School of Haskell (https://www.schoolofhaskell.com/) /
Starting with Haskell (https://www.schoolofhaskell.com/school/starting-with-haskell) /
Libraries and Frameworks (https://www.schoolofhaskell.com/school/starting-with-haskell/libraries-and-frameworks) /
Text manipulation (https://www.schoolofhaskell.com/school/starting-with-haskell/libraries-and-frameworks/text-manipulation) /
Parsing JSON with Aeson (https://www.schoolofhaskell.com/school/starting-with-haskell/libraries-and-frameworks/text-manipulation/json)

Parsing JSON with Aeson

1 Feb 2015 School of Haskell (https://www.schoolofhaskell.com/user/school)
View Markdown source (https://www.schoolofhaskell.com/tutorial-raw/30/0a48b1265c0af417b3881bbd9d261be7c8503ac8)

(https://twitter.com/home?status=https://www.schoolofhaskell.com/school/starting-with-haskell/libraries-and-frameworks/text-manipulation/json) 12 (/auth/login)
(http://www.facebook.com/sharer/sharer.php?u=https://www.schoolofhaskell.com/school/starting-with-haskell/libraries-and-frameworks/text-manipulation/json)
(https://plus.google.com/share?url=https://www.schoolofhaskell.com/school/starting-with-haskell/libraries-and-frameworks/text-manipulation/json)

Next content: HTML Parsing (https://www.schoolofhaskell.com/user/school/starting-with-haskell/libraries-and-frameworks/text-manipulation/tagsoup)

Go up to: Text manipulation (https://www.schoolofhaskell.com/school/starting-with-haskell/libraries-and-frameworks/text-manipulation)

Sections

Introduction

Plenty of data across the web is shared using the JSON format (http://tools.ietf.org /html/rfc4627), which is intended to be a simple and human-readable way to store data. It was designed to be a subset of JavaScript, so it defines objects as a list of name/value pairs.

Possible values in JSON are strings, numbers, booleans, null, arrays (one-dimensional) and objects. This is pretty much all you should know of JSON to work with it. Read the RFC 4627 linked before for more details about JSON.

JSON in Haskell

Many languages define parsers for JSON data, and Haskell is not an exception. The library we are using in this tutorial to parse JSON is `aeson` (http://hackage.haskell.org/package /aeson). Designed by Bryan O'Sullivan, `aeson` is a highly tuned, easy to use library. The module `Data.Aeson` contains further examples and explanations about the library. I encourage you to not stick to this tutorial and read the original documentation as well.

Alternatives and more tools for the use of JSON in Haskell exist. In fact, there is a separate category in Hackage with JSON libraries (http://hackage.haskell.org/package/#cat:json). In particular, the package `aeson-pretty` is useful to print JSON values in a more human-friendly way.

A tasteful example

An example of a JSON problem is now provided. We solve it paying attention to each step, so any newcomer in this topic can understand and reproduce the solution.

Suppose that we are running a survey about the relation between the age of a person and his/her taste for pizzas. *A priori*, our guess is that young people like pizzas more frequently than older people. However, we need data to verify or reject our hypothesis. We are storing this data in JSON format, and we want to run our statistical test in Haskell. Therefore, we need a way to read JSON to Haskell. This is exactly the kind of problem we are about to learn to solve!

Step 1: Define your type, collect your data

Probably, as in many others Haskell problems, it is a good idea to start defining types. We use types to describe the objects we are working with. Then we manipulate and transform these objects.

