Validation Data

Third party validation library to handle validation

Pariksha : Scala validation library

Recently i started working on a library that does validation on scala types.

The original aim was to write a library that was could be used for data validation on incoming api requests in a web application . Then i saw some other scala validation libraries like Accord and octopus and i thought to extend the ideas expressed there. The source code for Pariksha (which means test in Hindi) can be found at https://github.com/ayushworks/pariksha

The protagonist of our story is the Validator[T] trait which validates instances of T by using a list of Validation[T].

Consider a simple case class

case class Employee(name: String, age: Int)

We can define a list of Validation for this type

import pariksha.dsl_

implicit val validations = validator[Employee]

.check(_.name.nonEmpty, "name must not be empty")

.check(_.age > 18, "age must be above 18")

.check(_.name != "Bob Vance", "He owns Vance Refrigeration and is not an employee")

And then we can validate any instance of Employee type. All we need is Validator[Employee] implicitly in scope

import pareeskha.dsl_

val employee = Employee("Jim Halpert", 30)

employee.validate

validate returns a ValidationResult which can have two possible values

Valid

Invalid

Everybody knows Jenna Fischer from the office!

```
val beesly = Employee("Pam Beesly", 28)

beesly.validate == Valid(beesly)
```

And for an invalid employee

```
val bob = Employee("Bob Vance", 45)

bob.validate == Invalid(bob, List(ValidationError("He owns Vance Refrigeration and is not an employee")))
```

Nested Validations

When we have a type that contains another type, and we already have a Validator for the nested type, we can use the existing validator and delegate to that.

```
case class Manager(name: String, age: Int)

case class Office(manager: Manager, region: String)
```

We could define validations for Manager to be used in multiple places

```
val validations = validator[Manager]

  .check(_.name.nonEmpty, msgNameEmpty)

  .check(_.age > 25, msgAgeInvalid)
```

Then we we can define validations for the Office class and use the previous validator automatically for the manager field, assuming it is available as an implicit in the current scope.

Note, how we use the validate method on the contained type.

```
val validations = validator[Office]

          .check(_.manager.validate)

          .check(_.region.nonEmpty, msgRegionNonEmpty)

val validManager = Manager("Michael", 35)

val office = Office(validManager, "Scranton")
```

```
office.validate == Valid(office)
```

Fail Fast Validations

Sometimes it is desirable to not run all validations exhaustively but rather stop on first failed validation.

We can use the validateFailFast method on a type T . The requirements remain the same with a presence of Validator[T] needed.

```
val manager = Manager("", 18)

val validations = validator[Manager]

   .check(_.name.nonEmpty, msgNameEmpty)

   .check(_.age > 25, msgAgeInvalid)


manager.validate == Invalid(manager, List(

        ValidationError(Manager.msgNameEmpty),

        ValidationError(Manager.msgAgeInvalid)

      ))

manager.validateFailFast == Invalid(manager, List(

                                     ValidationError(Manager.msgNameEmpty))
```

The second check is not even called in this case as the first one had failed. This is useful when the validations are resource/time consuming and we would like to stop at the first sign of problems