The Road To
MONAD TRANSFORMERS

by 黎勇禪 Pawel Lisewski
pawel.lisewski@gmail.com

WHY TALK ABOUT MONAD TRANSFORMERS?

But why listen?

Do what?

PART I: THE BASE >>=

PART II: THE DISCOVERY >>=

PART III: THE JOURNEY

The true voyage of discovery is not in seeking new landscapes but in having new eyes.

- Marcel Proust

PART I
THE BASE 'MONAD'

A FEW THOUGHTS ABOUT MONADS:

- very common in modern FP code
- many implementations, e.g. Maybe, Reader, Writer, STM...
- specialized to solve very specific kinds of problems by providing an environment and services
- ...

STATE MONAD: ENVIRONMENT AND SERVICES

(Store the value for later reference, query the value)

```haskell
class Monad m => MonadState s m | m -> s where
    -- | Return the state from the internals of the monad.
    get :: m s

    -- | Replace the state inside the monad.
    put :: s -> m ()

    -- | Embed a simple state action into the monad.
    state :: (s -> (a, s)) -> m a

-- | Maps an old state to a new state inside a state monad.
-- The old state is thrown away.
modify :: MonadState s m => (s -> s) -> m ()
```

THE POWER OF MONAD:

- is about keeping state pure?
- is about side effects?
- is like a burrito?
- makes composition possible?

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)

func1, func2, func3 :: String -> Maybe String
func1 >=> func2 >=> func3

doState :: State Int Int
doState = put 1 >> modify (+1) >> get
```

THE POWER OF MONAD:

- makes composition possible

Monad takes care of threading the stored value through the whole computation

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)

func1, func2, func3 :: String -> Maybe String
func1 >=> func2 >=> func3

doState :: State Int Int
doState = put 1 >> modify (+1) >> get
```

IN OTHER WORDS, FORMALLY:

Monads promote modularity and code reuse by encapsulating often-used computational strategies into single blocks of code that can be used to construct many different computations.

MEANWHILE IN THE REAL WORLD... IT'S LAYERS AND LAYERS:

- talking to DBs
- using upstream APIs
- own business logic with configs and exceptions
- ...

QUESTION:

What happens when we need to use different 'monadic services' at the same time?

PART II

THE DISCOVERY

Example: Login Shell

```
main :: IO ()
main = do
  maybeUserName <- readUserName
  case maybeUserName of
    Nothing -> print "Invalid user name!"
    Just uName -> do
      maybeEmail <- readEmail
      case maybeEmail of
        Nothing -> print "Invalid email!"
        Just email -> do
          maybePassword <- readPassword
          case maybePassword of
            Nothing -> print "Invalid password"
            Just password -> login uName email password

readUserName :: IO (Maybe String)
readUserName = do
  str <- getLine
  if length str > 5
    then return $ Just str
```

The Impossible Join

```
join :: (Monad m) => m (m a) -> m a

-- this is impossible
join :: (Monad f, Monad g) => f (g (f (g a))) -> f (g a)
```

ANATOMY OF A TRANSFORMER

A transformer is a way to add the capability of the PRECURSOR monad to the BASE monad.

```
newtype MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }

instance (Monad m) => Monad (MaybeT m) where
    return = MaybeT . return . Just
    x >>= f = MaybeT $ do
        v <- runMaybeT x
        case v of
            Nothing -> return Nothing
            Just y  -> runMaybeT (f y)
    fail _ = MaybeT (return Nothing)
```

Example: Login Shell With A Transformer

```haskell
main :: IO ()
main = do
  maybeCreds <- runMaybeT $ do
    usr <- readUserName
    email <- readEmail
    pass <- readPassword
    return (usr, email, pass)
  case maybeCreds of
    Nothing -> print "Login denied!"
    Just (u, e, p) -> login u e p

readUserName, readEmail, readPassword :: MaybeT IO String

readUserName = MaybeT $ do
  str <- getLine
  if length str > 5
    then return $ Just str
    else return Nothing
```

EXAMPLE MONAD TRANSFORMERS

```haskell
data Maybe a = Just a | Nothing
newtype MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }
```

```haskell
data Either e a = Left e | Right a
newtype EitherT e m a = EitherT { runEitherT :: m (Either e a) }
```

```haskell
newtype Writer w a = Writer { runWriter :: (a, w) }
newtype WriterT w m a = WriterT { runWriterT :: m (a, w) }
```

```haskell
newtype Reader r a = Reader { runReader :: r -> a }
newtype ReaderT r m a = ReaderT { runReaderT :: r -> m a }
```

```haskell
newtype State s a = State {runState :: s -> (a, s)}
newtype StateT s m a = StateT { runStateT :: s -> m (a, s) }
```

QUESTION:

Does ordering matter? Do transformers commute?

