Databases and HTTP Requests

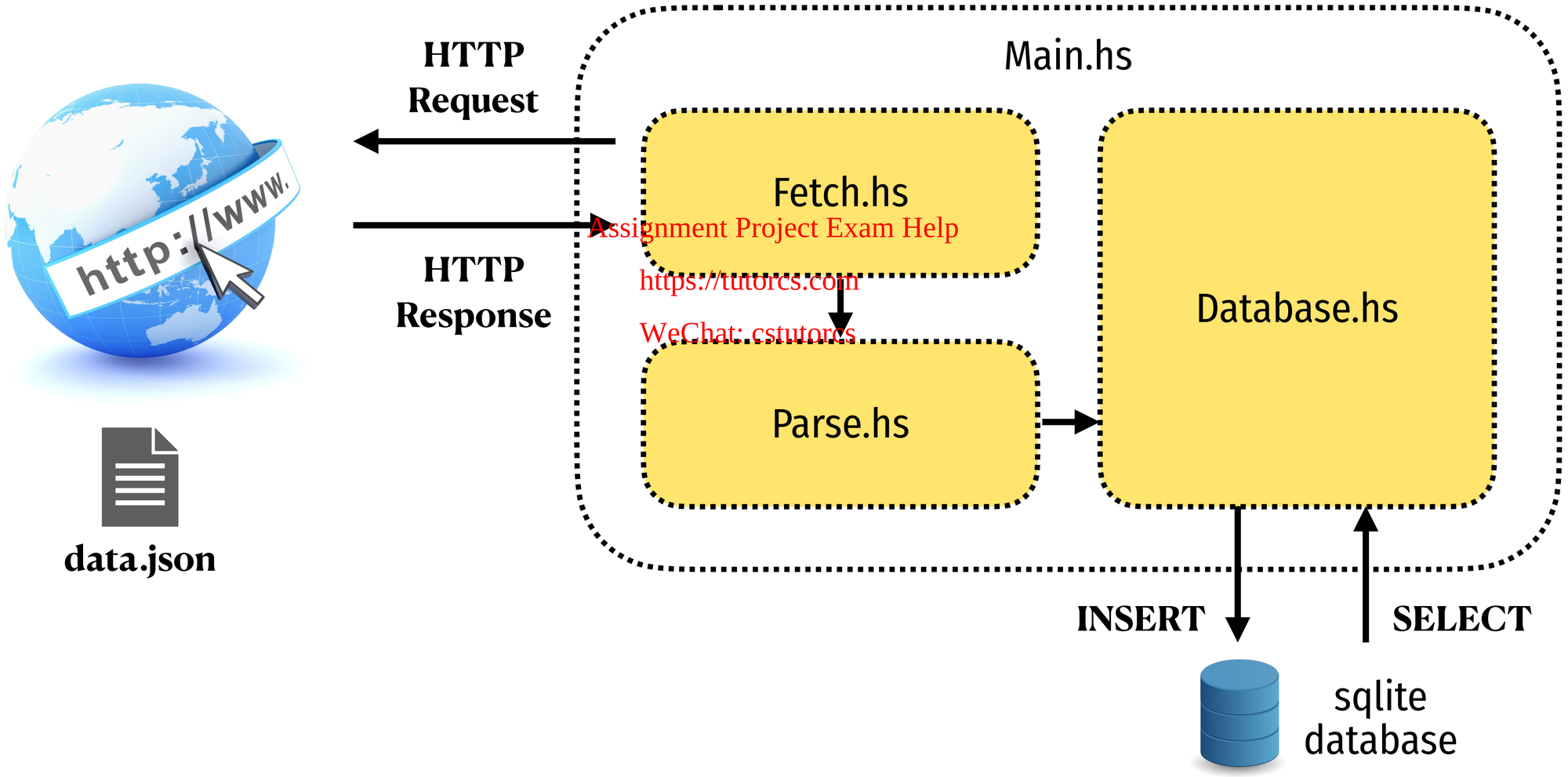
ECS713 : Functional Programming
Week 08

Prof. Edmund Robinson
Dr Paulo Oliva

Week 8: Contents

- Downloading data from the web (http-conduit)
- Parsing text into Haskell data types (aeson)
- Using a database (sqlite-simple)
- Example: Covid cases data
- Error handling in Haskell

