ozzo-validation

Description

ozzo-validation is a Go package that provides configurable and extensible data validation capabilities. It has the following features:

- use normal programming constructs rather than error-prone struct tags to specify how data should be validated.
- can validate data of different types, e.g., structs, strings, byte slices, slices, maps, arrays.
- can validate custom data types as long as they implement the `Validatable` interface.
- can validate data types that implement the `sql.Valuer` interface (e.g. `sql.NullString`).
- customizable and well-formatted validation errors.
- provide a rich set of validation rules right out of box.
- extremely easy to create and use custom validation rules.

Requirements

Go 1.8 or above.

Getting Started

The ozzo-validation package mainly includes a set of validation rules and two validation methods. You use validation rules to describe how a value should be considered valid, and you call either `validation.Validate()` or `validation.ValidateStruct()` to validate the value.

Installation

Run the following command to install the package:

```
go get github.com/go-ozzo/ozzo-validation
go get github.com/go-ozzo/ozzo-validation/is
```

Validating a Simple Value

For a simple value, such as a string or an integer, you may use `validation.Validate()` to validate it. For example,

```
package main

import (
    "fmt"

    "github.com/go-ozzo/ozzo-validation"
    "github.com/go-ozzo/ozzo-validation/is"
)
```

```
func main() {
    data := "example"
    err := validation.Validate(data,
        validation.Required,       // not empty
        validation.Length(5, 100), // length between 5 and 100
        is.URL,                    // is a valid URL
    )
    fmt.Println(err)
    // Output:
    // must be a valid URL
}
```

The method `validation.Validate()` will run through the rules in the order that they are listed. If a rule fails the validation, the method will return the corresponding error and skip the rest of the rules. The method will return nil if the value passes all validation rules.

Validating a Struct

For a struct value, you usually want to check if its fields are valid. For example, in a RESTful application, you may unmarshal the request payload into a struct and then validate the struct fields. If one or multiple fields are invalid, you may want to get an error describing which fields are invalid. You can use `validation.ValidateStruct()` to achieve this purpose. A single struct can have rules for multiple fields, and a field can be associated with multiple rules. For example,

```
package main

import (
    "fmt"
    "regexp"

    "github.com/go-ozzo/ozzo-validation"
    "github.com/go-ozzo/ozzo-validation/is"
)

type Address struct {
    Street string
    City   string
    State  string
    Zip    string
}

func (a Address) Validate() error {
    return validation.ValidateStruct(&a,
        // Street cannot be empty, and the length must between 5 and 50
        validation.Field(&a.Street, validation.Required, validation.Length(5, 50)),
        // City cannot be empty, and the length must between 5 and 50
        validation.Field(&a.City, validation.Required, validation.Length(5, 50)),
        // State cannot be empty, and must be a string consisting of two letters in
upper case
        validation.Field(&a.State, validation.Required,
validation.Match(regexp.MustCompile("^[A-Z]{2}$"))),
```

```
        // State cannot be empty, and must be a string consisting of five digits
        validation.Field(&a.Zip, validation.Required,
validation.Match(regexp.MustCompile("^[0-9]{5}$"))),
    )
}

func main() {
    a := Address{
        Street: "123",
        City:   "Unknown",
        State:  "Virginia",
        Zip:    "12345",
    }

    err := a.Validate()
    fmt.Println(err)
    // Output:
    // Street: the length must be between 5 and 50; State: must be in a valid
format.
}
```

Note that when calling `validation.ValidateStruct` to validate a struct, you should pass to the method a pointer to the struct instead of the struct itself. Similarly, when calling `validation.Field` to specify the rules for a struct field, you should use a pointer to the struct field.

When the struct validation is performed, the fields are validated in the order they are specified in `ValidateStruct`. And when each field is validated, its rules are also evaluated in the order they are associated with the field. If a rule fails, an error is recorded for that field, and the validation will continue with the next field.

Validation Errors

The `validation.ValidateStruct` method returns validation errors found in struct fields in terms of `validation.Errors` which is a map of fields and their corresponding errors. Nil is returned if validation passes.

By default, `validation.Errors` uses the struct tags named `json` to determine what names should be used to represent the invalid fields. The type also implements the `json.Marshaler` interface so that it can be marshaled into a proper JSON object. For example,

```
type Address struct {
    Street string `json:"street"`
    City   string `json:"city"`
    State  string `json:"state"`
    Zip    string `json:"zip"`
}

// ...perform validation here...

err := a.Validate()
b, _ := json.Marshal(err)
fmt.Println(string(b))
// Output:
```

```
// {"street":"the length must be between 5 and 50","state":"must be in a valid
format"}
```

You may modify `validation.ErrorTag` to use a different struct tag name.

