**sheets-db: Database powered by Google Spreadsheets API**

**Abstract.** sheets-db is a Haskell binding to Google Sheets API. sheets-db allows Haskell users to utilize google spread sheets as a light weight database. It provides various functions to create, read, update and delete rows in spreadsheets along with a way to construct simple structured queries.

1. **Introduction**

Traditionally, applications use different databases such as Oracle, Postgres, MongoDB, MySQL, etc to store data. Setting up this type of traditional database is a painful process which involves hosting database server, database management and requires people with special technical capabilities to retrieve and use information. The powerful features offered by traditional databases might not be required to certain class of applications. In this case using these traditional databases is an overkill. Non-technical users cannot read or manipulate data easily without knowledge of languages like SQL.

Sheets-db library tries to solve this problem by utilizing the spreadsheets as light weight database to work with data avoiding pains involved with traditional databases. Google’s default spreadsheets interface can be utilized by non-technical users to read and modify data easily. Although, sheets-db does not offer powerful features as traditional databases, it is reliable and complete to achieve purposes of small applications.

1. **How to use sheets-db**

**Obtaining Key of a Spreadsheet**

On a user’s google account create a spreadsheet with required column names. A spreadsheet is uniquely identified by google using a key which is assigned when it is first created. The URL looks like:

https://docs.google.com/spreadsheets/d/1hIEq4AAauzI8INelQRIvgxBhmzX44qAB\_1QQpFJQ2Xo/edit

The key of the spreadsheet created is highlighted is required for sheet-db to identify your spreadsheet.

**Registering your application on Google Developers Console for OAuth**

The spreadsheet created belongs to a particular user and is not public. To access non-public user spreadsheets, the web requests need to pass an authentication token that validates the usage of data and also the application accessing it.

The protocol OAuth 2.0 must be followed to accomplish the above process.

A developer who wishes to use sheets-db in their application must follow the below steps to generate <client id> and <client secret> required for OAuth authentication process. Generating the client id and secret gives an application unique identity.

1. We must obtain OAuth 2.0 credentials (client id and secret) from the [Google Developers Console](https://console.developers.google.com/).
2. After clicking the link above you will be redirected to login using your google account.
3. After login click on the left side top corner navigation button to draw out the side panel.
4. Click the API Manager Menu item.
5. In the sub menu of API manager click on the Credentials item. Here you can create a new application which gets unique client id and secret for that application.
6. Create a new application by clicking on “New Credentials” drop down to choose “OAuth Client ID”.
7. Then choose the application type as “other” and give your application a name you wish and click on create.
8. A new application gets created. Client ID and Client Secret are displayed against the application’s row. Copy these credentials for use in your application.

**Creating the Spreadsheet database table**

The first row of the worksheet is a header row and must contain all the column names. A column without header name is not considered part of data. When a spreadsheet is first created appropriate header names must be manually created without spaces.

**Datatypes in sheets-db**

Each worksheet is represented as a Sheet.

**data** Sheet = Sheet {

key :: String -- key of the spreadsheet  
 , worksheetId :: String -- current worksheet id  
 , url :: String –- URL of the spreadsheet  
 , columns :: [ColName] –- column names  
 , cid :: String –- client id created in dev console  
 , csecret :: String –- client secret created   
 } **deriving** (Show,Eq)

ColName is of Text type and represents a column name in spreadsheet.

A spreadsheet data table is made from list of “Row”s. A Row is made of list of “Cell”s where each cell data corresponds to respective header column name.

**type** Row = [Cell]

**data** Cell = (:=) {colname :: !ColName, value :: Value} **deriving** (Typeable, Eq)

(=:) :: (Val v) => ColName -> v -> Cell

Each Cell is a key value pair of column name and the corresponding data from the spreadsheet. Special symbol function =: can be used to construct key value pairs of Cell. The valid values in the cell are instances of class Val.

-- | Haskell types of this class correspond to Cell value types

**class** (Typeable a, Show a, Eq a) => Val a **where**

val :: a -> Value

cast' :: Value -> Maybe a

**instance** Val T.Text

**instance** Val String

**instance** Val Scientific

**instance** Val Bool

**instance** Val (Maybe Value)

**instance** Val UTCTime

**instance** Val POSIXTime

**instance** Val Float

**instance** Val Double

The 2 methods of the class Val are val and cast’. val specifies how a particular data is converted to Value type and cast’ shows how a Value can be converted back to its original datatype.

cast :: forall m a. (Val a, Monad m) => Value -> m a

The utility method cast should be used instead of cast’ to convert a Value back to its original type.

Finally, the Value datatype:

**data** Value =

Number !Scientific |

String !T.Text |

Bool Bool |

UTC UTCTime |

Null

**deriving** (Typeable, Eq)

**instance** Show Value

sheets-db allows a user to perform simple structured queries based on the abilities given by google sheets API. A structured query can be constructed using the Selector datatype.