We begin then by defining the type `Person`, which will contain the information from a single person that we ask in the survey. In this case, first name, last name, age and whether if he/she likes pizzas or not using a Haskell record:

```
data Person =
  Person { firstName  :: !Text
         , lastName   :: !Text
         , age        :: Int
         , likesPizza :: Bool
           } deriving Show
```

In the other hand, we have the JSON data of the survey (conveniently hidden because of its length).

```
[
   { "firstName"  : "Daniel"
   , "lastName"   : "Díaz"
   , "age"        :  24
   , "likesPizza" :  true
     }
,
   { "firstName"  : "Rose"
   , "lastName"   : "Red"
   , "age"        :  39
   , "likesPizza" :  false
     }
,  { "firstName"  : "John"
   , "lastName"   : "Doe"
   , "age"        :  45
   , "likesPizza" :  false
     }
,  { "firstName"  : "Vladimir"
   , "lastName"   : "Vygodsky"
   , "age"        :  27
   , "likesPizza" :  false
     }
,  { "firstName"  : "Foo"
   , "lastName"   : "Bar"
   , "age"        :  32
   , "likesPizza" :  true
     }
,  { "firstName"  : "María"
   , "lastName"   : "Delaoh"
   , "age"        :  52
   , "likesPizza" :  false
     }
,  { "firstName"  : "Victoria"
   , "lastName"   : "Haskell"
   , "age"        :  23
   , "likesPizza" :  true
     }
,  { "firstName"  : "François"
   , "lastName"   : "Beaulieu"
   , "age"        :  42
   , "likesPizza" :  false
     }
,  { "firstName"  : "Amalie"
   , "lastName"   : "Baumann"
   , "age"        :  28
   , "likesPizza" :  true
     }
,  { "firstName"  : "Rachel"
```

```
   , "lastName"   : "Scott"
   , "age"        :  23
   , "likesPizza" :  true
     }
]
```

The `Data.Aeson` module of the `aeson` package exports the following functions:

```
decode :: FromJSON a => ByteString -> Maybe a
encode :: ToJSON a => a -> ByteString
eitherDecode :: FromJSON a => ByteString -> Either String a
```

Their names are pretty descriptive. The function `decode` will *decode* a JSON input stored as a `ByteString` value to a JSON type. The `encode` function will *encode* a value of JSON type `a` to a `ByteString` output value. The function `decodeEither` is similar to `decode`. Instead of returning the output inside the `Maybe` type, it does it inside the `Either String` type. This way, in case of unsuccessful decoding, it returns a `String` describing the error.

Step 2: Write instances for `FromJSON` and `ToJSON`.

However, looking at the type signatures, we see that to use these functions, our type must be an instance of the `FromJSON` class to decode, and of the `ToJSON` class to encode. Once we define these two instances for our type, we will be able to read/write JSON to/from Haskell.

No problem. Writing these instances is a straightforward process. In fact, the instances are similar to the type definition.

```
instance FromJSON Person where
 parseJSON (Object v) =
    Person <$> v .: "firstName"
           <*> v .: "lastName"
           <*> v .: "age"
           <*> v .: "likesPizza"
 parseJSON _ = mzero

instance ToJSON Person where
 toJSON (Person firstName lastName age likesPizza) =
    object [ "firstName"  .= firstName
           , "lastName"   .= lastName
           , "age"        .= age
           , "likesPizza" .= likesPizza
             ]
```

Remember to enable the `OverloadedStrings` (http://www.haskell.org/ghc/docs/7.6.2/html/users_guide/type-class-extensions.html#overloaded-strings) language extension in the top of your source file, which allows *stringy* types like `Text` to be written using string literals with no explicit conversion. Note also that, even if the JSON data you are parsing contains additional fields, the parser will succeed, ignoring unknown fields. However, if a field in *your type* is not present in the JSON data, an error will arise. If you want to have optional records, use (`.:?`) instead of (`.:`) in the JSON parser. For example, if the `firstName` field were optional, we would write:

```
instance FromJSON Person where
 parseJSON (Object v) =
    Person <$> v .:? "firstName"
           <*> v .:  "lastName"
           <*> v .:  "age"
           <*> v .:  "likesPizza"
```

Step 2: The smart alternative

While writing instances yourself is powerful and flexible, if they are simple mechanical translations like those above, then writing these instances is not only mechanical, but straightforward and boring. Even a machine could do it. Wait... That's it! Let the computer do it for you!