PART III

THE JOURNEY

The familiar becomes new, the new becomes familiar.

WHY A SERVER-SIDE FRAMEWORK EXAMPLE?

- a familiar context to try something new
- good example of service layers and composition
- great follow-up resources

Code repo for this part on Github

FIRST ITERATION

An Introduction to the app

1st Iteration: Types

```
type Request = String
type Response = String

type Application = Request -> Response
type Middleware = Application -> Application

newtype AppState = AppState { routes :: [Middleware] }
type AppStateT = State AppState
```

1st Iteration: Route Actions

```
-- Route Handlers ----------------------------------------------
---
routeAction1 :: Request -> Response
routeAction1 request = textResponse request "Hello from Route 1"

routeAction2 :: Request -> Response
routeAction2 request = textResponse request "Hello from Route 2"

notFound :: Request -> Response
notFound request = textResponse request "Hello from the DEFAULT
route"

textResponse :: String -> String -> String
textResponse req msg = unwords ["Request:", req, "\nResponse:",
msg]
----------------------------------------------------------------
---
```

1st Iteration: App State

```
-- App State ------------------------------------------------------
---
myApp :: AppStateT ()
myApp = do
  addRoute "one" routeAction1
  addRoute "two" routeAction2

myServer :: AppStateT () -> IO ()
myServer myApp = do
  let appState = execState myApp AppState{routes=[]}
  userInputLoop appState

main :: IO ()
main = myServer myApp
------------------------------------------------------------------
---
```

1st Iteration: Adding Routes

```haskell
-- Adding Routes ---------------------------------------------------
---
addRoute :: String -> (Request -> Response) -> AppStateT ()
addRoute pat action = modify $ \s -> addRoute' (route pat action) s

addRoute' :: Middleware -> AppState -> AppState
addRoute' m s@AppState {routes = ms} = s {routes = m:ms}

route :: String -> (Request -> Response) -> Middleware
route pat action nextApp req =
  let tryNext = nextApp req in
  if pat == req
  then
    action req
  else
    tryNext
-----------------------------------------------------------------------
---
```

1st Iteration: Running The App

```
-- Running the App ---------------------------------------------
---
runMyApp :: (Request -> Response) -> AppState -> Request ->
Response
runMyApp defHandler appState =
  foldl (flip ($)) defHandler (routes appState)

userInputLoop :: AppState -> IO ()
userInputLoop appState = do
  putStrLn "Awaiting requests..."
  request <- getLine

  unless (request == "q") $ do
    let response = runMyApp notFound appState request
    putStrLn response
    userInputLoop appState
-----------------------------------------------------------------
---
```

SECOND ITERATION

有拜有保庇 (If Things Go Wrong)

2nd Iteration: Types

```
type Request = String
type Response = String

type Application = Request -> Response
type Middleware = Application -> Application

type ActionT = Either ActionError Response
type ActionError = String

newtype AppState = AppState { routes :: [Middleware] }
type AppStateT = State AppState
```

2nd Iteration: Route Actions

```haskell
-- Route Handlers -------------------------------
routeAction1 :: Request -> ActionT
routeAction1 request = return $
  textResponse request "Hello from Route 1"

routeAction2 :: Request -> ActionT
routeAction2 request = throwError "Error in Route 2"

notFound :: Request -> Response
notFound request = textResponse request "Hello from the DEFAULT
route"

textResponse :: String -> String -> String
textResponse req msg = unwords ["Request:", req, "\nResponse:",
msg]
-----------------------------------------------
```