**data** Selector = Gtr ColName Value

| Gteqr ColName Value

| Ltr ColName Value

| Lteqr ColName Value

| Eqr ColName Value

| And Selector Selector

| Or Selector Selector

| Empty

**deriving** (Show, Eq)

(~>) :: (Val v) => ColName -> v -> Selector

k ~> v = Gtr k (val v)

(~>=) :: (Val v) => ColName -> v -> Selector

k ~>= v = Gteqr k (val v)

(~<) :: (Val v) => ColName -> v -> Selector

k ~< v = Ltr k (val v)

(~<=) :: (Val v) => ColName -> v -> Selector

k ~<= v = Lteqr k (val v)

(~=) :: (Val v) => ColName -> v -> Selector

k ~= v = Eqr k (val v)

(~&&~) :: Selector -> Selector -> Selector

s ~&&~ t = And s t

(~||~) :: Selector -> Selector -> Selector

s ~||~ t = Or s t

sheets-db also allows a user to sort based on a specific column either in ascending or descending order using the sort value of type Order in Query data.

**data** Order = Order {

colname :: ColName,

reverse :: Bool

} | NoOrder -- Indicates no specific order is required

**deriving** (Show, Eq)

A query example may look like:

rows <- find (Query (Select ((p "type" ~< (2 :: Float)) ~||~   
 (p "category" ~= "meat" ) sheet) 0 10   
 (Order (p "category") False))

**Initialize the sheet datatype**

Various operations can be performed on a spreadsheet but before that sheet datatype mentioned above must be initialized. To do that access method must be used.

access :: String -- key of the spreadsheet

-> String -- worksheet id of current spreadsheet

-> String -- client Id of the application created in Dev   
 Console

-> String -- client Secret created in Dev Console

-> IO (Maybe Sheet)

The above method constructs Sheet data if all the credentials are correct.

**Insert a Row into Spreadsheet**

After initializing the Sheet data, various operations can be performed. To insert a new row into the spreadsheet a Row must be constructed using the [ColName] in the Sheet data and (=:) operator function.

Row construction:

For a spreadsheet with column names: name, category, healthiness, type. An example row can be constructed by

toInsert :: Row

toInsert= [p "name" =: "papaya",

p "category" =: "fruit",

p "healthiness" =: "adequate",

p "type" =: (3 :: Float)]

Here p is alias for Text.pack function which is used very frequently.

And insert function is

-- Adds a new row to google sheet after validation of data to be inserted

insert :: Row –- The row to be inserted containing all column   
 values

-> Sheet -- The spreadsheet in which the data must be   
 inserted

-> IO Outcome

Example insertion

clientid = "1086783968140-knfg08qu9onnn5b485veaskt2flr0loa.apps.googleusercontent.com"

clientsecret ="wRktcD6xif1\_z6Xv-RhCatBB"

toInsert :: Row

toInsert= [p "name" =: "papaya",

p "category" =: "fruit",

p "healthiness" =: "adequate",

p "type" =: (3 :: Float)]

main = **do**

mySheet <- access "1hIEq4AAauzI8INelQRIvgxBhmzX44qAB\_1QQpFJQ2Xo"   
 "od6" clientid clientsecret

**case** mySheet **of**

Just sheet -> **do**

result <- insert toInsert sheet

print result

Nothing -> putStrLn "Access to Sheet failed"

**Remove a Row from Spreadsheet**

A Row can be removed by using the remove method. The Row to be removed must contain the id of the Row else the operation fails.

-- Used to delete a Row from the Google Sheet.

remove :: Sheet -- The spreadsheet in which a row needs to be   
 removed

-> Row -- The row which is to be deleted with “id”

-> IO Outcome

**Update a Row in Spreadsheet**

After a row is fetched from spreadsheet it can be changed and inserted back into the spreadsheet using the update function. The “id” field must exist else the operation fails. The operation also fails if all the column names are not present including the original cells which do not change.

update :: Sheet -– The spreadsheet in which a row needs to be updated

-> Row -- Row to be updated along with “id” of row to be updated

-> IO Outcome

**Find Rows in Spreadsheet**

Structured queries can be constructed and rows satisfying these criteria are returned using the find function.

find :: Query -- Query to be run includes the sheet

-> IO (Either String [Row])

**data** Selection = Select {

selector :: Selector

, sheet :: Sheet

} **deriving** (Show, Eq)

-- Selects rows in spreadsheet that match selector

-- Query

-- Use 'Select' to create a basic query with defaults, then modify if desired. For example, @(Select sel sheet) {limit = 10}@

**data** Query = Query {

selection :: Selection,

skip :: Int, -- Number of initial matching documents to   
 skip. Default = 0

limit :: Int, -- Maximum number of documents to return, 0 = no   
 limit. Default = 0

sort :: Order -- Sort results by this order, NoOrder = no   
 sort. Default = NoOrder

} **deriving** (Show, Eq)

