

# Using Either[E, A] And MapN For Data Validation

## Quick Intro To Either[E, A] type

In addition to throwing exceptions to handle errors, scala also offers the **Either[E, A]** data type to perform the equivalent task of handling errors as well.

The email validation method we had:

```
def validateEmail(string: String): Unit =  
  if (string == null)  
    throw new Throwable("Email is null")  
  else  
    val split = string.split("@")  
    if (split.size != 2)  
      throw new Throwable(s"Email ${string} is malformed")  
    else ()
```

Can now be redefined to return an **Either[Error, Unit]**. Additionally, I introduce an, IMO, more readable way to check for null fields:

```
def validateEmailEither(string: String): Either[Throwable, Unit] = string match  
  case null =>  
    Left(Error("Email is null"))  
  case _ =>  
    val split = string.split("@")  
    if (split.size != 2)  
      Left(Error(s"Email ${string} is malformed"))  
    else Right()  
  
validateEmail("hernan#email.com")  
// Exception in thread "main" java.lang.Error: Email hernan#email.com is malformed
```

```
// The following throws no exception
val validated: Either[Throwable, Unit] = validateEmailEither("hernan#email.com")
// We can then selectively handle the error when it occurs while staying in functional
programming paradigm

// The error handling uses the same runtime stack as "normal code" now
validated match
  case Left(f) => println(s"validation fail: ${f.getMessage}")
// validation fail: Email hernan#email.com is malformed
```

## Building Validated Case Classes For Cleaner Code

Perhaps, you can't see yet the benefit of using **Either[A, E]** yet with such small example.

Here we are using case classes to return validated fields instead of still using String types.

```
final case class SSN private(area: Int, group: Int, serial: Int)
object SSN:
  def fromString(string: String): Either[Throwable, SSN] = string match
    case null =>
      Left(Throwable("Social security is null"))
    case _ =>
      val split = string.split("-")
      if (split.size != 3)
        Left(Throwable(s"Three different sets of digits expected but ${split.size}
found"))
      else if (split(0).filter(_ isDigit).isEmpty)
        Left(Throwable(s"No digits found in area position '${string}'"))
      else if (split(1).filter(_ isDigit).isEmpty)
        Left(Throwable(s"No digits found in group position '${string}'"))
      else if (split(2).filter(_ isDigit).isEmpty)
        Left(Throwable(s"No digits found in serial position '${string}'"))
      else if (split(0).filter(!_ isDigit).nonEmpty)
        Left(Throwable(s"Invalid digit found in area position '${string}'"))
      else if (split(1).filter(!_ isDigit).nonEmpty)
        Left(Throwable(s"Invalid digit found in group position '${string}'"))
      else if (split(2).filter(!_ isDigit).nonEmpty)
        Left(Throwable(s"Invalid digit found in serial position '${string}'"))
      else
        Right(SSN(area = split(0).toInt, group = split(1).toInt, serial = split(2).toInt))
```

```

final case class Email private(user: String, domain: String)
object Email:
  def fromString(string: String): Either[Throwable, Email] = string match
    case null =>
      Left(Throwable("Email is null"))
    case _ =>
      val split = string.split("@")
      if (split.size != 2)
        Left(Throwable(s"Email '${string}' is malformed"))
      else
        Right(Email(user = split(0), domain = split(1)))

```

## Putting It Together With For Comprehensions

Now we can validate our fields while expressing the "happy path" clearly eg

```

final case class Employee(ssn: SSN, email: Email)
val employee: Either[Throwable, Employee] = for
  email <- Email.fromString("hernan@email.com")
  ssn <- SSN.fromString("111-11-1111")
yield Employee(ssn = ssn, email = email)

val employeeBadEmail: Either[Throwable, Employee] = for
  email <- Email.fromString("hernan#email.com")
  ssn <- SSN.fromString("111-11-1111")
yield Employee(ssn = ssn, email = email)

val employeeBadSsn: Either[Throwable, Employee] = for
  email <- Email.fromString("hernan@email.com")
  ssn <- SSN.fromString("11111-1111")
yield Employee(ssn = ssn, email = email)

// Again, we handle errors using normal control data flow
employee match {
  case Right(o) => println(s"employee: Validated employee: $o")
  case Left(e) => println(s"employee: Validation error: $e")
}

// employee: Validated employee: Employee(SSN(111,11,1111),Email(hernan,email.com))