Meet MonadError

```haskell
class (Monad m) => MonadError e m | m -> e where
    -- | Is used within a monadic computation to begin exception
processing.
    throwError :: e -> m a

    -- do { action1; action2; action3 } `catchError` handler
    catchError :: m a -> (e -> m a) -> m a

instance MonadError e (Either e) where
    throwError             = Left
    Left  l `catchError` h = h l
    Right r `catchError` _ = Right r
```

2nd Iteration: App State

```
-- App State ---------------------------------------
myApp :: AppStateT ()
myApp = do
  addRoute "one" routeAction1
  addRoute "two" routeAction2

myServer :: AppStateT () -> IO ()
myServer myApp = do
  let appState = execState myApp AppState{routes=[]}
  userInputLoop appState

main :: IO ()
main = myServer myApp
-----------------------------------------------------
```

2nd Iteration: Adding Routes

```
-- Adding Routes ---------------------------------
addRoute :: String -> (Request -> ActionT) -> AppStateT ()
addRoute pat action = modify $ \s -> addRoute' (route pat action) s

addRoute' :: Middleware -> AppState -> AppState
addRoute' m s@AppState {routes = ms} = s {routes = m:ms}

route :: String -> (Request -> ActionT) -> Middleware
route pat action nextApp req =
  let tryNext = nextApp req in
  if pat == req
  then
    either ("Error: " ++ ) id (action req)
  else
    tryNext
-------------------------------------------------
```

2nd Iteration: Running The App

```haskell
-- Running the App -------------------------------
runMyApp :: Application -> AppState -> Application
runMyApp defHandler appState =
  foldl (flip ($)) defHandler (routes appState)

userInputLoop :: AppState -> IO ()
userInputLoop appState = do
  putStrLn "Awaiting requests..."
  request <- getLine

  unless (request == "q") $ do
    let response = runMyApp notFound appState request
    putStrLn response
    userInputLoop appState
-------------------------------------------------
```

THIRD ITERATION
Reading The Environment

3rd Iteration: Types

```
type Request = String
type Response = String

type Application = Request -> Response
type Middleware = Application -> Application

type ActionT = ExceptT ActionError (Reader Request) Response
type ActionError = String

newtype AppState = AppState { routes :: [Middleware] }
type AppStateT = State AppState
```

3rd Iteration: Route Actions

```
-- Route Handlers ----------------------------------------------
---
routeAction1 :: ActionT
routeAction1 = do
  request <- lift ask
  return $ textResponse request "Hello from Route 1"

routeAction2 :: ActionT
routeAction2 = throwError "Error in Route 2"

notFound :: Application
notFound request = "Hello from the DEFAULT route"

textResponse :: String -> String -> String
textResponse req msg = unwords ["Request:", req, "\nResponse:",
msg]
-----------------------------------------------------------------
---
```

Meet MonadTrans

```haskell
class MonadTrans t where
  -- | Lift a computation from the argument monad
  --   to the constructed monad.
  lift :: (Monad m) => m a -> t m a

instance MonadTrans MaybeT where
  lift = MaybeT . liftM Just

lift                  :: Monad m => m a -> t m a
(MaybeT . liftM Just) :: Monad m => m a -> MaybeT m a

MaybeT                :: m (Maybe a) -> MaybeT m a
(liftM Just)          :: Monad m => m a -> m (Maybe a)
```

3rd Iteration: Adding Routes

```haskell
-- Adding Routes ---------------------------------------------------
---
addRoute :: String -> ActionT -> AppStateT ()
addRoute pat action = modify $ \s -> addRoute' (route pat action) s

addRoute' :: Middleware -> AppState -> AppState
addRoute' m s@AppState {routes = ms} = s {routes = m:ms}

route :: String -> ActionT -> Middleware
route pat action nextApp req =
  let tryNext = nextApp req in
    if pat == req
      then
        runAction action req
      else
        tryNext
-----------------------------------------------------------------------
---
```

3rd Iteration: Running Actions

```
-- Running Actions -------------------------------------------------
---
runAction :: ActionT -> Request -> Response
runAction action request = either (const "Error") id
                         $ flip runReader request
                         $ runExceptT
                         $ action `catchError` errorHandler

errorHandler :: ActionError -> ActionT
errorHandler err = return $ "Oops: " ++ err
--------------------------------------------------------------------
---
```