**data** Order = Order {

colname :: ColName,

reverse :: Bool

} | NoOrder **deriving** (Show, Eq)

query :: Selector -> Sheet -> Query

Example of find and query construction:

clientid = "1086783968140-knfg08qu9onnn5b485veaskt2flr0loa.apps.googleusercontent.com"

clientsecret ="wRktcD6xif1\_z6Xv-RhCatBB"

toInsert :: Row

toInsert= [p "name" =: "papaya",

p "category" =: "fruit",

p "healthiness" =: "adequate",

p "type" =: (3 :: Float)]

main = **do**

mySheet <- access "1hIEq4AAauzI8INelQRIvgxBhmzX44qAB\_1QQpFJQ2Xo"   
 "od6" clientid clientsecret

**case** mySheet **of**

Just sheet -> **do**

rows <- find (query

((p "type" ~< (2 :: Float)) ~||~

(p "category" ~= "meat" )) sheet)

print rows

Nothing -> putStrLn "Invalid"

1. **Approach**

**Google Spreadsheet URL construction**

sheets-db communicates with Google sheets REST API to achieve various functions described above. Different URLs are constructed based on the requirement.

The basic URL used throughout the code is:

https://spreadsheets.google.com/feeds/list/<key>/<worksheetId>/private/full

This URL is known as list feed URL. A spreadsheet is uniquely identified by <key> and each spreadsheet can contain multiple worksheets which is uniquely identified by <worksheetId>.

**URL to fetch rows**

A GET request is sent to the list feed URL to retrieve the rows in the spreadsheet. This GET request returns content in XML but sheets-db retrieves rows in JSON format using the URL:

https://spreadsheets.google.com/feeds/list/<key>/<worksheetId>/private/full?alt=json

To get only the rows that meet a specified criteria structured query parameter “sq” can be used to construct query criteria.

https://spreadsheets.google.com/feeds/list/<key>/<worksheetId>/private/full?alt=json&sq=<encoded-query>

A query example from Sheets API itself

age > 25 and height < 175

Every query should be constructed in this way and should be encoded to use as a parameter value with sq in the above URL.

**URL to add new rows**

To add a new row to the spreadsheet a post request should be sent to list feed URL along with the new row xml constructed as in the example below.

POST https://spreadsheets.google.com/feeds/list/<key>/<worksheetId>/private/full

<entry xmlns="http://www.w3.org/2005/Atom"  
    xmlns:gsx="http://schemas.google.com/spreadsheets/2006/extended">  
  <gsx:hours>1</gsx:hours>  
  <gsx:ipm>1</gsx:ipm>  
  <gsx:items>60</gsx:items>  
  <gsx:name>Elizabeth Bennet</gsx:name>  
</entry>

The xml example is taken from Google sheets API documentation. In this xml hours, ipm, items, name correspond to the column names and the values of each xml node correspond to the values of the columns.

**URL to update a row**

An example row in the JSON format looks like:

**{**

"id"**:** **{**

"$t"**:** "https://spreadsheets.google.com/feeds/list/1hIEq4AAauzI8INelQRIvgxBhmzX44qAB\_1QQpFJQ2Xo/od6/private/full/chk2m"

**},**

"updated"**:** **{**

"$t"**:** "2015-11-25T19:08:07.834Z"

**},**

"category"**:** **[**

**{**

"scheme"**:** "http://schemas.google.com/spreadsheets/2006"**,**

"term"**:** "http://schemas.google.com/spreadsheets/2006#list"

**}**

**],**

"title"**:** **{**

"type"**:** "text"**,**

"$t"**:** "Carrot"

**},**

"content"**:** **{**

"type"**:** "text"**,**

"$t"**:** "category: Vegetable, healthiness: Adequate, type: 1"

**},**

"link"**:** **[**

**{**

"rel"**:** "self"**,**

"type"**:** "application/atom+xml"**,**

"href"**:** "https://spreadsheets.google.com/feeds/list/1hIEq4AAauzI8INelQRIvgxBhmzX44qAB\_1QQpFJQ2Xo/od6/private/full/chk2m"

**},**

**{**

"rel"**:** "edit"**,**

"type"**:** "application/atom+xml"**,**

"href"**:** "https://spreadsheets.google.com/feeds/list/1hIEq4AAauzI8INelQRIvgxBhmzX44qAB\_1QQpFJQ2Xo/od6/private/full/chk2m/9a586cd3cmc72"

**}**

**],**

"gsx$name"**:** **{**

"$t"**:** "Carrot"

**},**

"gsx$category"**:** **{**

"$t"**:** "Vegetable"

**},**

"gsx$healthiness"**:** **{**

"$t"**:** "Adequate"

**},**

"gsx$type"**:** **{**

"$t"**:** "1"

**}**

**}**

To update this row we must send a PUT request to the edit URL along with the xml of the updated row. Edit URL can be parsed by finding value of “href” of “rel” : “edit” in “link” section array.

**URL to remove a row**

In the above JSON when we send http DELETE request to the id URL the row will be deleted.