Group Project



Example 1. Covid dataset

https://opendata.ecdc.europa.eu/covid19/casedistribution/json/


```
{
   "records" : [
      {
         "dateRep" : "14/12/2020",
         "day" : "14",
         "month" : "12",
         "year" : "2020",
         "cases" : 746,
         "deaths" : 6,
         "countriesAndTerritories" : "Afghanistan",
         "geoId" : "AF",
         "countryterritoryCode" : "AFG",
         "popData2019" : 38041757,
         "continentExp" : "Asia",
         "Cumulative_number_for_14_days_of_COVID-19_cases_per_100000" : "9.01377925"
      },
      {
         "dateRep" : "13/12/2020",
         "day" : "13",
         "month" : "12",
         "year" : "2020",
         "cases" : 298,
         "deaths" : 9,
         "countriesAndTerritories" : "Afghanistan",
         "geoId" : "AF",
         "countryterritoryCode" : "AFG",
         "popData2019" : 38041757,
         "continentExp" : "Asia",
         "Cumulative_number_for_14_days_of_COVID-19_cases_per_100000" : "7.05277624"
      },
      {
         "dateRep" : "12/12/2020",
         "day" : "12"
```

Downloading from the Web

HTTP Modules

- Several libraries available for HTTP requests
  - **Network.HTTP (HTTP-4000)**
  - **Network.HTTP.Conduit (http-conduit)**
  - **Network.HTTP.Simple (http-conduit)**
- We recommend you use the last one
- Use URLs to create HTTP requests

URL

1. Adding http-conduit to project

package.yaml

```
name:                covid-project
version:             0.1.0.0
github:              "githubuser/covid-project"
license:             BSD3
author:              "Author name here"
maintainer:          "example@example.com"
copyright:           "2020 Author name here"

extra-source-files:
- README.md
- ChangeLog.md

# Metadata used when publishing your package
# synopsis:            Short description of your package
# category:            Web

# To avoid duplicated efforts in documentation and dealing with the
# complications of embedding Haddock markup inside cabal files, it is
# common to point users to the README.md file.
description:         Please see the README on GitHub at <https://github.com/githubuser/covid-project#readme>

dependencies:
- base >= 4.7 && < 5
- http-conduit
- sqlite-simple
...
```

Downloading a URL

src/Fetch.hs

```haskell
module HTTP (
    download
) where

import qualified Data.ByteString.Lazy.Char8 as L8
import Network.HTTP.Simple

type URL = String

download :: URL -> IO L8.ByteString
download url = do
    request <- parseRequest url
    response <- httpLBS request
    return $ getResponseBody response
```


parseRequest :: MonadThrow m => String -> m Request

http-client Network.HTTP.Client Network.HTTP.Client.Internal, http-conduit Network.HTTP.Client.Conduit Network.HTTP.Conduit Network.HTTP.Simple, stack Network.HTTP.StackClient

Convert a URL into a Request.

httpLBS :: MonadIO m => Request -> m (Response ByteString)

http-conduit Network.HTTP.Simple

Perform an HTTP request and return the body as a lazy `ByteString`. Note that the entire value will be read into memory at once (no lazy I/O will be performed). The advantage of a lazy `ByteString` here (versus using httpBS) is--if needed--a better in-memory representation.