Haskell has a language extension called `DeriveGeneric` (http://www.haskell.org/ghc/docs/7.6.2/html/users_guide/deriving.html#deriving-typeable). Using this extension will allow you to tell Haskell to write the instance for you. Of course, code has been written previously to automatize this process for each different type class. To *derive* an instance for our type `Person` all we need to do is make it an instance of the `Generic` typeclass. No worries, this will be done automatically if you ask it to!

```
data Person =
  Person { firstName  :: !Text
         , lastName   :: !Text
         , age        :: Int
         , likesPizza :: Bool
           } deriving (Show,Generic)

instance FromJSON Person
instance ToJSON Person
```

I strongly recommend you derive the instances this way to avoid some trivial work and focus on other (probably harder) problems.

Step 3: Get the data in Haskell

So far, we know how to encode/decode Haskell values to/from `ByteString`s. We also have some JSON data somewhere, in a local file or over the network. It is time to learn how to bring our JSON file to Haskell to be able to parse it.

From a file

If the JSON data is currently stored in a local file, all we need to do is to specify the path to the file and read it using the `ByteString`s `readFile` function. We have to import the `Data.ByteString.Lazy` module qualified in order to avoid `ByteString`s `readFile` function name to clashes with the `Prelude`s `readFile`.

```
import qualified Data.ByteString.Lazy as B

jsonFile :: FilePath
jsonFile = "pizza.json"

getJSON :: IO B.ByteString
getJSON = B.readFile jsonFile
```

From a URL

If the JSON data is stored under a URL in the web, the procedure is similar. Instead of using `readFile` we use `simpleHttp` from the `http-conduit` package. This function will read the *response body* after following a given URL.

```
import qualified Data.ByteString.Lazy as B
import Network.HTTP.Conduit (simpleHttp)

jsonURL :: String
jsonURL = "http://daniel-diaz.github.io/misc/pizza.json"

getJSON :: IO B.ByteString
getJSON = simpleHttp jsonURL
```

Other sources

It should be clear that in each use case the way of getting the JSON data can vary enormously. Files and URL's are two very common ways, but it may not be enough for you. For example, it is very common to need some kind of authentication before requesting the JSON data. You can find below an example request for Twitter, which uses

OAuth for the authentication.

