Improving twitter project

Working with error failish functions

We want to get a user from DB:

```
getById(Id): User
```

But it can throw error (ConnectionError, UserNotFound) we cannot understand it through signature, use annotation:

```
@throws[ConnectionError]
@throws[UserNotFound]
getById(Id): User
```

Working with error failish functions

```
try {
   val user = getById(1)
} catch {
   case ce: ConnectionError => ...
   case uf: UserNotFound => ...
}
```

What's wrong?

```
@throws[ConnectionError, UserInvalid] // not real code
updateSomething(User): ()
try {
    val user = getById(1)
    updateSomething(user)
} catch {
    Handle getById errors
    Handle updateSomething erros
}
```

Even worth

```
try {
    m1; m2; m3 ...
} catch {
    Handle m1 errors; handle m2 errors; handle m3 ...
}
```

Even even worth!

```
def someMethod1 // handles n errors

def someMethod2 // handles n1 errors

def someMethod3 // handles n2 errors

...
```

Can Option or Either help us?

getById(Id): User

To **getById(Id): Option[User]**

// Return None or Some(u: User) (no info about err):(

getById(Id): Either[AppError, User] // now better it will work as ‘try-catch’, but will use ‘match-case’

Returns **Left(err: AppError)** or **Right(u: User)**

Application Errors

```
abstract class AppError

case class DbConnection(err: String) extends AppError

case object UserNotFound(id: Id) extends AppError

...

showErr(err: AppError): String = err match {
  case DbConnection(err) => err
  case UserNotFound => “Cannot find user with ID: <id>”
  …
}
```

Handling Either

```
updateSomething(User): Either[AppError,()]

getById(1) match {
  case Right(u) => updateSomething(u) match {
    case Left(err) => showErr(err)
    case Right(_) => “ok”
  }
  case Left(err) => showErr(err)
```

Seems like try-catch but with different syntax

What is common?

If something goes wrong, stop it and return error! Than we handle it

What is common?

```
@throws[AppError]
getById(Id): User // fails or returns result
getById(Id): Option[User] // fails or returns result
getById(Id): Either[AppError,User] // fails or returns
result

f[E,A,B](arg: A): E B // more generally
// where E could be Option or Either[AppError,_]
```

```
f[E,A,B](arg: A): E B
```

E.flatmap(f: A => E B): E B // looks similar

E could be **List**, **Option**, **Either**

And we know that **flatMap** used in **for-comprehensions**. But what does it mean???

```
getById(Id): Either[AppError, User]
updateSomething(User): Either[AppError, ()]
```

Let’s evaluate: **getById(1).flatMap(updateSomething)**

```
  getById(1).flatMap(updateSomething)

val u: Either[AppError, User] = getById(1)
val res: Either[AppError, ()]= u.flatMap(updateSomething)


u.flatMap(updateSomething) ~
u match {
  case Right(user) => updateSomething(user) // same here
  case err => return err
}
```

```
class Either[Error,Value]

flatMap[B](f: Value => Either[Error,B]): Either[Error,B]{
  this match {
    case Right(v: Value) => f(v) // Either[Error,B]
    case Left(err) => Left(err) // type: Either[Error,B]
  }
}
map[B](f: Value => B):Either[Error,B] // same as flatMap
// but case Right(v: Value) => Right(f v)
```

For-comprehension recall

```
respond[A](res: Either[AppError,A]): HTTPResponse[A]
getById(1).flatMap(updateSomething).map(respond) ~
for {
  u <- getById(1)
  res <- updateSomething(u)
} yield (respond(res))
// respond converts AppError to a HTTPResponse with error
code and body with err
// and A to something with code 200 and appropriate JSON
```

For-comprehension recall

```
respond[A](res: Either[AppError,A]): HTTPResponse[A]
getById(1).flatMap(updateSomething).map(respond) ~
for {
  u <- getById(1)
  res <- updateSomething(u)
} yield (respond(res))
// respond converts AppError to a HTTPResponse with error
code and body with err
// and A to something with code 200 and appropriate JSON
```

What about validations?

```
for {
  u <- getById(1)
  if (validation(u)) // we want to stop here if false
  res <- updateSomething(u)
} yield (respond(res))


// We can implement
validate(e: AppError)(b: Boolean): Either[AppError,()] =
  if b Left(e) else Right(Ok)
```

Something related to twitter

```
type Action[A] = Either[AppError, A]
def unauthorized = validate(Unauthorized)
class User {
  …
  def isAuthor(t: Twit) = unauthorized(implementHere)
}
getParam[A]: String => Action[A] // gets param from url
getUserById: Id => Action[User]
auth: Action[User]
```

Something related to twitter

```
delete("\twits\:id") = for {
  user <- auth // Can return some errors
  twitId <- getParam("id") // Can return some errors
  twit <- getTwitById(twitId) // Can return some errors
  _ <- user.isAuthor(twit) // Can return Unauthorized
  res <- twit.delete // Can return some errors
} yield res
```

What about pure functions??

How to embed pure functions??

```
f: … => SomeValue // could not produce error
```

Pure it!

```
def pure[A](el: A): Action[A] = Right(el)

for {
  user <- auth // auth: Action[User]
  v <- pure(1+1) // 1+1: Int, but with pure: Action[Int]
  …
} yield ...
```

You are now Monads Pre-masters!

```
trait Monad[M, A, B] {
  def pure(A): M A              // return in Haskell
  def flatMap(A => M B): M B // v >>= f in Haskell
}

def >=>[M,A,B,C](f: A => M[B], g: B => M[C]): A => M[C] {
  a: A => pure(a).flatMap(f).flatMap(g)
}
f >=> g // f = getById, g = updateSomething
```

Modularity

For most of the projects. There was a problem with modularity :(

Class Server extends … {

…

400 lines of code

}

I wanted to split everything and it was difficult for me.

Traits saved me

```
trait ServerState {
  protected var users: List[User] = List()
  protected var tweets: List[Tweet] = List()
}
```

We can implement a bunch of methods for access data here.

In production there should be something like config.

Basic servlet - nothing special here

```
trait Servlet extends ScalatraServlet with
JacksonJsonSupport with ServerState {
  before() {
    contentType = formats("json")
  }

  protected implicit lazy val jsonFormats: Formats =
DefaultFormats
}
```

Trait hor Handlers

```
trait Handler extends Servlet {
  … utils for handlers implementation
   type Action[A] = Either[ServerException, A]
   def getParam(str: String): Action[String] = {
      Right(params.getOrElse(str,
         return Left(ParamError(str))))
   }
...
}
```

Handlers!

```
trait Authentication extends Handler {
  def auth(): Action[User] = … // implementation here
  def genToken(nickname: String): (String, String) = ...
}

trait UsersHandlers extends Handler with Authentication {
  def usersPost = for { …
  def usersGet = for { …
}
```

Now Server could be more convinient

```
object Server extends UsersHandlers with TwitsHandler {
  post("/users") { makeHandler(usersPost) }
  get("/users")  { makeHandler(usersGet) }
  … other routes
}
```