Parsing

Parsing Modules

- Several libraries available for parsing text into Haskell data
  - **Data.Aeson (parse json text)**
  - **Text.CSV (parse csv text)**
  - **Text.HTML (parse html text)**
- Use the parser most appropriate for the data you have chosen
- Most parsers are defined as type classes

1. Adding aeson to project

package.yaml

```
name:                covid-project
version:             0.1.0.0
github:              "githubuser/covid-project"
license:             BSD3
author:              "Author name here"
maintainer:          "example@example.com"
copyright:           "2020 Author name here"

extra-source-files:
- README.md
- ChangeLog.md

# Metadata used when publishing your package
# synopsis:            Short description of your package
# category:            Web

# To avoid duplicated efforts in documentation and dealing with the
# complications of embedding Haddock markup inside cabal files, it is
# common to point users to the README.md file.
description:         Please see the README on GitHub at <https://github.com/githubuser/covid-project#readme>

dependencies:
- base >= 4.7 && < 5
...
- bytestring
- aeson
```

2. Define Haskell Types

/src/Types.hs

```haskell
{-# LANGUAGE DeriveGeneric #-}

module Types (
    Entry (..),
    Country (..),
    Record (..),
    Records (..)
) where

...

data Record = Record {
    date :: String,
    day :: String,
    month :: String,
    year :: String,
    cases :: Int,
    deaths :: Int,
    country :: String,
    continent ::  String,
    population :: Maybe Int
} deriving (Show, Generic)

data Records = Records {
    records :: [Record]
} deriving (Show, Generic)
```


Define Haskell types that correspond to JSON data

Parsing JSON data

src/Parse.hs

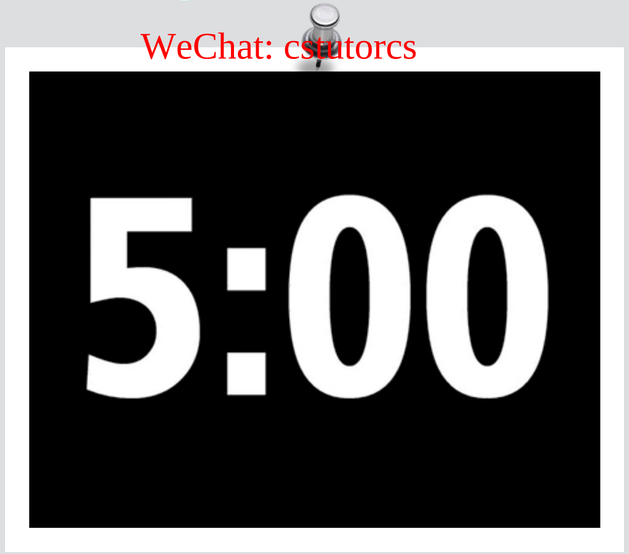
```haskell
module Parse (
    parseRecords,
) where

import Types
import Data.Aeson
import qualified Data.ByteString.Lazy.Char8 as L8

renameFields "date" = "dateRep"
renameFields "continent" = "continentExp"
renameFields "country" = "countriesAndTerritories"
renameFields "population" = "popData2019"
renameFields other = other

customOptions = defaultOptions {
    fieldLabelModifier = renameFields
}

instance FromJSON Record where
    parseJSON = genericParseJSON customOptions

instance FromJSON Records

parseRecords :: L8.ByteString -> Either String Records
parseRecords json = eitherDecode json :: Either String Records
```


eitherDecode :: FromJSON a => ByteString -> Either String a

aeson Data.Aeson, amazonka-core Network.AWS.Data.JSON Network.AWS.Prelude, aeson-compat Data.Aeson.Compat, pantry Pantry.Internal.AesonExtended, yesod-auth-oauth2 Yesod.Auth.OAuth2.Prelude, json-stream Data.JsonStream.Parser

Like decode but returns an error message when decoding fails.