DELETE https://spreadsheets.google.com/feeds/list/<key>/<worksheetId>/private/full/<Row ID>

**Module GoogleRequest**

To achieve various functions, GET, POST, PUT, DELETE HTTP requests need to be constructed and send to Google sheet API server. To make these requests along with OAuth authentication, methods get, post, put, delete respectively are created in GoogleRequest.

These methods are utilized by SheetDB module.

**OAuth Authentication and google-oauth2 library**

Before an application can access private data using any Google API, it must obtain an access token that grants access to that API. A variable parameter called scope controls the set of resources and operations that an access token permits.

During the access token request, our application sends one or more values in the scope parameter.

For google sheets API the scope we must send is

https://spreadsheets.google.com/feeds

When our application needs access to sheets data, it asks Google for above scope of access.

Google displays an OAuth dialog to the user, asking them to authorize your application to request their private data.

If the user approves, then Google gives our application a short-term access token.

Our application can then perform different http requests (GET, POST, PUT, and DELETE) by attaching the access token to the request.

If Google determines that our request and the token are valid, it performs the HTTP request.

The [Google-oauth2](https://hackage.haskell.org/package/google-oauth2) library on hackage interacts with the Google OAuth2 authorization API to perform these functions for us.

The example taken from the above link illustrates the process involved in using the library.

The steps involved are:

1. Prompt the user for a verification code

2. POST that code to the Google API for a set of tokens (access and refresh)

3. Use the access token until it expires

4. Use the refresh token to get a new access token

5. Repeat from 3

**import** Data.Monoid

**import** Network.Google.OAuth2

**import** Network.HTTP.Conduit

**import** Network.HTTP.Types (hAuthorization)

**import** **qualified** Data.ByteString.Char8 **as** B8

**import** **qualified** Data.ByteString.Lazy.Char8 **as** L8

main :: IO ()

main = **do**

**let** client = OAuth2Client clientId clientSecret

scopes = ["https://www.googleapis.com/auth/drive"]

token <- getAccessToken client scopes Nothing

request <- parseUrl "https://www.googleapis.com/drive/v2/files"

response <- withManager $ httpLbs $ authorize token request

L8.putStrLn $ responseBody response

**where**  
 authorize token request = request{ requestHeaders = [(hAuthorization,   
 B8.pack $ "Bearer " <> token)] }

-- Setup in Google Developers Console

clientId = "..."

clientSecret = "..."

The above example demonstrates the HTTP request process with the Google Drive API and it is similar for Google Sheets API. The exact steps are followed in GoogleRequest module for completing the authentication.

The library provides function called:

getAccessToken :: OAuth2Client

-> [OAuth2Scope]

-> Maybe FilePath --File in which to cache the token

-> IO OAuth2Token -- ^ Refreshed token

Which takes OAuth2Client (ClientID and Secret), scope [https://spreadsheets.google.com/feeds], and a temp file for caching the credentials once obtained from the server.

When we supply these parameters to the function and if it executes for the first time it prompts the user for a verification code. It displays a url which we need to use to generate verification code from the browser by authenticating with the google account that owns the sheet. After obtaining the verification code, supply to the prompt.

Then the library uses the verification code to obtain a set of tokens (access and refresh) from Google OAuth2 authorization API. The credentials which are obtained are stored in the file we supplied as parameter, for reuse. We then use the access token for subsequent requests until it expires. We use the refresh token to get a new access token.

These above steps are all done by getAccessToken function.

In GoogleRequest there are various functions that perform the above tasks by supplying correct parameters.

scopes = ["https://spreadsheets.google.com/feeds"]

createToken client = getAccessToken client scopes (Just "./key.txt")

authorize token request = request

{ requestHeaders = [(hAuthorization, B8.pack $ "Bearer " ++ token)] }

The above authorize function is used to add the token header to the request to be sent to Google Sheets API for authentication.

**HTTP requests with http-conduit**

GoogleRequest uses http-conduit for making web requests.

httpLbs :: MonadIO m => Request -- Request to make

-> Manager -- Connection manager

-> m (Response ByteString)

httpLbs method is used to make HTTP requests. The Request data is constructed based on the function being performed.

The below table gives an overview of what are the contents of the Request in each get, post, put, delete methods of GoogleRequest and their return types.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| GoogleRequest methods | Request method | Request Headers | Request Body | Return type | URL |
| get | GET | Authorization | None | responseBody | List feed URL |
| post | POST | Authorization, content-type, Gdata-version | XML of the row data | responseStatus | List feed URL |
| put | PUT | Authorization, content-type, Gdata-version | XML of row data to be modified | responseStatus | Edit URL of the row |
| delete | DELETE | Authorization | None | responseStatus | Id URL of the row |

**Module SheetDB**

This module exposes core functionality of the library and is considered as a main interface to the outside world. The main methods are find, insert, update, and remove.

Each method involves figuring out following and making a request using GoogleRequest.

Remaining steps happen in module SheetDB.