For the lazy: pre-made working program

Here you have a working example, putting everything together. Feel free to make any changes, since the code is editable. By default, the code reads the data from a URL. You may change the URL to any other JSON source, but note that you will have to change the type definition accordingly.

```
{-# START_FILE pizza.json #-}
[
   { "firstName"  : "Daniel"
   , "lastName"   : "Díaz"
   , "age"        :  24
   , "likesPizza" :  true
     }
,
   { "firstName"  : "Rose"
   , "lastName"   : "Red"
   , "age"        :  39
   , "likesPizza" :  false
     }
,  { "firstName"  : "John"
   , "lastName"   : "Doe"
   , "age"        :  45
   , "likesPizza" :  false
     }
,  { "firstName"  : "Vladimir"
   , "lastName"   : "Vygodsky"
   , "age"        :  27
   , "likesPizza" :  false
     }
,  { "firstName"  : "Foo"
   , "lastName"   : "Bar"
   , "age"        :  32
   , "likesPizza" :  true
     }
,  { "firstName"  : "María"
   , "lastName"   : "Delaoh"
   , "age"        :  52
   , "likesPizza" :  false
     }
,  { "firstName"  : "Victoria"
   , "lastName"   : "Haskell"
   , "age"        :  23
   , "likesPizza" :  true
     }
,  { "firstName"  : "François"
   , "lastName"   : "Beaulieu"
   , "age"        :  42
   , "likesPizza" :  false
     }
,  { "firstName"  : "Amalie"
   , "lastName"   : "Baumann"
   , "age"        :  28
   , "likesPizza" :  true
     }
,  { "firstName"  : "Rachel"
   , "lastName"   : "Scott"
   , "age"        :  23
   , "likesPizza" :  true
     }
]
-- show
-- /show
{-# START_FILE main.hs #-}
{-# LANGUAGE OverloadedStrings, DeriveGeneric #-}

import Data.Aeson
import Data.Text
import Control.Applicative
import Control.Monad
import qualified Data.ByteString.Lazy as B
import Network.HTTP.Conduit (simpleHttp)
import GHC.Generics

-- | Type of each JSON entry in record syntax.
data Person =
  Person { firstName  :: !Text
         , lastName   :: !Text
         , age        :: Int
         , likesPizza :: Bool
           } deriving (Show,Generic)

-- Instances to convert our type to/from JSON.

instance FromJSON Person
instance ToJSON Person

-- | Location of the local copy, in case you have it,
--   of the JSON file.
jsonFile :: FilePath
jsonFile = "pizza.json"

-- | URL that points to the remote JSON file, in case
--   you have it.
jsonURL :: String
jsonURL = "http://daniel-diaz.github.io/misc/pizza.json"

-- Move the right brace (}) from one comment to another
-- to switch from local to remote.

{--
-- Read the local copy of the JSON file.
getJSON :: IO B.ByteString
getJSON = B.readFile jsonFile
--}

{--}
```

```
-- Read the remote copy of the JSON file.
getJSON :: IO B.ByteString
getJSON = simpleHttp jsonURL
--}

main :: IO ()
main = do
 -- Get JSON data and decode it
 d <- (eitherDecode <$> getJSON) :: IO (Either String [Person])
 -- If d is Left, the JSON was malformed.
 -- In that case, we report the error.
 -- Otherwise, we perform the operation of
 -- our choice. In this case, just print it.
 case d of
  Left err -> putStrLn err
  Right ps -> print ps
```

Application: Rate Exchange JSON API

The web page http://rate-exchange.appspot.com (http://rate-exchange.appspot.com) provides a publicly available currency converter with output in JSON format and is obtained via URL's. For example, the URL

```
http://rate-exchange.appspot.com/currency?from=USD&to=EUR&q=1
```

gives us the following output:

```
{"to": "EUR", "rate": 0.74962518700000003, "from": "USD", "v":
0.74962518700000003}
```

This means that $1 is equivalent (in this moment) to 0.75€. Of course, we expect these values to change over time.

Step 1: Define a type for currency conversions

The meaning of the JSON output is:

- `to`: Target currency.
- `rate`: Rate of conversion.
- `from`: Currency of origin.
- `v`: Converted value in the target currency.

Therefore, a coherent type for this JSON would be:

```
{-# LANGUAGE DeriveGeneric #-}

import Data.Aeson
import GHC.Generics

data Conversion =
  Conversion { to :: !Text
             , rate :: Double
             , from :: !Text
             , v :: Double
               } deriving (Show, Generic)

instance FromJSON Conversion
instance ToJSON Conversion
```

Step 2: Obtaining the data from the URL

As we have seen before, it is easy to download the content from a URL using the function `simpleHttp` from the module `Network.HTTP.Conduit` of the `http-conduit` package. Therefore, given two strings representing the original and target currencies, and a quantity in the original currency, we get the JSON data of the conversion as follows.

```
getConversion :: Text -> Text -> Double -> IO (Maybe Conversion)
getConversion from to q =
  fmap decode $ simpleHttp $
      "http://rate-exchange.appspot.com/currency?from="
    ++ from ++ "&to=" ++ to ++ "&q=" ++ show q
```

It suffices now to extract the field `v` from the result to get the value in the converted currency.

```
-- | Convert a monetary value from one currency to another.
convert :: Double -- ^ Initial quantity.
        -> Text -- ^ Initial currency.
        -> Text -- ^ Target currency.
        -> IO (Maybe Double) -- ^ Result.
convert q from to = fmap (fmap v) $ getConversion from to q
```