FOURTH ITERATION
The State of Response

4th Iteration: Types

```
type Request = String
type Response = String

type Application = Request -> Response
type Middleware = Application -> Application

type ActionT a = ExceptT ActionError (ReaderT Request (State
Response)) a
type ActionError = String

newtype AppState = AppState { routes :: [Middleware] }
type AppStateT = State AppState
```

4th Iteration: Route Actions

```
-- Route Handlers ----------------------------------------------
---
routeAction1 :: ActionT ()
routeAction1 = do
  request <- lift ask
  lift . lift $ modify (const $ textResponse request "Hello from
Route 1")

routeAction2 :: ActionT ()
routeAction2 = throwError "Error in Route 2"

notFound :: Application
notFound request = "Hello from the DEFAULT route"

textResponse :: String -> String -> String
textResponse req msg = unwords ["Request:", req, "\nResponse:",
msg]
----------------------------------------------------------------
---
```

4th Iteration: Adding Routes

```haskell
-- Adding Routes ---------------------------------------------
---
addRoute :: String -> ActionT () -> AppStateT ()
addRoute pat action = modify $ \s -> addRoute' (route pat action) s

addRoute' :: Middleware -> AppState -> AppState
addRoute' m s@AppState {routes = ms} = s {routes = m:ms}

route :: String -> ActionT () -> Middleware
route pat action nextApp req =
  let tryNext = nextApp req in
    if pat == req
      then
        runAction action req
      else
        tryNext
-------------------------------------------------------------------
---
```

4th Iteration: Running Actions

```haskell
-- Running Actions -------------------------------------------------
runAction :: ActionT () -> Request -> Response
runAction action request = flip execState ""
                         $ flip runReaderT request
                         $ runExceptT
                         $ action `catchError` errorHandler

errorHandler :: ActionError -> ActionT ()
errorHandler err = lift . lift $ modify (const $ "Oops: " ++ err)
--------------------------------------------------------------------
```

FIFTH ITERATION
GET /launch-the-missiles

5th Iteration: Types

```
type Request = String
type Response = String

type Application = Request -> IO Response
type Middleware = Application -> Application

type ActionT a = ExceptT ActionError (ReaderT Request (StateT
Response IO)) a
type ActionError = String

newtype AppState = AppState { routes :: [Middleware] }
type AppStateT = State AppState
```

Meet MonadIO

```haskell
class (Monad m) => MonadIO m where
  -- | Lift a computation from the 'IO' monad.
  liftIO :: IO a ->  m a

instance (MonadIO m) => MonadIO (MaybeT m) where
    liftIO = lift . liftIO
```

5th Iteration: Route Actions

```
-- Route Handlers ----------------------------------------------
---
routeAction1 :: ActionT ()
routeAction1 = do
  request <- lift ask
  liftIO $ putStrLn "We're doing IO"
  lift . lift $ modify (const $ textResponse request "Hello from
Route 1")

routeAction2 :: ActionT ()
routeAction2 = throwError "Error in Route 2"

notFound :: Application
notFound request = return "Hello from the DEFAULT route"

textResponse :: String -> String -> String
textResponse req msg = unwords ["Request:", req, "\nResponse:",
msg]
----------------------------------------------------------------
---
```

5th Iteration: Adding Routes

```haskell
-- Adding Routes ---------------------------------------------------
---
addRoute :: String -> ActionT () -> AppStateT ()
addRoute pat action = modify $ \s -> addRoute' (route pat action) s

addRoute' :: Middleware -> AppState -> AppState
addRoute' m s@AppState {routes = ms} = s {routes = m:ms}

route :: String -> ActionT () -> Middleware
route pat action nextApp req =
  let tryNext = nextApp req in
    if pat == req
      then
        runAction action req
      else
        tryNext
-----------------------------------------------------------------------
---
```

5th Iteration: Running Actions

```
-- Running Actions -----------------------------------------------
---
runAction :: ActionT () -> Request -> IO Response
runAction action request = do
  (a,s) <- flip runStateT ""
           $ flip runReaderT request
           $ runExceptT
           $ action `catchError` errorHandler
  return $ either (const "Error") (const s) a

errorHandler :: ActionError -> ActionT ()
errorHandler err = lift . lift $ modify (const $ "Oops: " ++ err)
-------------------------------------------------------------------
---
```

