IRIS: Asistente virtual para la redacción personalizada de correos electrónicos
IRIS: Virtual Assistant for Personalized Email Writing

Trabajo de Fin de Grado
Curso 2019–2020

Autor
Carlos Moreno Morera

Director
Raquel Hervás Ballesteros
Gonzalo Méndez Pozo

Doble Grado en Ingeniería Informática y Matemáticas
Facultad de Informática
Universidad Complutense de Madrid

IRIS: Asistente virtual para la redacción personalizada de correos electrónicos
IRIS: Virtual Assistant for Personalized Email Writing

Trabajo de Fin de Grado en Ingeniería Informática
Departamento de Ingeniería del Software e Inteligencia Artificial

Autor
Carlos Moreno Morera

Director
Raquel Hervás Ballesteros
Gonzalo Méndez Pozo

Doble Grado en Ingeniería Informática y Matemáticas
Facultad de Informática
Universidad Complutense de Madrid

1 de mayo de 2020

A Pedro Pablo y Marco Antonio, por crear TeXiS
e iluminar nuestro camino

Acknowledgments

A Guillermo, por el tiempo empleado en hacer estas plantillas. A Adrián, Enrique y Nacho, por sus comentarios para mejorar lo que hicimos. Y a Narciso, a quien no le ha hecho falta el Anillo Único para coordinarnos a todos.

Resumen

IRIS: Asistente virtual para la redacción personalizada de correos electrónicos

Un resumen en castellano de media página, incluyendo el título en castellano. A continuación, se escribirá una lista de no más de 10 palabras clave.

Palabras clave

Máximo 10 palabras clave separadas por comas

Abstract

IRIS: Virtual Assistant for Personalized Email Writing

An abstract in English, half a page long, including the title in English. Below, a list with no more than 10 keywords.

Keywords

10 keywords max., separated by commas.

Contents

List of figures

List of tables

Chapter 1

Introduction

"Have you ever retired a human by mistake?"
— Rachael - Blade Runner (1982)

Smartphone development meant not only a technological advance but a social revolution too. This intelligent telephones have brought with them countless paradigm shifts in terms of the social sphere. Since then, we are able to speak of a new model of human relationship both between people and with our technology. This current relation standard is due to the easy and quick way of accessing the different information that our mobile devices provide us. Long waits (nowadays the meaning of "long" waits has changed too, people consider more than two or three second too much time) for obtaining anything such as accessing to a website or showing any operation result, are excessively tedious and could be even frustrating for some smartphone users. When we are using our mobile, we want, as fast as possible, the information we are looking for. Precisely because of this, Human-Computer Interaction (HCI) becomes a very important part in the process of development of most applications, not only in terms of speed of response and efficiency of algorithms, but also in how we show different information and the easiness for obtaining it.

As for the relationships between people, as we have said, they have dramatically changed. There is no doubt that the main driving technologies behind this transformation of our relational paradigm are the social networks and the instant messaging. Focusing on the latter, it is necessary to make a breakdown of what consequences to our interpersonal interaction the instant communication have brought with itself. Just as it happens with the HCI, easiness and speed are probably the first features we look for when we are going to send or receive any information to anybody. If we also expect a reply, the ideal would be to obtain it as quickly as possible. Therefore, in most of occasions, in practice we are looking for an "automatic" response from a human, what practically implies that everyone is "obligated" to be connected at any time with the answer we are asking for prepared. This new insight into the relationships between people, that perceives the humans as servers who send a request waiting for a quickly reply with the expected data, has promoted a very fast sending of short messages which intends to substitute and simulate an spoken conversation. These little texts are often concise and summarised, and they form an atomic semantic unit, namely they have their own independent meaning.

1.1. Incentive

Introducción al tema del TFM.

1.2. Objectives

Descripción de los objetivos del trabajo.

1.3. Working plan

Aquí se describe el plan de trabajo a seguir para la consecución de los objetivos descritos en el apartado anterior.

1.4. Explicaciones adicionales sobre el uso de esta plantilla

Si quieres cambiar el **estilo del título** de los capítulos, edita `TeXiS\TeXiS_pream.tex` y comenta la línea `\usepackage[Lenny]{fncychap}` para dejar el estilo básico de LaTeX.

Si no te gusta que no haya **espacios entre párrafos** y quieres dejar un pequeño espacio en blanco, no metas saltos de línea (\\) al final de los párrafos. En su lugar, busca el comando `\setlength{\parskip}{0.2ex}` en `TeXiS\TeXiS_pream.tex` y aumenta el valor de $0.2ex$ a, por ejemplo, $1ex$.