```

```

employeeBadEmail match {
  case Right(o) => println(s"employeeBadEmail: Validated employee: $o")
  case Left(e) => println(s"employeeBadEmail: Validation error: $e")
}
// employeeBadEmail: Validation error: java.lang.Throwable: Email 'hernan#email.com' is
malformed

employeeBadSsn match {
  case Right(o) => println(s"employeeBadSsn: Validated employee: $o")
  case Left(e) => println(s"employeeBadSsn: Validation error: $e")
}
// employeeBadSsn: Validation error: java.lang.Throwable: Three different sets of digits
expected but 2 found

```

## But There Is A Problem With For Comprehensions: It's Sequential Nature

I love **for comprehensions** because enables me to clearly express the happy path while handling potential errors. But not everything is perfect here. The **sequential nature of for comprehensions doesn't allow us to catch all errors** if that is what we need. eg, how do we know if both fields ssn and email are incorrect?

```

val employeeBadEmailAndBadSsn: Either[Throwable, Employee] = for
  email <- Email.fromString("hernan#email.com")
  ssn <- SSN.fromString("11111-1111")
yield Employee(ssn = ssn, email = email)

// Two fields are invalid but only one will be evaluated. Therefore, you will only be able to
collect one error
employeeBadEmailAndBadSsn match {
  case Right(o) => println(s"employeeBadEmailAndBadSsn: Validated employee: $o")
  case Left(e) => println(s"employeeBadEmailAndBadSsn: Validation error: $e")
}
// employeeBadEmailAndBadSsn: Validation error: java.lang.Throwable: Email 'hernan#email.com'
is malformed

```

The printed errors above only shows the first invalid field, the email. However, the format of the ssn is also incorrect. But due to the sequential nature of for comprehensions, all computations after the first error are cancelled and email doesn't get a chance to get evaluated.

For comprehensions are useful for the common use case when you need to fail fast with no need

to evaluate other bad fields.

For comprehensions will cancel the next steps when the first error is generated. This means it's not equipped to evaluate multiple fields.

## Using **Either[List[Throwable], A]** instead of **Either[Throwable, A]**

If we want to collect many validation errors, we first need a data type able to handle them. Enter **Either[List[Throwable], A]**

```
// Convenience type alias
type EitherError[A] = Either[List[Throwable], A]

// Usage
val goodText1: Either[List[Throwable], String] = Right("good text")
val badText1: Either[List[Throwable], String] = Left(List(Throwable("bad text found")))
// Or
val goodText2: EitherError[String] = Right("good text")
val badText2: EitherError[String] = Left(List(Throwable("bad text found")))
```

In the example above I provided a simplified version of **Either[List[Throwable], A]** to reduce verbosity, alias type **EitherError[A]**.

Here is how our case classes and builders would look after modifying Either[E, A]

```
final case class SSN2 private(area: Int, group: Int, serial: Int)
object SSN2:
  def fromString(string: String): Either[List[Throwable], SSN2] = string match
    case null =>
      Left(List(Throwable("Social security is null")))
    case _ =>
      val split = string.split("-")
      if (split.size != 3)
        Left(List(Throwable(s"Three different sets of digits expected but ${split.size} found")))
      else if (split(0).filter(_ isDigit).isEmpty)
        Left(List(Throwable(s"No digits found in area position '${string}'")))
      else if (split(1).filter(_ isDigit).isEmpty)
        Left(List(Throwable(s"No digits found in group position '${string}'")))
      else if (split(2).filter(_ isDigit).isEmpty)
```

```

        Left(List(Throwable(s"No digits found in serial position '${string}'")))
    else if (split(0).filter(!_._isDigit).nonEmpty)
        Left(List(Throwable(s"Invalid digit found in area position '${string}'")))
    else if (split(1).filter(!_._isDigit).nonEmpty)
        Left(List(Throwable(s"Invalid digit found in group position '${string}'")))
    else if (split(2).filter(!_._isDigit).nonEmpty)
        Left(List(Throwable(s"Invalid digit found in serial position '${string}'")))
    else
        Right(SSN2(area = split(0).toInt, group = split(1).toInt, serial = split(2).toInt))

final case class Email2 private(user: String, domain: String)
object Email2:
    def fromString(string: String): Either[List[Throwable], Email2] = string match
        case null =>
            Left(List(Throwable("Email is null")))
        case _ =>
            val split = string.split("@")
            if (split.size != 2)
                Left(List(Throwable(s"Email '${string}' is malformed")))
            else
                Right(Email2(user = split(0), domain = split(1)))