5th Iteration: Running The App

```
-- Running the App ------------------------------------------------
---
runMyApp :: Application -> AppState -> Application
runMyApp defHandler appState =
  foldl (flip ($)) defHandler (routes appState)

userInputLoop :: AppState -> IO ()
userInputLoop appState = do
  putStrLn "Awaiting requests..."
  request <- getLine

  unless (request == "q") $ do
    let response = runMyApp notFound appState request
    response >>= putStrLn
    putStrLn "---"
    userInputLoop appState
------------------------------------------------------------------
---
```

SIXTH ITERATION

愛拼才會贏? Newtype Deriving

6th Iteration: Types

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

type Request = String
type Response = String

type Application = Request -> IO Response
type Middleware = Application -> Application

newtype ActionT a = ActionT { runAT :: ExceptT ActionError
                                       (RT.ReaderT Request
                                       (ST.StateT Response IO)) a }
                            deriving (Functor, Applicative, Monad,
                                      MonadIO, MonadReader
Request,MonadState Response, MonadError ActionError)
type ActionError = String

newtype AppState = AppState { routes :: [Middleware] }
type AppStateT = ST.State AppState
```

6th Iteration: Route Actions

```haskell
-- Route Handlers ----------------------------------------------
---
routeAction1 :: ActionT ()
routeAction1 = do
  request <- ask
  liftIO $ putStrLn "We're doing IO"
  modify (const $ textResponse request "Hello from Route 1")

routeAction2 :: ActionT ()
routeAction2 = throwError "Error in Route 2"

notFound :: Application
notFound request = return "Hello from the DEFAULT route"

textResponse :: String -> String -> String
textResponse req msg = unwords ["Request:", req, "\nResponse:",
msg]
----------------------------------------------------------------
---
```

6th Iteration: App State

```
-- App State ------------------------------------------------------
---
myApp :: AppStateT ()
myApp = do
  addRoute "one" routeAction1
  addRoute "two" routeAction2

myServer :: AppStateT () -> IO ()
myServer myApp = do
  let appState = ST.execState myApp AppState{routes=[]}
  userInputLoop appState

main :: IO ()
main = myServer myApp
------------------------------------------------------------------
---
```

6th Iteration: Adding Routes

```
-- Adding Routes ---------------------------------------------------
---
addRoute :: String -> ActionT () -> AppStateT ()
addRoute pat action = modify $ \s -> addRoute' (route pat action) s

addRoute' :: Middleware -> AppState -> AppState
addRoute' m s@AppState {routes = ms} = s {routes = m:ms}

route :: String -> ActionT () -> Middleware
route pat action nextApp req =
  let tryNext = nextApp req in
    if pat == req
      then
        runAction action req
      else
        tryNext
-----------------------------------------------------------------------
---
```

6th Iteration: Running Actions

```haskell
-- Running Actions -------------------------------------------------
---
runAction :: ActionT () -> Request -> IO Response
runAction action request = do
  (a,s) <- flip ST.runStateT ""
           $ flip RT.runReaderT request
           $ runExceptT
           $ runAT
           $ action `catchError` errorHandler
  return $ either (const "Error") (const s) a