TFMTeXiS se ha elaborado a partir de la plantilla de TeXiS[1], creada por Marco Antonio y Pedro Pablo Gómez Martín para escribir su tesis doctoral. Para explicaciones más extensas y detalladas sobre cómo usar esta plantilla, recomendamos la lectura del documento `TeXiS-Manual-1.0.pdf` que acompaña a esta plantilla.

El siguiente texto se genera con el comando `\lipsum[2-20]` que viene a continuación en el fichero .tex. El único propósito es mostrar el aspecto de las páginas usando esta plantilla. Quita este comando y, si quieres, comenta o elimina el paquete *lipsum* al final de `TeXiS\TeXiS_pream.tex`

1.4.1. Texto de prueba

[1] `http://gaia.fdi.ucm.es/research/texis/`

Chapter 2

State of the Art

2.1. Gmail API

In order to be able to read and send emails, it is necessary to access to the user's email data. For this reason, the different ways to obtain this information were studied. One of them is the Gmail API, which allows developers to perform all the actions we need in an easy way.

Gmail API can be used in several programming languages such as Python, PHP, Go, Java, .NET, ... Due to the greater number of examples in the starting guides of the Gmail API (Google, 2019a) and the previous knowledge that was already had of it, Python was chosen for the first contact with this technology.

The following tries to be a step-by-step explanation of what is necessary to know to access the user's Gmail account, create a message, send an email previously created, create and update a draft, reply a received message (for this it is necessary to know how to create an email) and read important information of message threads and individual emails (such as who is the sender, who has received the message, the subject, the date, the email's body, the attached files, ...). Hence, in Section 2.1.1, we are going to study the necessary protocol for accessing the Gmail API and consequently for being able to get into the user's email data. Further on, we will require a resource (like a programming object) we can work with and how to obtain it is explained in Section 2.1.2. Once we count on this general resource, we have the necessary tools to be able to understand and handle the internal architecture of the Gmail API and the different means it provides in order to achieve our goal. Therefore, in Sections 2.1.4, 2.1.5, 2.1.6 and 2.1.7 we are going to delve into the essential resources for our purpose: labels, messages, threads and drafts, respectively. Both the representations and the methods of this resources are studied to achieve this aim.

Finally, as this API is not the only means of accessing the user's mail data (we will study other ways in later sections), we will end with a brief description about the API usage limits (in Section 2.1.8) to assess its use with respect to other methods of email access.

2.1.1. OAuth 2.0 Protocol

Gmail API, as it also happens in the case of other Google APIs, uses OAuth 2.0 protocol (Google, 2019f) to handle authentication and authorization. As it will be seen later in this section, to be in possession of OAuth 2.0 client credentials from the Google API Console is required for having the appropriate permissions to use the Gmail API.

The Google API Console, also known as Google Console Developer[1], built into Google Cloud Platform, makes possible an authorized access to a user's Gmail data. In order to achieve it, having a Google account is a prerequisite because accessing to this platform will be necessary. Once this web has been accessed, at first we have to create a new development project by clicking in "New Project" in the control panel (which is the main tab of the Google Console Developer and the one that opens by default when you access it). When we have already created a project, we will enable the API we are going to work with, in this case the Gmail API. To do this we will look for it in the search engine that we can find in the library of APIs of this platform. Now we can apply for the credentials we need. Accessing to the "Credentials" tab and clicking on "Create Credentials" will lead us to an easy questionnaire, about what type of credentials we prefer, that we have to answer by basing on what type of application we are building. Then we must download the .json file and save it in the folder we are going to work in.

Before starting the development of the implementation of the OAuth 2.0 protocol which will provides us a secure and trusted login system to access to the user's Gmail data, we must install the Google Client Library[2] of our choice of language (we will use Python, so we have to install the libraries *google-api-python-client*, *google-auth-httplib2* and *google-auth-oauthlib*).

There are many ways to obtain the necessary permissions for accessing to the user's emails data following the OAuth 2.0 protocol. As this is a first contact with the Gmail API only with the intention of knowing the possibilities it offers to us and its advantages and disadvantages for future implementations, we are going to develop a simple script which is using a class very useful for local development and applications that are installed on a desktop operating system. The class *InstalledAppFlow*, in *google_auth_oauthlib.flow* (Google, 2019b), is a *Flow* subclass (which belongs to the same library). Thanks to this last class we have mentioned, *InstalledAppFlow* uses a *requests_oauthlib.OAuth2Session* instance at *oauth2session* to perform all of the OAuth 2.0 logic. Besides it also inherits from *Flow* the class method *from_client_secrets_file* which creates a *Flow* instance from a Google client secrets file (this file will be the .json file that we obtained through the Google API Console) and a list of OAuth 2.0 Scopes (Google, 2019e), which are a mechanism in OAuth 2.0 to limit an application's access to a user's account. An application can request one or more scopes. This information is then presented to the user in the consent screen, and the access token issued to the application will be limited to the scopes granted (we will use the Gmail API OAuth 2.0 Scope which allows us to read, compose, send, and delete emails).

