Password { get; set; }
    public string CompanyName { get; set; }
    public bool IsCompanyMember { get; set; }
}
```

We'd like `CompanyName` to be required only if the user belongs to some company (defined by `IsCompanyMemeber`). Of course, we could create two separate validators to handle both scenarios, but a lot of code would be duplicated. That's where conditional rules come into play:

```
void ValidateModel(RegisterModel model)
{

    var result = ValitRules<RegisterModel>
        .Create()
        .Ensure(m => m.Email, _=>_
            .Required()
            .Email())
        .Ensure(m => m.Password, _=>_
            .Required()
            .MinLength(10))
        .Ensure(m => m.CompanyName, _=>_
            .Required()
                .When(m => m.IsCompanyMember))
        .For(model)
        .Validate();
}
```

Using `When()` we created simple validation condition which solves the issue.

Note: You can apply as much conditions as you want on each rule. If so, they will be merged into one condition using **logical AND** operator.

2.6 Tagging rules

Each validation rule can have its own set of tags. Tags are used for defining subset of rules that will be checked during the validation process. To add tags to the validation rule use `Tag()` extension. The code below presents the example usage:

```
public class RegisterModel
{
    public string Email { get; set; }
    public int Age { get; set; }
}

void ValidateModel(RegisterModel model)
{
    IValitRules<RegisterModel> valitRules = ValitRules<RegisterModel>
        .Create()
        .Ensure(m => m.Email, _=>_
            .Required()
                .Tag("A")
            .Email()
                .Tag("A", "B"))
        .Ensure(m => m.Age, _=>_
            .IsGreaterThan(16)
                .Tag("B"))
        .For(model);

    IValitResult result1 = valitRules.Validate("A"); // Checks Required, Email
    IValitResult result2 = valitRules.Validate("B"); //Checks Email, IsGreaterThan
    IValitResult result3 = valitRules.Validate("A", "B"); //Checks all rules
}
```

As you see in the example above, instead of invoking the `Validate()` right after the `For()` method, we assigned the rules to the variable. Then, using the `Validate()` overload which accepts tags, we created three different validation processes. That gave as a lot of flexibility by creating one general set of validation rules instead of three separated. It is important that set of tags passed to the `Validate()` method defines that **each rule must have at least one of them, NOT all.**

`Validate()` method has also an overload which accepts `Predicate<IValitRule<TModel>>`:

```
var result1 = rules.Validate(rule => rule.Tags.Any(tag => tag == "A")); // Checks␣
↪Required, Email
```

`ValitRules<T>` object also provides three methods for getting set of your validation rules. The methods are:

- `GetAllRules()` - gets all rules.
- `GetTaggedRules()` - gets rules which has at least one tag.
- `GetUntaggedRules()` - gets rules with no tags.

Of course if you want to get rules with the specific tags, you can use **Linq** for that purpose like in the example below:

```
IEnumerable<IValitRule<RegisterModel>> rulesOnlyWithATag= ValitRules<RegisterModel>
    .Create()
    .Ensure(m => m.Email, _=>_
        .Required().Tag("A")
        .Email().Tag("B"))
    .GetTaggedRules()
        .Where(rule => rule.Tags.Contains("A")); // only Required method is selected
```

Having the set of rules, you can use them for instantiating new `ValitRules<T>` object using `Create()` method overload:

```
void ValidateModel(RegisterModel model, IEnumerable<IValitRule<RegisterModel>>␣
↪rulesOnlyWithATag)
{
    IValitResult result = ValitRules<RegisterModel>
        .Create(rulesOnlyWithATag)
        .For(model)
        .Validate(); // validates the model using only Required rule on Email property
}
```

CHAPTER 3

Valitators

In most of scenarios, you might not want to create new instance of `IValitRules<TObject>` for each new object. It's better to have one class responsible for handling the validation for the particular type. Using other words, it's better to have a valitator ;) Valit offers three ways of creating a valitators for a particular type:

- Transforming `IValitRules<TObject>` object using `CreateValitator()` method.
- Transforming `IValitRulesProvider<TObject>` object using `CreateValitator()` method.
- Creating class which implements `IValitator<TObject>` interface

`IValitator<TObject>` object provides `Validate()` method which accepts two arguments:

- `TObject @object` - object of the particular type for validation
- `IValitStrategy strategy` - strategy for validation process. If this argument is not given, **Complete** strategy is picked as a default.

Let's say we'd like to create a valitator for the following type:

```
public class RegisterModel
{
    public string Email { get; set; }
    public string Password { get; set; }
    public ushort Age { get; set ;}
}
```

Below you can find three mentioned ways of doing that.

3.1 Creating valitator from rules

```
class RegistrationService
{
    private readonly IValitator<RegisterModel> _registerModelValitator;
```

(continues on next page)

(continued from previous page)

```
    public RegistrationService()
    {
        _registerModelValitator = CreateValitator();
    }

    public bool Register(RegisterModel model)
    {
        var result = _registerModelValitator.Validate(model);

        if(!result.Succeeded)
        {
            return false;
        }
        ...
    }

    private IValitator<RegisterModel> CreateValitator()
        => ValitRules<RegisterModel>
            .Create()
            .Ensure(m => m.Email, _=>_
                    .Required()
                    .Email())
            .Ensure(m => m.Password, _=>_
                    .Required()
                    .MinLength(10))
            .Ensure(m => m.Age, _=>_
                    .IsGreaterThan(16))
            .CreateValitator();
}
```

3.2 Creating valitator class

```
class RegistrationService
{
    private readonly IValitator<RegisterModel> _registerModelValitator;

    public RegistrationService()
    {
        _registerModelValitator = new RegisterModelValitator();
    }

    public bool Register(RegisterModel model)
    {
        var result = _registerModelValitator.Validate(model);

        if(!result.Succeeded)
        {
            return false;
        }
        ...
    }
}

class RegisterModelValitator : IValitator<RegisterModel>
```

(continues on next page)

(continued from previous page)

```
{
    private readonly IValitRulesStrategyPicker<TObject> _strategyPicker;

    public RegisterModelValitator()
    {
        var rules = GetValidationRules();
        _strategyPicker = ValitRules<RegisterModel>.Create(rules);
    }

    public IValitResult Validate(RegisterModel @object, IValitStrategy strategy)
    {
        var selectedStrategy = strategy ?? new CompleteValitStrategy();

        return _strategyPicker
            .WithStrategy(selectedStrategy)
            .For(@object)
            .Validate();
    }

    private IValitRulesStrategyPicker<RegisterModel> GetValidationRules()
        => ValitRules<RegisterModel>
            .Create()
            .Ensure(m => m.Email, _=>_
                .Required()
                .Email())
            .Ensure(m => m.Password, _=>_
                .Required()
                .MinLength(10))
            .Ensure(m => m.Age, _=>_
                .IsGreaterThan(16))
            .GetAllRules();
}
```

CHAPTER 4

Validation errors

Having a validation result (so succeeded or failed) is fine but in most cases we want to provide more details to the end-user or other parts of the system. Valit provides two different mechanisms for such a thing:

- error messages
- error codes

4.1 Error messages

In order to add an error message to the specific validation rule use `WithMessage()` extension. All messages are stored inside `ErrorMessages` property in the `IValitResult` object. The example below present the usage of error messages:

```
public class RegisterModel
{
    public string Email { get; set; }
    public int Age { get; set; }
}

void ValidateModel(RegisterModel model)
{
    IValitResult result = ValitRules<RegisterModel>
        .Create()
        .Ensure(m => m.Email, _=>_
            .Required()
                .WithMessage("Email is required")
            .Email()
                .WithMessage("Email is incorrect"))
        .For(model)
        .Validate();

    if(!result.Succeeded)
    {
```

(continues on next page)

(continued from previous page)