Ready-to-use currency converter

We now have all the ingredients to have a working application that converts money within different currencies.

```
{-# LANGUAGE DeriveGeneric #-}

import Data.Aeson
import GHC.Generics
import Network.HTTP.Conduit (simpleHttp)
import Data.Text (Text)

-- | Type of conversion, analogous to the JSON data obtainable
--   from the URL.
data Conversion =
  Conversion { to :: !Text
             , rate :: Double
             , from :: !Text
             , v :: Double
               } deriving (Show, Generic)

-- Automatically generated instances

instance FromJSON Conversion
instance ToJSON Conversion

-- | Read the JSON data from the URL of a conversion, decoding it
```

Further application: Reading Twitter timelines

Twitter query results use the JSON format. Therefore, we can ask the Twitter API for the last statuses of a given profile and parse the response in the way we described above. Interestingly, the main issue here is to get the response, since Twitter ask us to authenticate using OAuth every single HTTP header. Fortunately, the package `authenticate-oauth` is in Hackage to help us.

Step 1: Register your application in Twitter

This is a prerequisite to authenticate our queries from version 1.1 of the Twitter REST API. If you already have a Twitter account, it will be very easy. Just go to https://dev.twitter.com/apps (https://dev.twitter.com/apps) and create a new application. It will automatically create a *read only* application with a *consumer key* and a *consumer secret*. You will also need to generate an *access token* and an *access token secret*. Once you have these tokens, you are ready to create your Twitter application.

Step 2: Create your `OAuth` and `Credential` values

We now group together our keys using the `authenticate-oauth` package. Particularly, using the `OAuth` and `Credential` types.

```
import Web.Authenticate.OAuth

myoauth :: OAuth
myoauth =
  newOAuth { oauthServerName     = "api.twitter.com"
           , oauthConsumerKey    = "your consumer key here"
           , oauthConsumerSecret = "your consumer secret here"
             }

mycred :: Credential
mycred = newCredential "your access token here"
                       "your access token secret here"
```

Step 3: Define a type for tweets

As we before parsed the JSON data to a list of `Person` type values, we need to define a type for each tweet (naturally named `Tweet`). When requesting the timeline statuses of a Twitter profile, the response JSON will contain a list of tweets. It is our job to select the information we want from each tweet, and to define a type to store them in Haskell. Depending on your needs, you will include different fields in the `Tweet` type. I have included only the `text` and `created_at` fields to keep the example simple.

```
import Data.Text (Text)
import Data.Time.Clock (UTCTime)
import Data.Aeson
import GHC.Generics

data Tweet =
  Tweet { text :: !Text
        , created_at :: !UTCTime
          } deriving (Show, Generic)

instance FromJSON Tweet
instance ToJSON Tweet
```

The nice thing here is that, when parsing the JSON data, our parser will ignore the fields we didn't define in our type.

Step 4: Timeline request and parsing

We are going to check for the five latest uploaded packages to Hackage using the Hackage twitter (https://twitter.com/Hackage). The URL we make the request to is `https://api.twitter.com/1.1/statuses/user_timeline.json?screen_name=Hackage`. You can see more info about this request here (https://dev.twitter.com/docs/api/1.1/get/statuses/user_timeline). Using the `Network.HTTP.Conduit` interface, we send the GET HTTP request.

```
import Network.HTTP.Conduit

timeline :: String -- ^ Screen name of the user
         -> IO (Either String [Tweet]) -- ^ If there is any error parsing the J
SON data, it
                                       --   will return 'Left String', where th
e 'String'
                                       --   contains the error information.
timeline name = do
  -- Firstly, we create a HTTP request with method GET (it is the default so we
 don't have to change that).
  req <- parseUrl $ "https://api.twitter.com/1.1/statuses/user_timeline.json?sc
reen_name=" ++ name
  -- Using a HTTP manager, we authenticate the request and send it to get a res
ponse.
  res <- withManager $ \m -> do
           -- OAuth Authentication.
           signedreq <- signOAuth myoauth mycred req
           -- Send request.
           httpLbs signedreq m
  -- Decode the response body.
  return $ decodeEither $ responseBody res
```

