



Validate! (or fail...)

Wil de Bruin

e-mail: wil@site4u.nl

blog: <https://shiftinsert.nl>

Twitter: @wpdebruin

Who am I

- Wil de Bruin – Software / system engineer
- Wageningen, The Netherlands
- Graduated in Environmental Sciences
- Research microbiologist (1987)
- CTO – Founder Site4u (1994)
- CFML since Allaire ColdFusion 1.5
- Coldfusion training, consultancy, software development and hosting
- Coldbox user since 2008



Agenda

- Why validate?
- Installation and configuration
- Validators and constraints
- Validation and your validation results
- Validation examples: UDF and CustomValidators
- Message replacements and internationalization (cbi18n)
- The good, the bad.. and possible improvements

Why validate?

- Database integrity rules
- Model rules
- API limitations
- Hack attempts
- Provide user-friendly validation error messages



Installation and configuration

Installation

Box install cbvalidation

Or drop cbvalidation files in your modules folder

Cbvalidation will register the following methods in handlers/views/layouts/interceptors

- Validate()
- ValidateOrFail()
- getValidationManager()

Configuration

```
validation = {
    // manager = "models.utils.YourOwnValidationManager",
    // The shared constraints for your application.
    sharedConstraints = {
        FormSharedMailBoxes = {
            "BoxName" = {
                "required"=true,
                "validator" = "id:SharedMailBoxValidator",
                "regex" = "^[a-zA-Z0-9_.-]+$"
            },
            "Password" = {
                "required" = true,
                "validator" = "id:PasswordValidator"
            }
        },
        NewServerWizard = {
            "Name" = {
                required = true,
                validator: "UniqueValidator@cborm"
            },
            // more constraints ...
        }
    }
}
```

Validators and constraints

Constraint

The state of being restricted or confined within
prescribed bounds.

How to define constraints

```
// Define the field by name
// The contents are the constraints
fieldName1 = {
    validator1 = validationData,
    validator2 = validationData
},
fieldName2 = {
    validator1 = validationData,
    validator2 = validationData
}
```

Where to define your constraints

- In your config/Coldbox.cfc configuration file
- In your model component
- On the fly as a struct, most of the time in your handler.

Standard validators

- accepted
 - alpha
 - discrete
 - inList
 - max
 - method
 - min
 - range
 - regex
 - required
- requiredIf
 - requiredUnless
 - sameAsNoCase
 - sameAs
 - size
 - type
 - udf
 - unique
 - validator

<https://coldbox-validation.ortusbooks.com/overview/valid-constraints>

Validation examples

- Validate request scope (or struct) with a shared constraint
- Validate request scope with a-la-carte constraint
- Validate model



Anatomy of a validate() call (1/3)

- validate(target = rc)
- validate(target =myModel)
- validate (target = rc, constraints = { name="required" })
- validate(target = rc, constraints = "mySpecialSharedConstraint")
- Validate(target = myModel, constraints = { name="required" })

Anatomy of a validate() call (2/3)

Validate arguments:

- @target An object or structure to validate
- @fields The fields to validate on the target. By default, it validates on all fields
- @constraints A structure of constraint rules or the name of the shared constraint rules to use for validation
- @locale The i18n locale to use for validation messages
- @excludeFields The fields to exclude from the validation
- @includeFields The fields to include in the validation
- @profiles If passed, a list of profile names to use for validation constraints



Anatomy of a validate() call (3/3)

- If target is not an object: convert struct to some general Object
- Find your constraints!
 - Constraints param a struct() -> return constraints param
 - Constraints param a string -> return shared constraints from config
 - All other cases (including no constraint param) -> return constraints from target object
- Create includeFields param from profiles
- Loop over all constraints
 - If there are includefields, do not process if field is not in includeFields
 - If there are excludefields do not process if field is in excludeFields
- Return a result object (with all failed validations) or an empty Result object on success
- ValidateOrFail processing is similar, but throws an error on failed validations
- ValidateOrFail returns an object or a struct if validation passes

How to validate

Validate your request

Validate request collection



(populate your model)