```

Since I don't like the ugly nested **Left(List(Throwable**, I created a [convenience method wrapper](#) called **LeftThrowable** that you can find in [this repo](#). This is how the code above will look like:

```

final case class SSN3 private(area: Int, group: Int, serial: Int)
object SSN3:
    def fromString(string: String): Either[List[Throwable], SSN3] = string match
        case null =>
            LeftThrowable("Social security is null")
        case _ =>
            val split = string.split("-")
            if (split.size != 3)
                LeftThrowable(s"Three different sets of digits expected but ${split.size} found")
            else if (split(0).filter(!_._isDigit).isEmpty)
                LeftThrowable(s"No digits found in area position '${string}'")
            else if (split(1).filter(!_._isDigit).isEmpty)
                LeftThrowable(s"No digits found in group position '${string}'")

```

```

    else if (split(2).filter(_._isDigit).isEmpty)
      LeftThrowable(s"No digits found in serial position '${string}'")
    else if (split(0).filter(!_._isDigit).nonEmpty)
      LeftThrowable(s"Invalid digit found in area position '${string}'")
    else if (split(1).filter(!_._isDigit).nonEmpty)
      LeftThrowable(s"Invalid digit found in group position '${string}'")
    else if (split(2).filter(!_._isDigit).nonEmpty)
      LeftThrowable(s"Invalid digit found in serial position '${string}'")
    else
      Right(SSN3(area = split(0).toInt, group = split(1).toInt, serial = split(2).toInt))

final case class Email3 private(user: String, domain: String)
object Email3:
  def fromString(string: String): Either[List[Throwable], Email3] = string match
    case null =>
      LeftThrowable("Email is null")
    case _ =>
      val split = string.split("@")
      if (split.size != 2)
        LeftThrowable(s"Email '${string}' is malformed")
      else
        Right(Email3(user = split(0), domain = split(1)))

```

## Introducing mapN (AKA Applicatives)

Applicatives are helpful when all fields need to be evaluated. Unfortunately, this capability is not included in the scala standard library. [Libraries like cats provide it](#). I am using a [homegrown version of applicatives you can use in this link](#) in case you don't want to deal with the somehow heavy cats library.

Here is how it would look when putting together these validated fields into a case class. Similar to the example above:

```

final case class Employee3 private(ssn: SSN3, email: Email3)

val employeeGood: EitherError[Employee3] = Applicative.mapN(
  Email3.fromString("hernan@gmail.com"),
  SSN3.fromString("111-11-1111")
)((email, ssn) => Employee3(email = email, ssn = ssn))
println(employeeGood)

```

```
// Right(Employee( SSN2(111,11,1111), Email2(hernan, gmail.com)))

val employeeBadEmail: EitherError[Employee3] = Applicative.mapN(
  Email3.fromString("hernan#gmail.com"),
  SSN3.fromString("111-11-1111")
)((email, ssn) => Employee3(email = email, ssn = ssn))
println(employeeBadEmail)
// Left(List(java.lang.Throwable: Email 'hernan#gmail.com' is malformed))

val employeeBadEmailAndSsn: EitherError[Employee3] = Applicative.mapN(
  Email3.fromString("hernan#gmail.com"),
  SSN3.fromString("111111111")
)((email, ssn) => Employee3(email = email, ssn = ssn))
println(employeeBadEmailAndSsn)
// Left(List(java.lang.Throwable: Email 'hernan#gmail.com' is malformed, java.lang.Throwable:
// Three different sets of digits expected but 1 found))
```

So far, we've learned two useful techniques for evaluating and validating fields, for comprehensions for the simple stop-on-first-fail and mapN when catching multiple errors is beneficial.

---

Revision #21

Created Thu, Aug 10, 2023 1:27 AM by [hernan saab](#)

Updated Sun, Oct 22, 2023 4:55 PM by [hernan saab](#)