After constructing an *InstalledAppFlow* by calling *from_client_secrets_file* as we have explained, we can invoke the class method *run_local_server* which instructs the user to open the authorization URL in the browser and will try to automatically open it. This function will start a local web server to listen for the authorization response. Once there is a reply, the authorization server will redirect the user's browser to the local web server. As we can see in Figure 2.1, the web server will get the authorization code from the response and shutdown, that code is then exchanged for a token.

Then, we will be in possession of the OAuth 2.0 credentials for the user (Google, 2019d) which we are going to use for accessing the user's Gmail account. In summary, it is possible to obtain the necessary permissions from the user and to follow the OAuth 2.0 protocol, by executing these instructions (written in Python):

```
from google_auth_oauthlib.flow import InstalledAppFlow
```

[1] https://console.developers.google.com/

[2] https://developers.google.com/gmail/api/downloads

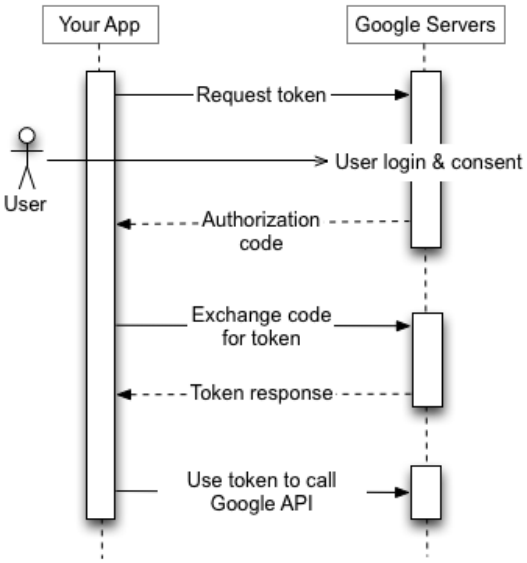

Figure 2.1: OAuth 2.0 for Web Server Applications and Installed Applications. Image extracted from Google (2019f)

```
# Create a flow instance
flow = InstalledAppFlow.from_client_secrets_file('credentials.json',
    ['https://mail.google.com/'])
# Obtain OAuth 2.0 credentials for the user
creds = flow.run_local_server(port = 0)
```

Now, we are able to call Gmail API by using the token (which is stored in the variable *creds*). However, before starting working on the email data, we should save the OAuth 2.0 credentials since otherwise the user would need to go through the consent screen every time the application is opened. To prevent the latter from happening, to differentiate access from mail management and consequently to reuse as much code as possible, we have implemented the following class *auth*, in *auth.py*, with a main method *get_credentials*:

```
import pickle
import os.path
from google_auth_oauthlib.flow import InstalledAppFlow
from google.auth.transport.requests import Request

class auth:
    def __init__(self, SCOPES, CLIENT_SECRET_FILE):
        self.SCOPES = SCOPES
        self.CLIENT_SECRET_FILE = CLIENT_SECRET_FILE

    def get_credentials(self):
        """
        Obtains valid credentials for accessing Gmail API
        """
        creds = None
        # The file token.pickle stores the user's access and refresh tokens
        if os.path.exists('token.pickle'):
            with open('token.pickle', 'rb') as token:
                creds = pickle.load(token)
```

```
20        # If there are no (valid) credentials available, let the user log in
          if not creds or not creds.valid:
              if creds and creds.expired and creds.refresh_token:
                  creds.refresh(Request())
              else:
25                flow = InstalledAppFlow.from_client_secrets_file(
                      self.CLIENT_SECRET_FILE, self.SCOPES)
                  creds = flow.run_local_server(port=0)
              # Create token.pickle and save the credentials for the next run
              with open('token.pickle', 'wb') as token:
30                pickle.dump(creds, token)
          return creds
```

As we can observe in line 17 within *get_credentials* method, at first we check if the file called *token.pickle* exists, and in that case, it is opened and its information is stored in the variable *creds*. Thus, we avoid to force the user to open the authorization screen. By contrast, as we have seen before, if it does not exists, we obtain the credentials by calling the class methods *from_client_secrets_file* and *run_local_server* (it is written between lines 25 and 30).