SAVE

Validate your model

Populate your model



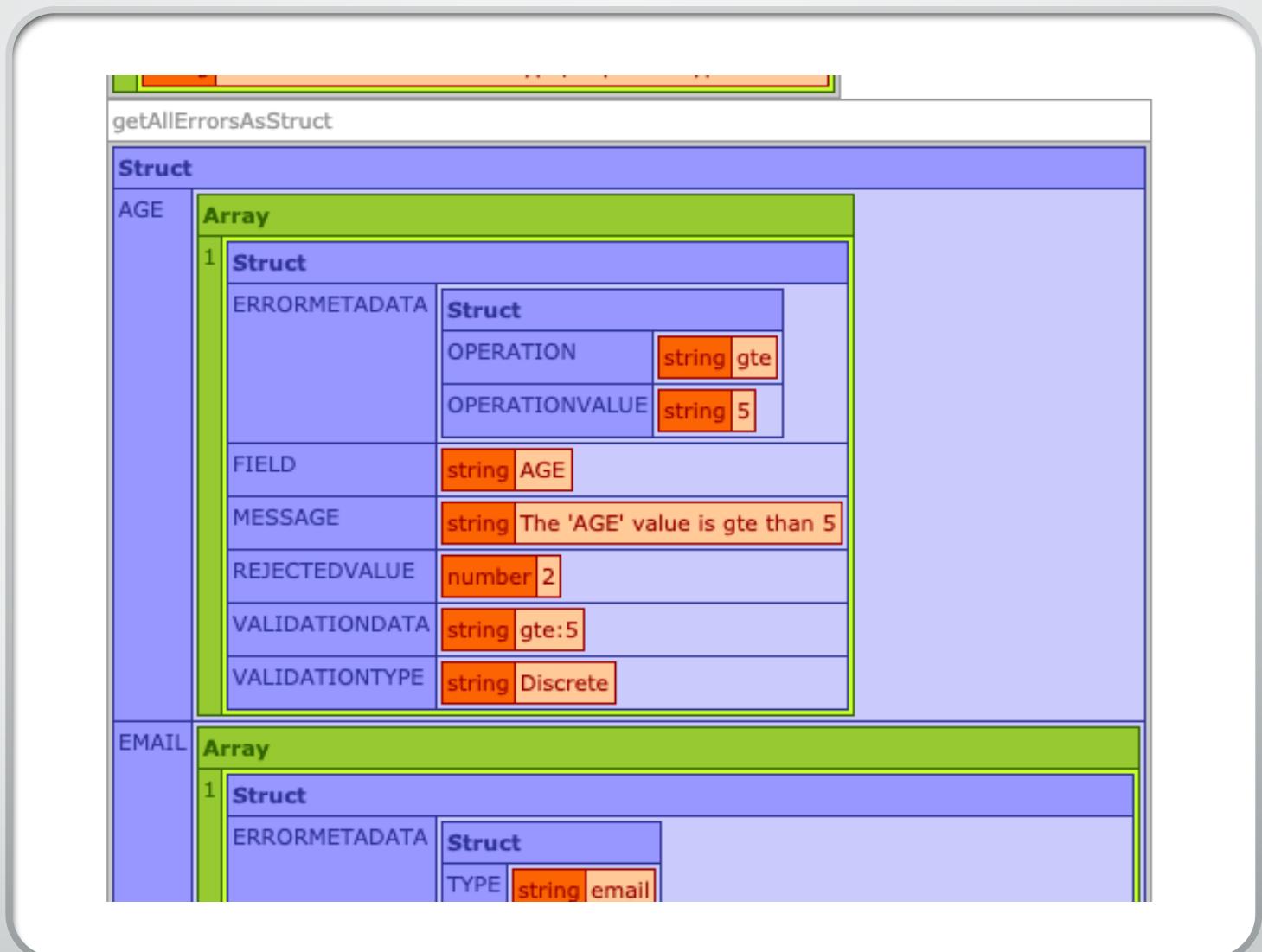
Validate model



SAVE

Validation result object

- getResultMetadata()
- getFieldErrors([field])
- getAllErrors([field])
- getAllErrorsAsJSON([field])
- getAllErrorsAsStruct([field])
- getErrorCount([field])
- hasErrors([field])
- getErrors()



Validation examples

Custom messages and message replacements



The power of Method, UDF and CustomValidators

MethodValidator

- The methodName will be called on the target object and it will pass in the target, validationData, targetValue. It must return a boolean response: **true** = pass, **false** = fail.
- Validation of related properties in target.
- But: no parameters passed, so limited use.
- Only one METHOD validator per field.

UDF validator

- Very flexible
 - Input from validation Target
 - Input from request scope
- Very suitable for a-la-carte validation in handlers.
- Not very DRY.

Custom validators (1/2)

- Very flexible
- Reusable
- Easy integration with other libraries, e.g java
- Syntax: `fieldname = { Validator = "MyOwnValidator"}`
- Or more flexible(easy to pass extra parameters and more validators per fieldname: `fieldname = { MyOwnValidator = { someLimit=10, param="abc"}}`

Custom validators (2/2)

How to create a custom validator

- Init function (set validator name)
- getName()
- Validate()
 - @validationResult The result object of the validation
 - @target The target object to validate on
 - @field The field on the target object to validate on
 - @targetValue The target value to validate
 - @validationData The validation data the validator was created with
- Create validation tests and return true on Validation → exit validator
- Create arguments for error message (message, field, ValidationType, rejectedValue, validationData)
- Add error to ValidationResult and return false

Foutmeldingen in het Nederlands

a.k.a Customizing your error messages
AND
Internationalization / cbi18n

Customizing your error messages

- customMessages by adding ...Message to the constraint definition, e.g
 - "name" = {
 - required = true,
 - requiredMessage = "This is a custom message for {field}"
 - }
- Several message replacements, e.g: {targetValue}, {field} and several Validator specific replacements e.g {range}, {min}, {max}
- Simple cbi18n integration in a la carte constraints in handlers
 - "name" = {
 - required = true,
 - requiredMessage = getResource("someMessageKey@myAlias")
 - }
- Keys in resource file based on **Target.Field.Validator**

Limitations & future enhancements

- Old validators use strings for constraints, newer validators use structs. This breaks custom messages for CustomValidators and some newer validators.
- It also breaks ALL translations for CustomValidators and some newer validators.
- You can't customize default messages
- Internationalization support is very limited
 - If you want to translate default messages, you have to specify it for every single constraint definition.
 - You can only specify ONE resource file for translations, so no resource aliases possible. This will break code in other places
 - You always have to specify locale = something in your validate calls
 - Cbi18n should be part of cbvalidation and cbvalidation should listen to the current locale.

Thank you!

Questions

Slides: <https://shiftinsert.nl/itb2020>

E-mail: wil@site4u.nl

Slack: @wil-site4u

Twitter: @wpdebruin