Happens in GoogleRequest through get, post, put, delete methods

End

Parse response body if applicable

Send HTTP request

Construct Request Body if applicable

Construct Headers

Decide HTTP request method

Find or Construct URL

Start

**Find method**

find method in SheetDB is used to get the rows from a spreadsheet and perform basic structured queries. The steps in find operation are:

End

Parse JSON from GET operation

Return Just [Row] or Nothing

Perform HTTP GET with get from GoogleRequest using constructed URL

Start

Construct Selection string for ‘sq’ and Order string for orderby parameter

Construct list feed URL with (alt=json), (sq=<selection\_string>), (orderby=<order\_string>) params

**Construction of find URL**

find method takes Query type as input and Query constructor takes Selection and Order type data.

The Selection data is converted to string, encoded and passed as ‘sq’ parameter value in list feed URL.

**data** Selector = Gtr ColName ST.Value

| Gteqr ColName ST.Value

| Ltr ColName ST.Value

| Lteqr ColName ST.Value

| Eqr ColName ST.Value

| And Selector Selector

| Or Selector Selector

| Empty

**deriving** (Show, Eq)

-- Converts Selector to String. example: Gtr "age" 25 -> age > 25

selectorQueryUrl :: Selector -> String

selectorQueryUrl (Gtr colname value) = T.unpack colname ++ " > " ++ show value

selectorQueryUrl (Gteqr colname value) = T.unpack colname ++ " >= " ++ show value

selectorQueryUrl (Ltr colname value) = T.unpack colname ++ " < " ++ show value

selectorQueryUrl (Lteqr colname value) = T.unpack colname ++ " <= " ++ show value

selectorQueryUrl (Eqr colname value) = T.unpack colname ++ " = " ++ show value

selectorQueryUrl (And sel1 sel2) = selectorQueryUrl sel1 ++ " and " ++ selectorQueryUrl sel2

selectorQueryUrl (Or sel1 sel2) = selectorQueryUrl sel1 ++ " or " ++ selectorQueryUrl sel2

selectorQueryUrl Empty = ""

-- Encodes Query parameter

encodeSelectorQueryUrl :: String -> String

encodeSelectorQueryUrl q = B8.unpack $ urlEncode True $ B8.pack q

The Order data in Query specifies the sort order based on a particular column. Google Sheets API currently supports sorting only on single column. The Order data also has “reverse” constructor which specified ascending or descending order of rows. The Order data is converted to string and encoded as other parameters.

**data** Order = Order {

colname :: ColName,

reverse :: Bool

} | NoOrder -- Indicates no specific order is required

**deriving** (Show, Eq)

All the parameters “sq”, “orderby”, “reverse”, “alt” are all created in makeQueryUrl method

makeQueryUrl :: Query -> Maybe String

makeQueryUrl q = **do**

**let** spreadsheet = sheet $ selection q

**let** order = sort q

**let** param1 = [("alt","json")]::[(String,String)]

**let** selectString = selectorQueryUrl $ selector $ selection q

**let** param2 = **if** selectString == "" **then** param1 **else** ("sq", selectString ) : param1

**let** params = **case** order **of**

NoOrder -> param2

Order col rev -> param2 ++ [("orderby","column:" ++ T.unpack col),("reverse", map toLower (show rev))]

url <- formURL (key spreadsheet ) (worksheetId spreadsheet) params

return $ exportURL url

List feed URL template used is:

urlTemplate = "https://spreadsheets.google.com/feeds/list/${key}/${worksheetid}/private/full" :: String

The above template is converted to find URL with relevant parameters using formURL method.

-- Takes spreadsheet key and worksheet id along with parameters to construct URL

formURL :: String -> String-> [(String,String)] -> Maybe URL

formURL key worksheetid params = **do**

**let** template = T.pack urlTemplate

**let** keyPattern = T.pack "${key}"

**let** worksheetidPattern = T.pack "${worksheetid}"

**let** keyUrl = T.replace keyPattern (T.pack key) template

**let** keyWorksheetUrl = T.replace worksheetidPattern (T.pack worksheetid) keyUrl

url <- importURL $ T.unpack keyWorksheetUrl

return $ foldl add\_param url params

**Parsing find response JSON**

After an HTTP GET request is performed on above Query URL a JSON is returned, which is parsed and rows are extracted.