Short Break



Using a Database

Database Modules

- Several libraries available for Sqlite DB management
  - **Database.HDBC** and **Database.HDBC.Sqlite3**
  - **Database.Sqlite3 (direct-sqlite)**
  - **Database.Sqlite.Simple (sqlite-simple)**
- We recommend you use the last one: it is simple, powerful and well-documented
- Use SQL queries to access the database (review if not familiar)

sqlite-simple

1. Add sqlite-simple to your project
2. Open a database connection
3. Create DB tables
4. Converting between DB and Haskell types
5. Perform SQL queries using the DB connection

1. Adding sqlite-simple to project

package.yaml

```
name:                covid-project
version:             0.1.0.0
github:              "githubuser/covid-project"
license:             BSD3
author:              "Author name here"
maintainer:          "example@example.com"
copyright:           "2020 Author name here"

extra-source-files:
- README.md
- ChangeLog.md

# Metadata used when publishing your package
# synopsis:            Short description of your package
# category:            Web

# To avoid duplicated efforts in documentation and dealing with the
# complications of embedding Haddock markup inside cabal files, it is
# common to point users to the README.md file.
description:         Please see the README on GitHub at <https://github.com/githubuser/covid-project#readme>

dependencies:
- base >= 4.7 && < 5
- http-conduit
- sqlite-simple
```

2. Open database connection

/src/Database.hs

```haskell
module Database (
    initialiseDB,
) where

import Types
import Control.Applicative
import Database.SQLite.Simple
import Database.SQLite.Simple.Internal

-- See more Database.SQLite.Simple examples at
-- https://hackage.haskell.org/package/sqlite-simple-0.4.18.0/docs/Database-SQLite-Simple.html

initialiseDB :: IO Connection
initialiseDB = open "covid.sqlite"
```



```
open :: String -> IO Connection
```

sqlite-simple Database.SQLite.Simple

Open a database connection to a given file. Will throw an exception if it cannot connect.

3. Create DB Tables

/src/Database.hs

```
createTables :: Connection -> IO ()
createTables conn = do
        execute_ conn "CREATE TABLE IF NOT EXISTS countries (\
            \id INTEGER PRIMARY KEY AUTOINCREMENT,\
            \country VARCHAR(80) NOT NULL, \
            \continent VARCHAR(50) NOT NULL, \
            \population INT DEFAULT NULL\
            \)"
        execute_ conn "CREATE TABLE IF NOT EXISTS entries (\
            \date VARCHAR(40) NOT NULL, \
            \day VARCHAR(40) NOT NULL, \
            \month VARCHAR(40) NOT NULL, \
            \year VARCHAR(40) NOT NULL, \
            \cases INT DEFAULT NULL, \
            \deaths INT DEFAULT NULL, \
            \fk_country INTEGER\
            \)"
```


execute_ :: Connection -> Query -> IO ()

sqlite-simple Database.SQLite.Simple

A version of execute that does not perform query substitution.

execute :: ToRow q => Connection -> Query -> q -> IO ()

sqlite-simple Database.SQLite.Simple

Execute an `INSERT`, `UPDATE`, or other SQL query that is not expected to return results.

Throws FormatError if the query could not be formatted correctly.

Data Types (Modifiers)

DATA TYPES	MODIFIER
INTEGER	PRIMARY KEY
VARCHAR(80)	AUTOINCREMENT
TEXT	NOT NULL
DATE	...
...	


```
CREATE TABLE person
  ( id INTEGER PRIMARY KEY AUTOINCREMENT
  , name VARCHAR(40) NOT NULL
  , birth DATE NOT NULL
  , children INTEGER
  , address TEXT
  )
```

4. Convert Haskell / DB Types

/src/Types.hs

```haskell
module Types (
    Entry (..),
    Country (..),
    Record (..),
    Records (..)
) where

import GHC.Generics

data Entry = Entry {
    date_ :: String,
    day_ :: String,
    month_ :: String,
    year_ :: String,
    cases_ :: Int,
    deaths_ :: Int,
    fk_country :: Int
} deriving (Show)
```


Define Haskell types that correspond to DB tables

4. Convert Haskell / DB Types

/src/Database.hs

```haskell
import Control.Applicative
import Database.SQLite.Simple.FromRow
import Database.SQLite.Simple.ToRow

instance FromRow Record where
    fromRow = Record <$> field <*> field <*> field <*> field <*> field <*> field <*> field <*> field

instance FromRow Country where
    fromRow = Country <$> field <*> field <*> field <*> field

instance ToRow Country where
    toRow (Country id_ country_ continent_ population_)
        = toRow (id_, country_, continent_, population_)

instance FromRow Entry where
    fromRow = Entry <$> field <*> field <*> field <*> field <*> field <*> field <*> field

instance ToRow Entry where
    toRow (Entry date_ day_ month_ year_ cases_ deaths_ fk_country)
        = toRow (date_, day_, month_, year_, cases_, deaths_, fk_country)
```


FromRow and ToRow are type classes!