If you do not like the magic that `ValidateStruct` determines error keys based on struct field names or corresponding tag values, you may use the following alternative approach:

```
c := Customer{
    Name:  "Qiang Xue",
    Email: "q",
    Address: Address{
        State:  "Virginia",
    },
}

err := validation.Errors{
    "name": validation.Validate(c.Name, validation.Required, validation.Length(5,
20)),
    "email": validation.Validate(c.Name, validation.Required, is.Email),
    "zip": validation.Validate(c.Address.Zip, validation.Required,
validation.Match(regexp.MustCompile("^[0-9]{5}$"))),
}.Filter()
fmt.Println(err)
// Output:
// email: must be a valid email address; zip: cannot be blank.
```

In the above example, we build a `validation.Errors` by a list of names and the corresponding validation results. At the end we call `Errors.Filter()` to remove from `Errors` all nils which correspond to those successful validation results. The method will return nil if `Errors` is empty.

The above approach is very flexible as it allows you to freely build up your validation error structure. You can use it to validate both struct and non-struct values. Compared to using `ValidateStruct` to validate a struct, it has the drawback that you have to redundantly specify the error keys while `ValidateStruct` can automatically find them out.

Internal Errors

Internal errors are different from validation errors in that internal errors are caused by malfunctioning code (e.g. a validator making a remote call to validate some data when the remote service is down) rather than the data being validated. When an internal error happens during data validation, you may allow the user to resubmit the same data to perform validation again, hoping the program resumes functioning. On the other hand, if data validation fails due to data error, the user should generally not resubmit the same data again.

To differentiate internal errors from validation errors, when an internal error occurs in a validator, wrap it into `validation.InternalError` by calling `validation.NewInternalError()`. The user of the validator can then check if a returned error is an internal error or not. For example,

```
if err := a.Validate(); err != nil {
    if e, ok := err.(validation.InternalError); ok {
        // an internal error happened
```

```
        fmt.Println(e.InternalError())
    }
}
```

Validatable Types

A type is validatable if it implements the `validation.Validatable` interface.

When `validation.Validate` is used to validate a validatable value, if it does not find any error with the given validation rules, it will further call the value's `Validate()` method.

Similarly, when `validation.ValidateStruct` is validating a struct field whose type is validatable, it will call the field's `Validate` method after it passes the listed rules.

In the following example, the `Address` field of `Customer` is validatable because `Address` implements `validation.Validatable`. Therefore, when validating a `Customer` struct with `validation.ValidateStruct`, validation will "dive" into the `Address` field.

```
type Customer struct {
    Name    string
    Gender  string
    Email   string
    Address Address
}

func (c Customer) Validate() error {
    return validation.ValidateStruct(&c,
        // Name cannot be empty, and the length must be between 5 and 20.
        validation.Field(&c.Name, validation.Required, validation.Length(5, 20)),
        // Gender is optional, and should be either "Female" or "Male".
        validation.Field(&c.Gender, validation.In("Female", "Male")),
        // Email cannot be empty and should be in a valid email format.
        validation.Field(&c.Email, validation.Required, is.Email),
        // Validate Address using its own validation rules
        validation.Field(&c.Address),
    )
}

c := Customer{
    Name:  "Qiang Xue",
    Email: "q",
    Address: Address{
        Street: "123 Main Street",
        City:   "Unknown",
        State:  "Virginia",
        Zip:    "12345",
    },
}

err := c.Validate()
fmt.Println(err)
```

```
// Output:
// Address: (State: must be in a valid format.); Email: must be a valid email
address.
```

Sometimes, you may want to skip the invocation of a type's `Validate` method. To do so, simply associate a `validation.Skip` rule with the value being validated.

Maps/Slices/Arrays of Validatables

When validating a map, slice, or array, whose element type implements the `validation.Validatable` interface, the `validation.Validate` method will call the `Validate` method of every non-nil element. The validation errors of the elements will be returned as `validation.Errors` which maps the keys of the invalid elements to their corresponding validation errors. For example,

```
addresses := []Address{
    Address{State: "MD", Zip: "12345"},
    Address{Street: "123 Main St", City: "Vienna", State: "VA", Zip: "12345"},
    Address{City: "Unknown", State: "NC", Zip: "123"},
}
err := validation.Validate(addresses)
fmt.Println(err)
// Output:
// 0: (City: cannot be blank; Street: cannot be blank.); 2: (Street: cannot be
blank; Zip: must be in a valid format.).
```

When using `validation.ValidateStruct` to validate a struct, the above validation procedure also applies to those struct fields which are map/slices/arrays of validatables.