An example of JSON returned:

**{**

"version"**:** "1.0"**,**

"encoding"**:** "UTF-8"**,**

"feed"**:** **{**

"xmlns"**:** "http://www.w3.org/2005/Atom"**,**

"xmlns$openSearch"**:** "http://a9.com/-/spec/opensearchrss/1.0/"**,**

"xmlns$gsx"**:** "http://schemas.google.com/spreadsheets/2006/extended"**,**

"id"**:** **{**

"$t"**:** "https://spreadsheets.google.com/feeds/list/1hIEq4AAauzI8INelQRIvgxBhmzX44qAB\_1QQpFJQ2Xo/od6/private/full"

**},**

"updated"**:** **{**

"$t"**:** "2015-11-25T19:08:07.834Z"

**},**

"category"**:** **[**

**{**

"scheme"**:** "http://schemas.google.com/spreadsheets/2006"**,**

"term"**:** "http://schemas.google.com/spreadsheets/2006#list"

**}**

**],**

"title"**:** **{**

"type"**:** "text"**,**

"$t"**:** "Sheet1"

**},**

"link"**:** **[**

**{**

"rel"**:** "alternate"**,**

"type"**:** "application/atom+xml"**,**

"href"**:** "https://docs.google.com/spreadsheets/d/1hIEq4AAauzI8INelQRIvgxBhmzX44qAB\_1QQpFJQ2Xo/edit"

**},**

**{**

"rel"**:** "http://schemas.google.com/g/2005#feed"**,**

"type"**:** "application/atom+xml"**,**

"href"**:** "https://spreadsheets.google.com/feeds/list/1hIEq4AAauzI8INelQRIvgxBhmzX44qAB\_1QQpFJQ2Xo/od6/private/full"

**},**

**{**

"rel"**:** "http://schemas.google.com/g/2005#post"**,**

"type"**:** "application/atom+xml"**,**

"href"**:** "https://spreadsheets.google.com/feeds/list/1hIEq4AAauzI8INelQRIvgxBhmzX44qAB\_1QQpFJQ2Xo/od6/private/full"

**},**

**{**

"rel"**:** "self"**,**

"type"**:** "application/atom+xml"**,**

"href"**:** "https://spreadsheets.google.com/feeds/list/1hIEq4AAauzI8INelQRIvgxBhmzX44qAB\_1QQpFJQ2Xo/od6/private/full?alt=json"

**}**

**],**

"author"**:** **[**

**{**

"name"**:** **{**

"$t"**:** "adityaravikanti.vnit"

**},**

"email"**:** **{**

"$t"**:** "adityaravikanti.vnit@gmail.com"

**}**

**}**

**],**

"openSearch$totalResults"**:** **{**

"$t"**:** "2"

**},**

"openSearch$startIndex"**:** **{**

"$t"**:** "1"

**},**

"entry"**:** **[**

**{**

"id"**:** **{**

"$t"**:** "https://spreadsheets.google.com/feeds/list/1hIEq4AAauzI8INelQRIvgxBhmzX44qAB\_1QQpFJQ2Xo/od6/private/full/cokwr"

**},**

"updated"**:** **{**

"$t"**:** "2015-11-25T19:08:07.834Z"

**},**

"category"**:** **[**

**{**

"scheme"**:** "http://schemas.google.com/spreadsheets/2006"**,**

"term"**:** "http://schemas.google.com/spreadsheets/2006#list"

**}**

**],**

"title"**:** **{**

"type"**:** "text"**,**

"$t"**:** "Pork Shoulder"

**},**

"content"**:** **{**

"type"**:** "text"**,**

"$t"**:** "category: meat, healthiness: questionable, type: 1"

**},**

"link"**:** **[**

**{**

"rel"**:** "self"**,**

"type"**:** "application/atom+xml"**,**

"href"**:** "https://spreadsheets.google.com/feeds/list/1hIEq4AAauzI8INelQRIvgxBhmzX44qAB\_1QQpFJQ2Xo/od6/private/full/cokwr"

**},**

**{**

"rel"**:** "edit"**,**

"type"**:** "application/atom+xml"**,**

"href"**:** "https://spreadsheets.google.com/feeds/list/1hIEq4AAauzI8INelQRIvgxBhmzX44qAB\_1QQpFJQ2Xo/od6/private/full/cokwr/14nmk169a0mpo7"

**}**

**],**

"gsx$name"**:** **{**

"$t"**:** "Pork"

**},**

"gsx$category"**:** **{**

"$t"**:** "meat"

**},**

"gsx$healthiness"**:** **{**

"$t"**:** "questionable"

**},**

"gsx$type"**:** **{**

"$t"**:** "1"

**}**

**},**

**{**

"id"**:** **{**

"$t"**:** "https://spreadsheets.google.com/feeds/list/1hIEq4AAauzI8INelQRIvgxBhmzX44qAB\_1QQpFJQ2Xo/od6/private/full/cpzh4"

**},**

"updated"**:** **{**

"$t"**:** "2015-11-25T19:08:07.834Z"

**},**

"category"**:** **[**

**{**

"scheme"**:** "http://schemas.google.com/spreadsheets/2006"**,**

"term"**:** "http://schemas.google.com/spreadsheets/2006#list"

**}**

**],**

"title"**:** **{**

"type"**:** "text"**,**

"$t"**:** "Bubblegum"

**},**

"content"**:** **{**

"type"**:** "text"**,**

"$t"**:** "category: candy, healthiness: Super High, type: 1"