5. Read/Write on DB

/src/Database.hs

```haskell
getOrCreateCountry :: Connection -> String -> String -> Maybe Int -> IO Country
getOrCreateCountry conn country continent population = do
    let selectQuery = "SELECT * FROM countries WHERE country=:country AND continent=:continent"
    results <- queryNamed conn selectQuery [":country" := country, ":continent" := continent]
    if length results > 0 then
        return . head $ results
    else do
        let insertQuery = "INSERT INTO countries (country, continent, population) VALUES (?, ?, ?)"
        execute conn insertQuery (country, continent, population)
        getOrCreateCountry conn country continent population
```


queryNamed :: FromRow r => Connection -> Query -> [NamedParam] -> IO [r]

sqlite-simple Database.SQLite.Simple

A version of query where the query parameters (placeholders) are named.

Error Handling

When things go wrong…

```
ghci> map tail ["bob", "john", "peter"]
["ob", "ohn", "eter"]
ghci> map tail ["bob","john","","peter"]
["ob", "ohn", "*** Exception: Prelude.tail: empty list
```


un-handled exceptions crash the program!

Measure 1: Avoid exception being raised

Measure 2: Handle exceptions appropriately

Avoiding Exceptions

Using Maybe

```
data Maybe a = Nothing | Just a
```

```
ghci> tail []
*** Exception: Prelude.tail: empty list
```



```
safeTail :: [a] -> Maybe [a]
safeTail [] = Nothing
safeTail (_:xs) = Just xs
```

```
ghci> safeTail []
Nothing
ghci> safeTail [1,2,3]
Just [2,3]
ghci> map safeTail ["bob","john","","peter"]
[Just "ob", Just "ohn", Nothing, Just "eter"]
```

Using Maybe

Data.List

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```


Network.CGI.Cookie

```
cookieExpires :: Cookie -> Maybe CalendarTime
```

Network.URI

```
uriAuthority :: URI -> Maybe URIAuth
```

Using Either

```
data Either a b = Left a | Right b
```

Convention is **Left = error** and **Right = success**


```
safeTail :: [a] -> Either String [a]
safeTail [] = Left "Empty List"
safeTail (_:xs) = Right xs
```

```
ghci> safeTail []
Left "Empty List"
ghci> safeTail [1,2,3]
Right [2,3]
```

Using Either

Network.Stream

```
type Result a = Either ConnError a
```


Control.Exception

```
try :: Exception e => IO a -> IO (Either e a)
```

System.IO.Error

```
tryIOError :: IO a -> IO (Either IOError a)
```

Custom Error Type

```
data Result = DivByZero | OneIsBoring | Result Double
              deriving (Show)
```

```
safeInv :: Double -> Result
safeInv 0 = DivByZero
safeInv 1 = OneIsBoring
safeInv n = Result (1/n)
```



```
ghci> safeInv 0
DivByZero
ghci> safeInv 1
OneIsBoring
ghci> safeInv 2
Result 0.5
```

Handling Exceptions

Exceptions in Haskell

Package **Control.Exception**

Exceptions may be **thrown** from any location in the program

Can only be **caught** in the **IO monad**

Exceptions are values

Mechanism to catch and handle exceptions are **functions**

Exception is a **type class**

Trying to Run…

Function - try

```
ghci> :module Control.Exception
ghci> :type try
try :: Exception e => IO a -> IO (Either e a)
ghci> let x = 5 'div' 0
ghci> let y = 5 'div' 1
ghci> print x
*** Exception: divide by zero
ghci> print y
5
ghci> try (print x) :: IO (Either SomeException ())
Left divide by zero
ghci> try (print y) :: IO (Either SomeException ())
5
Right ()
```


Exception type has to be named explicitly

Why can't we do "try (5 'div' 0)"?