Pointers

When a value being validated is a pointer, most validation rules will validate the actual value pointed to by the pointer. If the pointer is nil, these rules will skip the validation.

An exception is the `validation.Required` and `validation.NotNil` rules. When a pointer is nil, they will report a validation error.

Types Implementing `sql.Valuer`

If a data type implements the `sql.Valuer` interface (e.g. `sql.NullString`), the built-in validation rules will handle it properly. In particular, when a rule is validating such data, it will call the `Value()` method and validate the returned value instead.

Required vs. Not Nil

When validating input values, there are two different scenarios about checking if input values are provided or not.

In the first scenario, an input value is considered missing if it is not entered or it is entered as a zero value (e.g. an empty string, a zero integer). You can use the `validation.Required` rule in this case.

In the second scenario, an input value is considered missing only if it is not entered. A pointer field is usually used in this case so that you can detect if a value is entered or not by checking if the pointer is nil or not. You can use the `validation.NotNil` rule to ensure a value is entered (even if it is a zero value).

Embedded Structs

The `validation.ValidateStruct` method will properly validate a struct that contains embedded structs. In particular, the fields of an embedded struct are treated as if they belong directly to the containing struct. For example,

```
type Employee struct {
    Name string
}

func ()

type Manager struct {
    Employee
    Level int
}

m := Manager{}
err := validation.ValidateStruct(&m,
    validation.Field(&m.Name, validation.Required),
    validation.Field(&m.Level, validation.Required),
)
fmt.Println(err)
// Output:
// Level: cannot be blank; Name: cannot be blank.
```

In the above code, we use `&m.Name` to specify the validation of the `Name` field of the embedded struct `Employee`. And the validation error uses `Name` as the key for the error associated with the `Name` field as if `Name` a field directly belonging to `Manager`.

If `Employee` implements the `validation.Validatable` interface, we can also use the following code to validate `Manager`, which generates the same validation result:

```
func (e Employee) Validate() error {
    return validation.ValidateStruct(&e,
        validation.Field(&e.Name, validation.Required),
    )
}

err := validation.ValidateStruct(&m,
    validation.Field(&m.Employee),
    validation.Field(&m.Level, validation.Required),
)
fmt.Println(err)
// Output:
// Level: cannot be blank; Name: cannot be blank.
```

Built-in Validation Rules

The following rules are provided in the `validation` package:

- `In(...interface{})` : checks if a value can be found in the given list of values.
- `Length(min, max int)` : checks if the length of a value is within the specified range. This rule should only be used for validating strings, slices, maps, and arrays.
- `RuneLength(min, max int)` : checks if the length of a string is within the specified range. This rule is similar as `Length` except that when the value being validated is a string, it checks its rune length instead of byte length.
- `Min(min interface{})` and `Max(max interface{})` : checks if a value is within the specified range. These two rules should only be used for validating int, uint, float and time.Time types.
- `Match(*regexp.Regexp)` : checks if a value matches the specified regular expression. This rule should only be used for strings and byte slices.
- `Date(layout string)` : checks if a string value is a date whose format is specified by the layout. By calling `Min()` and/or `Max()` , you can check additionally if the date is within the specified range.
- `Required` : checks if a value is not empty (neither nil nor zero).
- `NotNil` : checks if a pointer value is not nil. Non-pointer values are considered valid.
- `NilOrNotEmpty` : checks if a value is a nil pointer or a non-empty value. This differs from `Required` in that it treats a nil pointer as valid.
- `Skip` : this is a special rule used to indicate that all rules following it should be skipped (including the nested ones).
- `MultipleOf` : checks if the value is a multiple of the specified range.

The `is` sub-package provides a list of commonly used string validation rules that can be used to check if the format of a value satisfies certain requirements. Note that these rules only handle strings and byte slices and if a string or byte slice is empty, it is considered valid. You may use a `Required` rule to ensure a value is not empty. Below is the whole list of the rules provided by the `is` package:

- `Email` : validates if a string is an email or not
- `URL` : validates if a string is a valid URL
- `RequestURL` : validates if a string is a valid request URL
- `RequestURI` : validates if a string is a valid request URI
- `Alpha` : validates if a string contains English letters only (a-zA-Z)
- `Digit` : validates if a string contains digits only (0-9)
- `Alphanumeric` : validates if a string contains English letters and digits only (a-zA-Z0-9)
- `UTFLetter` : validates if a string contains unicode letters only
- `UTFDigit` : validates if a string contains unicode decimal digits only
- `UTFLetterNumeric` : validates if a string contains unicode letters and numbers only
- `UTFNumeric` : validates if a string contains unicode number characters (category N) only
- `LowerCase` : validates if a string contains lower case unicode letters only
- `UpperCase` : validates if a string contains upper case unicode letters only
- `Hexadecimal` : validates if a string is a valid hexadecimal number
- `HexColor` : validates if a string is a valid hexadecimal color code
- `RGBColor` : validates if a string is a valid RGB color in the form of rgb(R, G, B)
- `Int` : validates if a string is a valid integer number
- `Float` : validates if a string is a floating point number
- `UUIDv3` : validates if a string is a valid version 3 UUID
- `UUIDv4` : validates if a string is a valid version 4 UUID
- `UUIDv5` : validates if a string is a valid version 5 UUID
- `UUID` : validates if a string is a valid UUID
- `CreditCard` : validates if a string is a valid credit card number

- `ISBN10` : validates if a string is an ISBN version 10
- `ISBN13` : validates if a string is an ISBN version 13
- `ISBN` : validates if a string is an ISBN (either version 10 or 13)
- `JSON` : validates if a string is in valid JSON format
- `ASCII` : validates if a string contains ASCII characters only
- `PrintableASCII` : validates if a string contains printable ASCII characters only
- `Multibyte` : validates if a string contains multibyte characters
- `FullWidth` : validates if a string contains full-width characters
- `HalfWidth` : validates if a string contains half-width characters
- `VariableWidth` : validates if a string contains both full-width and half-width characters
- `Base64` : validates if a string is encoded in Base64
- `DataURI` : validates if a string is a valid base64-encoded data URI
- `E164` : validates if a string is a valid E164 phone number (+19251232233)
- `CountryCode2` : validates if a string is a valid ISO3166 Alpha 2 country code
- `CountryCode3` : validates if a string is a valid ISO3166 Alpha 3 country code
- `DialString` : validates if a string is a valid dial string that can be passed to Dial()
- `MAC` : validates if a string is a MAC address
- `IP` : validates if a string is a valid IP address (either version 4 or 6)
- `IPv4` : validates if a string is a valid version 4 IP address
- `IPv6` : validates if a string is a valid version 6 IP address
- `Subdomain` : validates if a string is valid subdomain
- `Domain` : validates if a string is valid domain
- `DNSName` : validates if a string is valid DNS name
- `Host` : validates if a string is a valid IP (both v4 and v6) or a valid DNS name
- `Port` : validates if a string is a valid port number
- `MongoID` : validates if a string is a valid Mongo ID
- `Latitude` : validates if a string is a valid latitude
- `Longitude` : validates if a string is a valid longitude
- `SSN` : validates if a string is a social security number (SSN)
- `Semver` : validates if a string is a valid semantic version

Customizing Error Messages

All built-in validation rules allow you to customize error messages. To do so, simply call the `Error()` method of the rules. For example,

```
data := "2123"
err := validation.Validate(data,
    validation.Required.Error("is required"),
    validation.Match(regexp.MustCompile("^[0-9]{5}$")).Error("must be a string with
five digits"),
)
fmt.Println(err)
// Output:
// must be a string with five digits
```

Creating Custom Rules

Creating a custom rule is as simple as implementing the `validation.Rule` interface. The interface contains a single method as shown below, which should validate the value and return the validation error, if any:

```
// Validate validates a value and returns an error if validation fails.
Validate(value interface{}) error
```

If you already have a function with the same signature as shown above, you can call `validation.By()` to turn it into a validation rule. For example,

```
func checkAbc(value interface{}) error {
    s, _ := value.(string)
    if s != "abc" {
        return errors.New("must be abc")
    }
    return nil
}

err := validation.Validate("xyz", validation.By(checkAbc))
fmt.Println(err)
// Output: must be abc
```

Rule Groups

When a combination of several rules are used in multiple places, you may use the following trick to create a rule group so that your code is more maintainable.

```
var NameRule = []validation.Rule{
    validation.Required,
    validation.Length(5, 20),
}

type User struct {
    FirstName string
    LastName  string
}

func (u User) Validate() error {
    return validation.ValidateStruct(&u,
        validation.Field(&u.FirstName, NameRule...),
        validation.Field(&u.LastName, NameRule...),
    )
}
```

In the above example, we create a rule group `NameRule` which consists of two validation rules. We then use this rule group to validate both `FirstName` and `LastName`.

Credits

The `is` sub-package wraps the excellent validators provided by the govalidator package.