**},**

"link"**:** **[**

**{**

"rel"**:** "self"**,**

"type"**:** "application/atom+xml"**,**

"href"**:** "https://spreadsheets.google.com/feeds/list/1hIEq4AAauzI8INelQRIvgxBhmzX44qAB\_1QQpFJQ2Xo/od6/private/full/cpzh4"

**},**

**{**

"rel"**:** "edit"**,**

"type"**:** "application/atom+xml"**,**

"href"**:** "https://spreadsheets.google.com/feeds/list/1hIEq4AAauzI8INelQRIvgxBhmzX44qAB\_1QQpFJQ2Xo/od6/private/full/cpzh4/1cm513nm0e853k"

**}**

**],**

"gsx$name"**:** **{**

"$t"**:** "Bubblegum"

**},**

"gsx$category"**:** **{**

"$t"**:** "candy"

**},**

"gsx$healthiness"**:** **{**

"$t"**:** "Super High"

**},**

"gsx$type"**:** **{**

"$t"**:** "1"

**}**

**}**

**]**

**}**

**}**

In the above JSON the value of “entry” key is an array of objects which correspond to rows in the spreadsheet. In each such object the column names are the keys and the cells are values. Each column key starts with “gsx$”.

The Aeson library is used to convert raw json bytestring to Aeson Value data, which makes it convenient to traverse and fetch specific values we need.

The ‘parseSheetJson’ method in SheetDB is used to parse and make list of ‘Row’s.

-- Converts the Aeson Value of the sheet to list of rows.

-- Parses the json tree and iterates through it to produce the list.

parseSheetJson :: A.Value -> IO (Maybe [Row])

parseSheetJson json = return $ parseMaybe parseSheet json

**where** parseSheet = withObject "value" $ \obj -> **do**

feed <- obj .: "feed"

entry <- feed .: "entry"

columns <- getColumnHelper entry

**let** parseRows = withObject "object" $ \obj2 ->**do**

**let** makePairs :: T.Text -> Parser Cell

makePairs key = **do**

valObject <- obj2 .: T.append (T.pack "gsx$") key

val <- (valObject .: "$t") :: Parser A.Value

**let** cell = **case** val **of**

AT.String x -> key =: x

AT.Bool b -> key =: b

AT.Number n -> key =: n

**\_** -> key =: (Nothing :: Maybe ST.Value)

return cell

currRow <- mapM makePairs columns

linkId <- obj2 .: "id"

link <- (linkId .: "$t" ) :: Parser A.Value

**let** rowid = **case** link **of**

AT.String u -> idKey =: u

**\_** -> idKey =: (Nothing :: Maybe ST.Value)

return $ rowid : currRow

**let** parseEntry = withArray "array" $ \arr -> return $ map parseRows (V.toList arr)

rows <-parseEntry entry

sequence rows

All the above process is done part of find method.

find :: Query -> IO (Either String [Row])

find q = **do**

**let** sel = selection q

**if** validate q

**then** **do**

**let** spreadsheet = sheet sel

mayberows <- runMaybeT $ **do**

url <- MaybeT $ return $ makeQueryUrl q

jsonValueForm <-MaybeT $ getJson url (oauth spreadsheet)

MaybeT $ parseSheetJson jsonValueForm

**case** mayberows **of**

Just rs ->**do**

**let** rows

| skip q == 0 && limit q == 0 = rs

| skip q > 0 && limit q == 0 = drop (skip q) rs

| skip q == 0 && limit q > 0 = take (limit q) rs

| otherwise = take (limit q) $ drop (skip q) rs

return $ Right rows

Nothing -> return $ Left "Parse error or sheet data error."

**else**

return $ Left "Query skip or Query limit may be invalid."

**Insert Method**

insert method in SheetDB is used to insert a new row into the spreadsheet.

The steps in insert operation are:

**Constructing XML from Row**

End

Returns Success or Failure

Perform HTTP POST with post from GoogleRequest using list feed URL

Construct Row XML

Start

Google sheets API server specifies the XML format of the row to be inserted and was mentioned earlier about the format.

insert method takes Row data to be inserted and constructs XML in the format accepted by Google Sheets API. Along with construction insert method validates data that is being inserted by validating if the column names are valid and also checks if all the columns are present.

[xml](https://hackage.haskell.org/package/xml) library is used to construct the XML.