Laziness and Exceptions

```
ghci> :module Control.Exception
ghci> let x = 5 'div' 0
ghci> try (print x) :: IO (Either SomeException ())
Left divide by zero
ghci> try (return x) :: IO (Either SomeException Integer)
Right *** Exception: divide by zero
ghci> try (evaluate x) :: IO (Either SomeException Integer)
Left divide by zero
```



```
return   :: a -> IO a
evaluate :: a -> IO a
```

return x = encapsulates x as an IO action
evaluate x = evaluates x and return result

Handling Exception

Function
- handle

Use **handle** to perform an action if a piece of code raises an exception

```
ghci> :module Control.Exception
ghci> :type handle
handle :: Exception e => (e -> IO a) -> IO a -> IO a
ghci> let f = (\e -> print "Argh!") :: (ArithException -> IO ())
ghci> let x = 5 'div' 0
ghci> let y = 5 'div' 1
ghci> handle f (print y)
5
ghci> handle f (print x)
Argh!
ghci> handle f (print (tail ""))
"*** Exception: Prelude.tail: empty list
```

Throwing Exceptions

Function
- throw

```
ghci> :module Control.Exception
ghci> :type throw
throw :: Exception e => e -> a
ghci> :type Overflow
Overflow :: ArithException
ghci> throw Overflow
*** Exception: arithmetic overflow
ghci> :type throw Overflow
throw Overflow :: a
ghci> let bigInc x = if x == 10 then throw Overflow else x+1
ghci> :type bigInc
bigInc :: (Eq a, Num a) => a -> a
ghci> bigInc 9
10
ghci> bigInc 10
*** Exception: arithmetic overflow
```

Your Own Exceptions…

```
{-# LANGUAGE DeriveDataTypeable #-}

import Control.Exception
import Data.Typeable

data MyError = NumberTooBig | NumberTooSmall deriving (Show, Typeable)

instance Exception MyError

goodNum :: Int -> Bool
goodNum n | n < 5 = throw NumberTooSmall
          | n > 10 = throw NumberTooBig
          | otherwise = True
```



```
ghci> :load week8.hs
ghci> goodNum 7
True
ghci> goodNum 4
*** Exception: NumberTooSmall
ghci> goodNum 11
*** Exception: NumberTooBig
```

Exceptions and IO

Functions
- **tryIOError**
- **catchIOError**

```
import System.IO.Error

main :: IO ()
main = do
     input <- tryIOError (readFile "input.txt")
     case input of
        Left e -> print "Oh no, IO exception raised"
        Right x -> print x
```