There is another case that is also reflected in the code above (in lines 23 and 24): the credentials are expired (it is possible to check it by executing *creds.expired*) and they can be refreshed (the OAuth 2.0 refresh token is *creds.refresh_token*) (Google, 2019d). In this situation, we will refresh the access token by invoking the method known as *refresh* and by giving it a *Request* object (Google, 2019c) from *google.auth.transport.requests* as the function parameter which used to make HTTP requests.

2.1.2. Building a Gmail Resource

At this point, with the OAuth 2.0 credentials, we are able to call the Gmail API. For this purpose, it is necessary to construct a resource (Google, 2019a, /v1/reference) for interacting with the API. The *build* method, from *googleapiclient.discovery* library (Gregorio, 2019), create that object. As we will see later, this resource will lead us to manage emails, drafts, threads and everything we will like to do with the user's Gmail data. This is why, using the *auth.py* file explained in Section 2.1.1, we are going to start every user session with the instructions below (or their equivalents in the language we are using):

```
from googleapiclient.discovery import build
import auth

SCOPES = ['https://mail.google.com/']
CLIENT_SECRET_FILE = 'credentials.json'

# Creation of an auth instance
authInst = auth.auth(SCOPES, CLIENT_SECRET_FILE)
# Constructing the resource API object
service = build('gmail', 'v1', credentials = authInst.get_credentials())
```

Henceforth, we will use the *service* variable to relate it with the resource object created by the *build* method.

2.1.3. Users resource

The *build* method could be called for obtaining any resource of any Google API (by giving it the suitable parameters). Our specific created *service*[3] has an important instance method that we are going to invoke for every execution: the *users()* method. It returns what is known as users resource (Google, 2019a, /v1/reference/users).

The users resource has also instance methods, which return other Gmail API resources that we are going to need, such as *drafts()* (see Section 2.1.7), *labels()* (see Section 2.1.4), *messages()* (see Section 2.1.5) and *threads()* (see Section 2.1.6) which return drafts, labels, messages and threads resources respectively. Moreover, it possesses the three methods that we explain hereunder (we must remember that for being able to execute any method that we are going to explain in this and next sections, it is necessary to have the appropriate authorization with at least one of the required scopes that we can look up in its documentation):

- *getProfile(userId)*: it returns an object with a dictionary structure as it follows:

```
{
'threadsTotal' : integer, # Total number of threads in the mailbox
'emailAddress' : string, # User's email address
'historyId' : string, # ID of the mailbox's current history record
'messagesTotal' : integer # Total number of messages in the mailbox
}
```

  The parameter is a string with the user's email address. If we remember the authentication process, at no time we ask the user about the email address because we decided to let the Google API functions to handle all that procedure. Therefore we have no way to know this information. Nevertheless, the special string value *'me'* can be used to indicate the authenticated user. For knowing the required scopes for invoking this function look up in (Google, 2019a, /v1/reference/users/getProfile).

- *stop(userId)*: stop receiving push notifications for the given user mailbox. As it happens with *getProfile*, the parameter is a string with the user's email address, but it is possible to use the especial string value *'me'*.

- *whatch(userId, body)*: set up or update a push notification watch on the given user mailbox.

As we are going to call only the *getProfile* method, we have described on details this first function and we have just given an idea about what the rest of them do. Now, in next sections, we are going to explain all the resources we can create with the user resource.

2.1.4. Labels resource

As we have seen in the explanation of the users resource (Section 2.1.3), we can obtain the labels resource (Google, 2019a, /v1/reference/users/labels) by invoking *labels()* instance method of our users resource, that is to say, by using our *service* variable, the instruction *service.users().labels()* will return the label resource.

Thanks to labels we are able to categorize messages and threads within the user's mailbox. They have a dictionary structure and their representation is what we can observe hereunder:

[3] `http://googleapis.github.io/google-api-python-client/docs/dyn/gmail_v1.html`

```
{
'id' : string, # The immutable identifier of the label
'name' : string, # The display name
# The visibility of messages in the Gmail web interface
'messageListVisibility' : string,
'labelListVisibility' : string, # The visibility of label
'type' : string, # The owner type of the label ('system' or 'user')
'messagesTotal' : integer, # Total number of messages with the label
'messagesUnread' : integer, # Number of unread messages with the label
'threadsTotal' : integer, # Total number of threads with the label
'threadsUnread' : integer, # Number of unread threads with the label
'color' : {
    'textColor' : string, # Text color of the label, represented as hex string
    'backgroundColor' : string # Background color represented as hex string #RRGGBB
    }
}
```

The important fields we are going to need are the *name* and the *type*. Labels with *system* type, such as *INBOX*, *SENT*, *DRAFTS* and *UNREAD*, are internally created and cannot be added, modified or deleted.