The function we have created reads the timeline of any user who shares his/her timeline publicly. Now we read the timeline from Hackage and restrict the result to the last five tweets.

```
main :: IO ()
main = do
  -- Read the timeline from Hackage user.
  ets <- timeline "Hackage"
  case ets of
   -- When the parsing of the JSON data fails, we report it.
   Left err -> putStrLn err
   -- When successful, print in the screen the first 5 tweets.
   Right ts  -> mapM_ print $ take 5 ts
```

Putting everything together

Here is an editable version of the code above, that will allow you to read any Twitter timeline, using Haskell and our JSON parsing techniques. We have learnt that the JSON data can come from very different sources, but the process of parsing it is always similar. Define a type corresponding to the JSON data you are reading (or even a subset of it), and use `decode` or `decodeEither` to transform the JSON input to Haskell values you can handle.

```
{-# LANGUAGE OverloadedStrings, DeriveGeneric #-}

import Data.ByteString (ByteString)
import Network.HTTP.Conduit
import Web.Authenticate.OAuth
import Data.Aeson
import Data.Time.Clock (UTCTime)
import Data.Text (Text)
import GHC.Generics

-- Insert here your own credentials

myoauth :: OAuth
myoauth =
  newOAuth { oauthServerName     = "api.twitter.com"
           , oauthConsumerKey    = "your consumer key here"
           , oauthConsumerSecret = "your consumer secret here"
             }

mycred :: Credential
mycred = newCredential "your access token here"
                       "your access token secret here"

-- | Type for tweets. Use only the fields you are interested in.
--   The parser will filter them. To see a list of available fields
--   see <https://dev.twitter.com/docs/platform-objects/tweets>.
data Tweet =
  Tweet { text :: !Text
        , created_at :: !UTCTime
          } deriving (Show, Generic)

instance FromJSON Tweet
instance ToJSON Tweet

-- | This function reads a timeline JSON and parse it using the 'Tweet' type.
timeline :: String -- ^ Screen name of the user
         -> IO (Either String [Tweet]) -- ^ If there is any error parsing the JSO
                                       --   will return 'Left String', where the
                                       --   contains the error information.
timeline name = do
  -- Firstly, we create a HTTP request with method GET (it is the default so we d
  req <- parseUrl $ "https://api.twitter.com/1.1/statuses/user_timeline.json?scre
  -- Using a HTTP manager, we authenticate the request and send it to get a respo
  res <- withManager $ \m -> do
           -- OAuth Authentication. 'signOAuth' modifies the HTTP header adding t
           -- appropriate authentication.
           signedreq <- signOAuth myoauth mycred req
           -- Send request.
           httpLbs signedreq m
  -- Decode the response body.
  return $ eitherDecode $ responseBody res

-- | The main function, as an example of how to use the 'timeline'
--   function.
main :: IO ()
main = do
  -- Read the timeline from Hackage user. Feel free to change the screen
  -- name to any other.
  ets <- timeline "Hackage"
  case ets of
   -- When the parsing of the JSON data fails, we report it.
   Left err -> putStrLn err
   -- When successful, print in the screen the first 5 tweets.
   Right ts  -> mapM_ print $ take 5 ts
```

Conclusion

We have seen how to use `aeson` to easily parse JSON data, and some examples where we apply this knowledge. Currently, there are thousands of websites using the JSON format, so you can now think how to use it in your particular case. The provided examples show the pattern to follow. We define a datatype using record syntax and fields according to the JSON data. Then we obtain the desired JSON from a local or remote source. We have seen that there are different approaches to access to sources, mostly depending on the case. Finally we use `decode` or `decodeEither` to do the actual decoding. Haskell is great at parsing. With little code, you can produce both *correct* and *fast* parsers. The code is also easy to follow, and, as a consequence easy to change, giving Efficiency in both development and execution.