```
        foreach(var message in result.ErrorMessages)
        {
            Console.WriteLine(message);
        }
    }
}
```

`WithMessage()` has also an overload which accepts `Func<string>` as parameter:

```
public class RegisterModel
{
    public string Email { get; set; }
    public int Age { get; set; }
}

void ValidateModel(RegisterModel model)
{
    Func<string> getRequiredMessage = () => "Email is required";
    Func<string> getIncorrectMessage = () => "Email is incorrect";

    IValitResult result = ValitRules<RegisterModel>
        .Create()
        .Ensure(m => m.Email, _=>_
            .Required()
                .WithMessage(getRequiredMessage)
            .Email()
                .WithMessage(getIncorrectMessage)
        .For(model)
        .Validate();

    if(!result.Succeeded)
    {
        foreach(var message in result.ErrorMessages)
        {
            Console.WriteLine(message);
        }
    }
}
```

Note: If no error message is specified for particular rule, Valit uses its own, default message template. The full list of default error messages can be found here.

4.2 Message provider

The above example is cool but it has one disadvantage - all messages are hardcoded. Of course it's nothing really bad for a small projects but in a typical scenario you typically store that kind of data somewhere else like database. That's why besides error messages Valit supports also **message providers**. The usage is fairly simple:

1. Create a class which implements generic interface `IValitMessageProvider<TKey>`. This interface provides one method called `GetByKey(TKey key)`.
2. Pass the instance of your provider to `ValitRules` using `WithMessageProvider()` extension.
3. Use `WithMessageKey()` extension on each rule you want to extend with custom message.

Here's an example:

```
public class RegisterModel
{
    public string Email { get; set; }
    public int Age { get; set; }
}

public class MyMessageProvider : IValitMessageProvider<string>
{
    private readonly IReadOnlyDictionary<string, string> _messages = new Dictionary
↪<string, string>
    {
        { "Key1", "Email is required" },
        { "Key2", "Email is incorrect" },
    };

    public string GetByKey(string key)
        => _messages[key];
}

void ValidateModel(RegisterModel model)
{
    IValitResult result = ValitRules<RegisterModel>
        .Create()
        .WithMessageProvider(new MyMessageProvider())
        .Ensure(m => m.Email, _=>_
            .Required()
                .WithMessageKey("Key1")
            .Email()
                .WithMessageKey("Key2"))
        .For(model)
        .Validate();

    if(!result.Succeeded)
    {
        foreach(var message in result.ErrorMessages)
        {
            Console.WriteLine(message);
        }
    }
}
```

Notice that `MyMessageProvider` uses `string` as a key type, but you can choose any other type you want.

4.3 Error codes

Valit also supports error codes as an alternative approach to messages. You can add one using `WithErrorCode()` extension on each validation rule. All error codes are stored inside `ErrorCodes` property in the `IValitResult` object. The example below present the usage of error codes:

```
public class RegisterModel
{
    public string Email { get; set; }
    public int Age { get; set; }
}
```

(continues on next page)

(continued from previous page)

```
void ValidateModel(RegisterModel model)
{
    IValitResult result = ValitRules<RegisterModel>
        .Create()
        .WithMessageProvider(new MyMessageProvider())
        .Ensure(m => m.Email, _=>_
            .Required()
                .WithErrorCode(1103)
            .Email()
                .WithErrorCode(1107))
        .For(model)
        .Validate();

    if(!result.Succeeded)
    {
        foreach(var code in result.ErrorCodes)
        {
            Console.WriteLine(code);
        }
    }
}
```

CHAPTER 5

Validation strategies

Different business logic require different validation strategies. Sometimes it might be necessary to validate the entire model when in other cases you might need only information that model has at least one property which does not fulfil the condition. Because of that, Valit offers three different behaviours for your validation process:

- Complete strategy (default)
- Fail fast startegy
- Custom strategy

All of the above can be activated using `WithStrategy()` extension on your `ValitRules` object.

5.1 Complete strategy

Complete strategy is a default one for Valit. That's why it's not required to define it's explicitly in your code. The strategy simply makes your model validated entirely. Here's an example:

```
public class RegisterModel
{
    public string Email { get; set; }
    public int Age { get; set; }
}

void ValidateModel(RegisterModel model)
{
    IValitResult result = ValitRules<RegisterModel>
        .Create()
        .WithStrategy(strategyPicker => strategyPicker.Complete) // not required
        .Ensure(m => m.Email, _=>_
            .Required()
                .WithMessage("Email is required")
            .Email()
                .WithMessage("Email is incorrect"))
```

(continues on next page)

(continued from previous page)

```
        .For(model)
        .Validate();

    if(!result.Succeeded)
    {
        foreach(var message in result.ErrorMessages)
        {
            Console.WriteLine(message); // prints both messages
        }
    }
}
```

As mentioned, because Complete strategy is a **default one** it is not required to define it explicitly.

5.2 Fail fast strategy

Fail fast strategy is handy when you only care about the overall result (so does the validation succeeded or not). In this mode, your validation process runs until first error is reached. The example below present described strategy:

```
public class RegisterModel
{
    public string Email { get; set; }
    public int Age { get; set; }
}

void ValidateModel(RegisterModel model)
{
    IValitResult result = ValitRules<RegisterModel>
        .Create()
        .WithStrategy(strategyPicker => strategyPicker.FailFast)
        .Ensure(m => m.Email, _=>_
            .Required()
                .WithMessage("Email is required")
            .Email()
                .WithMessage("Email is incorrect"))
        .For(model)
        .Validate();

    if(!result.Succeeded)
    {
        foreach(var message in result.ErrorMessages)
        {
            Console.WriteLine(message); // prints only Required message
        }
    }
}
```

In the example above, the `ErrorMessages` contains only one error message added to the `Required()` rule. Since this one is not fulfilled, Fail Fast strategy stops the validation process right after that.

5.3 Custom strategy

Of course, both Complete and Fail Fast strategies are the ones we consider as the most useful. But if they still are not enough for your needs, your can create custom strategy as well. The whole process is straightforward:

1. Create a class which implements `IValitStrategy` interface
2. Put "failing logic" inside `Fail()` method
3. Put logic on validation completion inside `Done()` method
4. Pass the instance of your custom strategy inside `WithStrategy()` extension

Let's say that our validation strategy should stop the whole process after third unfulfilled rule. The example implementation might look as follows:

```
public class RegisterModel
{
    public string Email { get; set; }
    public int Age { get; set; }
}

public class MyCustomStrategy : IValitStrategy
{
    private int _failCounter;

    public void Fail<TObject>(IValitRule<TObject> rule, IValitResult result, out bool␣
↪cancel) where TObject : class
    {
        if(! result.Succeeded)
        {
            _failCounter++;
        }

        cancel = _failCounter >= 3;
    }

    public void Done(IValitResult result)
    {
        var message = result.Succeeded? "We did it!" : "We failed!";
        Console.WriteLine(message);
    }
}

void ValidateModel(RegisterModel model)
{
    IValitResult result = ValitRules<RegisterModel>
        .Create()
        .WithStrategy(new MyCustomStrategy())
        .Ensure(m => m.Email, _=>_
            .Required()
                .WithMessage("Email is required")
            .Email()
                .WithMessage("Email is incorrect"))
        .Ensure(m => m.Age, _=>_
            .IsGreaterThan(16)
                .WithMessage("Age must be greater than 16")
            .IsLessThan(50)
                .WithMessage("Age must be less than 50"))
```

(continues on next page)

(continued from previous page)

```
        .For(model)
        .Validate();

    if(!result.Succeeded)
    {
        foreach(var message in result.ErrorMessages)
        {
            Console.WriteLine(message); // Doesn't print last message
        }
    }
}
```

CHAPTER 6

Release Notes

All release notes could be found on GitHub.

CHAPTER 7

Contributor Guide

If you want to help us develop Valit, please visit our contributor's guide on GitHub.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search