-- Checks if Row has all the Columns of the corresponding spreadsheet

isvalid :: Row -> [ColName] -> Bool

isvalid [] **\_** = True

isvalid ((key := **\_**):xs) cols = elem key cols && isvalid xs cols

isEqualLength :: [a] -> [b] -> Bool

isEqualLength x y = length x == length y

xmlns = Attr{

attrKey= unqual "xmlns",

attrVal="http://www.w3.org/2005/Atom"

}

xmlnsgsx = Attr{

attrKey = unqual "xmlns:gsx",

attrVal = "http://schemas.google.com/spreadsheets/2006/extended"

}

cellToXml :: Cell -> Element

cellToXml (key := v) = **do**

**let** valstring = **case** v **of**

ST.String x -> T.unpack x

**\_** -> show v

unode ("gsx:" ++ T.unpack key) valstring

-- Create xml string from list of key value tuples used for insert method

rowToXml :: Row -> String

rowToXml row = showElement $ add\_attrs [xmlns, xmlnsgsx] $ unode "entry" $ map cellToXml row

rowToXml method converts a Row to XML which looks like:

<entry xmlns="http://www.w3.org/2005/Atom"  
    xmlns:gsx="http://schemas.google.com/spreadsheets/2006/extended">  
  <gsx:hours>1</gsx:hours>  
  <gsx:ipm>1</gsx:ipm>  
  <gsx:items>60</gsx:items>  
  <gsx:name>Elizabeth Bennet</gsx:name>  
</entry>

After constructing the XML, a HTTP POST request is performed using the post method in GoogleRequest to get an Outcome.

All the above process is performed in insert method.

-- Adds a new row to google sheet after validation of data to be inserted

insert :: Row -> Sheet -> IO Outcome

insert rawRow sheet =**do**

**let** cols = columns sheet

**if** not (isEqualLength rawRow cols && isvalid rawRow cols)

**then**

-- print cols

-- print rawRow

return SheetDB.Failure

**else** **do**

**let** xml = rowToXml rawRow

-- print xml

**let** maybeurl = formURL ( key sheet) (worksheetId sheet) []

**case** maybeurl **of**

Just url ->**do**

status <- post (exportURL url) (oauth sheet) xml

**if** status == status201

**then** return Success

**else** return Failure

Nothing -> return Failure

**Update Method**

update method in SheetDB is used to update an existing row with new data.

The steps in update operation are

Start

**Parse Edit URL**  
Each Row has an edit URL which must be used to perform a PUT request with new Row. The edit URL changes after each update so there is a need to fetch the Row and parse edit URL before an update. In the JSON example above the row object in the entry contains “id” field which is a URL, uniquely identifies a row in spreadsheet. The row id URL can be used to perform a GET request to get the JSON corresponding to the row. The example of JSON corresponding to a row was also mentioned in section “Google Spreadsheet URL construction”. After JSON is fetched the edit URL is parsed from it using this method.

End

Return Success or Failure

Perform PUT request on the edit URL using put from GoogleRequest

Construct XML of Row along with id to update

Parse Edit URL of the Row from the JSON

Perform GET request on id URL of Row to be updated

parseEditURL :: A.Value -> IO (Maybe String)

parseEditURL json =

**case** parseMaybe parseRow json **of**

Just (AT.String url) ->return $ Just $ T.unpack url

**\_** ->return Nothing

**where** parseRow = withObject "value" $ \obj -> **do**

entry <- obj .: "entry"

link <- entry .: "link" :: Parser A.Value

**let** isEdit = withObject "object" $ \ rowobj ->**do**

rel <- rowobj .: "rel" :: Parser A.Value

return $ rel == AT.String "edit"

**let** parseURLSObjs = withArray "array" $ \arr -> filterM isEdit (V.toList arr)

editurlobjlist <- parseURLSObjs link

**let** editurlobj = head editurlobjlist

**let** extract = withObject "object" $ \ obj-> obj .: "href" :: Parser A.Value

extract editurlobj

update method takes Row as argument and this Row data must contain “id” field that helps to know which corresponding row needs to be updated in spreadsheet. If the “id” field is not present the operation fails.

The Row data is converted to xml in the same way as insert method mentioned previously. After the XML is constructed a HTTP PUT request is performed using the GoogleRequest’s put method to return Outcome data.

All the above process is done as part of update method.

update :: Sheet -> Row -> IO Outcome

update sheet toRow = **do**

puturl <- runMaybeT $ **do**

jsonValueForm <- MaybeT $ getJson (T.unpack (at idKey toRow) ++ "?alt=json") (oauth sheet)

MaybeT $ parseEditURL jsonValueForm

**case** puturl **of**

Just purl -> **do**

**let** cols = idKey : columns sheet

**if** not (isEqualLength toRow cols && isvalid toRow cols)

**then** **do**

return Failure

**else** **do**

**let** xml = rowToXml toRow

-- print xml

status <- put purl (oauth sheet) xml

**if** status == status200

**then** return Success

**else** return Failure

**\_** -> return Failure

**Remove Method**

remove method in SheetDB is used to remove an existing row from spreadsheet.

The steps in remove operation are:

Start

remove method utilizes the delete method in GoogleRequest and returns Outcome as a result.

End

Return Success or Failure

Perform DELETE request on id URL of the row to be deleted

And finally, the actual remove method.

-- Used to send a delete request to the Google Sheet.

remove :: Sheet -> Row -> IO Outcome

remove sheet r = **do**

**let** url = at idKey r

-- print $ T.unpack url

status<-delete (T.unpack url) (oauth sheet)

**if** status == status200 **then** return Success

**else** return Failure