errorHandler :: ActionError -> ActionT ()
errorHandler err = modify (const $ "Oops: " ++ err)
-------------------------------------------------------------------
---
```

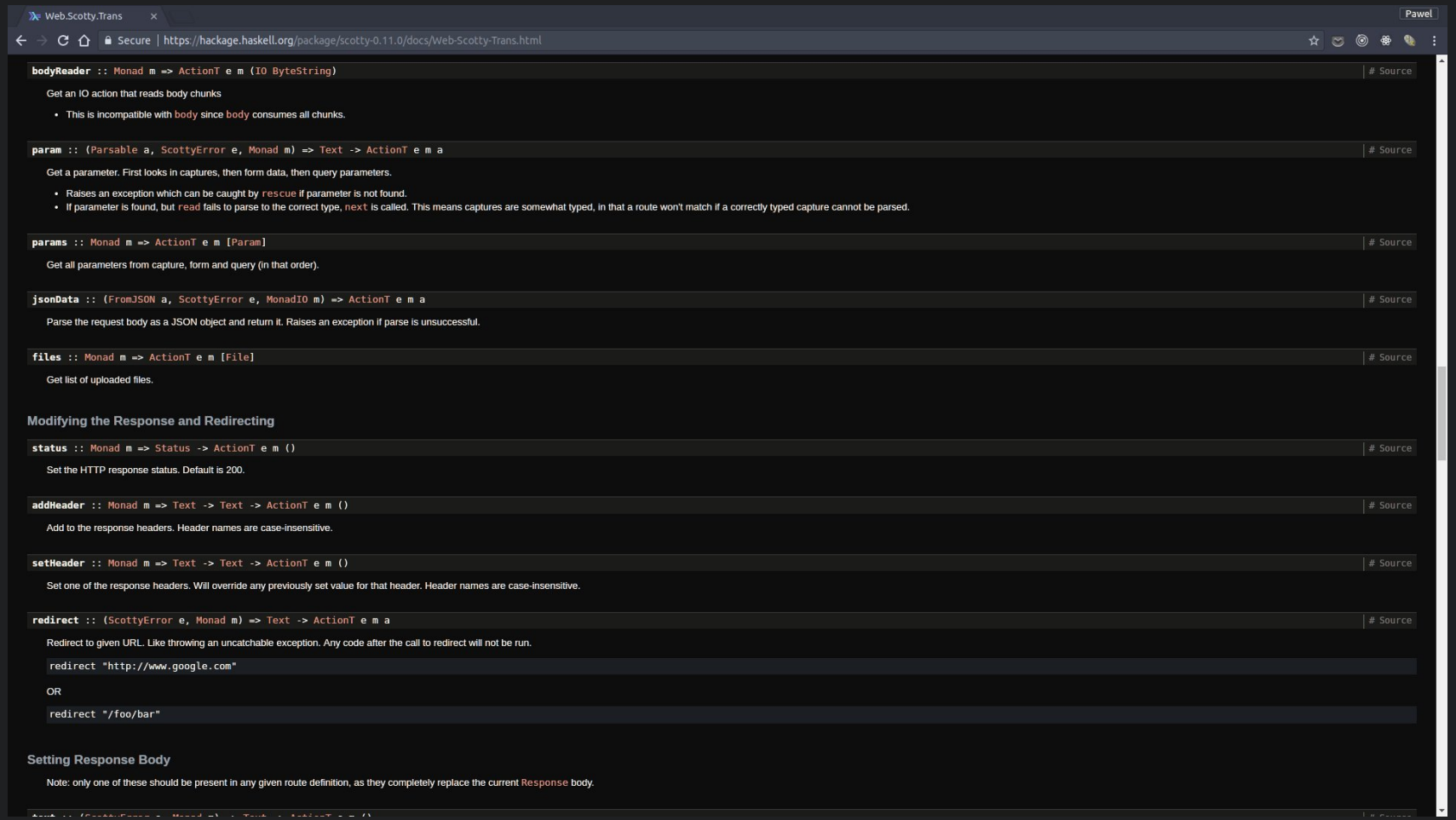
6th Iteration: Running The App

```haskell
-- Running the App ------------------------------------------------
---
runMyApp :: Application -> AppState -> Application
runMyApp defHandler appState =
  foldl (flip ($)) defHandler (routes appState)

userInputLoop :: AppState -> IO ()
userInputLoop appState = do
  putStrLn "Awaiting requests..."
  request <- getLine

  unless (request == "q") $ do
    let response = runMyApp notFound appState request
    response >>= putStrLn
    putStrLn "---"
    userInputLoop appState
-----------------------------------------------------------------
---
```

Sneak Peak into Scotty



OUTRO

Give me six hours to chop a tree and I will spend the first four sharpening the axe.

- Abraham Lincoln

WHAT NEXT?

- Write instances of transfomers
- Study Scotty on Hackage
- Check Read You A Scotty
- Look into Parsers
- Is there an IO_T transformer?

You can contact me at
pawel.lisewski@gmail.com