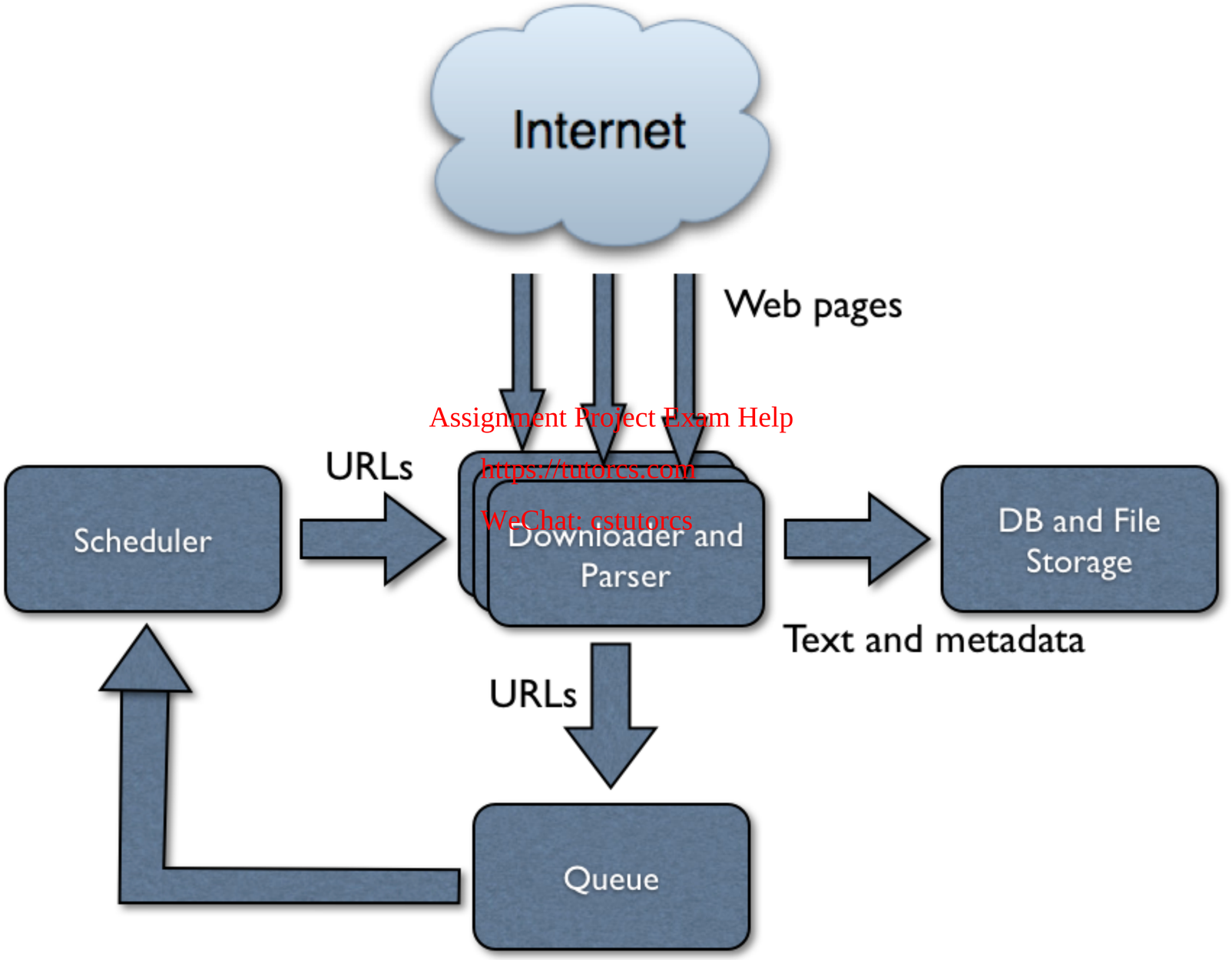
```
$ runghc week8.hs
"Oh no, IO exception raised"
```



```
import System.IO.Error

f :: IOError -> IO String
f e | ioeGetErrorType e == doesNotExistErrorType = return "File doesn't exist"
    | otherwise = return "Some other error"

main :: IO ()
main = do
    input <- catchIOError (readFile "input.txt") f
    print input
```

```
$ runghc week8.hs
"File doesn't exist"
```

Simple Web Application: Crawler


Internet
Web pages
Assignment Project Exam Help
https://tutorcs.com
WeChat: cstutorcs
URLs
Scheduler
Downloader and Parser
DB and File Storage
Text and metadata
URLs
Queue

References

- **HTTP Conduit Package**

  https://hackage.haskell.org/package/http-conduit-2.3.8/docs/Network-HTTP-Simple.html

- **Aeson Package**

  https://hackage.haskell.org/package/aeson-2.0.1.0/docs/Data-Aeson.html

- **Sqlite Simple Package**

  https://hackage.haskell.org/package/sqlite-simple-0.4.18.0/docs/Database-SQLite-Simple.html