In order to obtain a label object, we will use the methods of this resource: create, delete, get, list, patch and update. In this manner, for example, we can store a label object by calling the next instructions:

```
labels = service.users().labels()
labelList = labels.list(userId = 'me').execute()
label = labels.get(id = labelList[0]['id'], userId = 'me')
```

It is necessary to use the *get* method because, as we can look up in (Google, 2019a, /v1/reference/users/labels/list), the *list* method only contains an *id*, *name*, *messageListVisibility*, *labelListVisibility* and *type* of each label, whereas the *get* method returns the label resource with all the information.

2.1.5. Messages resource

In most of the operations we are going to execute the correct management of messages will be essential. Therefore, knowing how the emails are represented in Gmail API and how to use them is imperative to understand how to work with this API. For this reason, in this section we are going to delve into the messages resource of the Gmail API, its structure and its methods. As we saw in Section 2.1.3, we can access to this resource by invoking the *messages()* method when we have a users resource.

2.1.5.1. Resource representation

Regardless of which programming language is used, a messages resource (Google, 2019a, /v1/reference/users/messages) internally has a dictionary structure as we can see down below:

```
{
'id' : string,
'threadId' : string,
'labelIds' : [ string ],
'snippet' : string,
'historyId' : unsigned long,
'internalDate' : long,
```

```
'payload' : {
    'partId' : string,
    'mimeType' : string,
    'filename' : string,
    'headers' : [
        {
        'name' : string,
        'value' : string
        }
    ],
    'body' : {
        'attachmentId' : string,
        'size' : integer,
        'data' : bytes
    },
    'parts' : [ (MessagePart) ]
    },
'sizeEstimate' : integer,
'raw' : bytes
}
```

The more important keys of this data structure for this work are:

- *id*: an immutable string which identifies the message.
- *threadId*: we will explain the thread resource in Section 2.1.6 and we will see that a thread is composed of different messages that share common characteristics. The value of this field is a string which represent the identifier of the thread the message belongs to.
- *labelIds*: a list of the identifiers of labels (see Section 2.1.4) applied to the message.
- *payload*: as we can see in the resource representation above, it has a dictionary data structure. The *payload* field is the parsed email structure in the message parts. The more important keys of the *payload* field are:
    - *mimeType*: the MIME type (see the explanation of *Content-Type* header in Section 2.1.5.3) of the message part.
    - *headers*: a list of headers. It contains the standard RFC 2822 (Resnick, 2001) email headers such as *To*, *From*, *Subject* and *Date*. Each header has a *name* field, which is the name of the header (for example *From*), and a *value* field, which is the value of the header (following the same example as with the *name* field: *example@gmail.com* could be the value).
    - *parts*: a list which contains the different MIME message child parts (we will delve into this field in Section 2.1.5.3).
    - *body*: a dictionary structure which contains the body data of this part (see Section 2.1.5.3) in case it does not contain MIME message parts (otherwise it will be empty). This structure should not be confused with an attached file. Each MIME part contains a body property regardless of MIME type of the part.
- *raw*: the entire email message in an RFC 2822 (Resnick, 2001) formatted and base64url (see Section 2.1.5.4) encoded string.

2.1.5.2. Methods

As any other resource, the messages resource has different methods, many of whom we are going to need in the work. We will limit ourselves to describing the methods we may need to use:

- *attachments()*: returns the attachments resource (for more information about this resource, that we will not explain in detail, refer to (Google, 2019a, /v1/reference/users/messages/attachm
- *get(userId, id, format = 'full', metadataHeaders = None)*: if successful, this method returns the requested messages resource. Its parameters are:
  - *id*: the identifier string of the message we are looking for.
  - *userId*: the user's email address. As it happens with the *getProfile* method of the users resource (see Section 2.1.3), the special string value *'me'* can be used to indicate the authenticated user.
  - *format* (optional parameter): the format in which we want the message returned. This field can take the following punctual values: *'full'* (returns the entirely email data with body content parsed in the *payload* messages resource field and the *raw* field is empty), *'metadata'* (returns only an email message with its identifier, email headers and labels), *'minimal'* (returns only an email message with its identifier and labels) and *'raw'* (returns the entirely email message data with the body content in the *raw* messages resource field as a base64url (see Section 2.1.5.4) encoded string and the *payload* field is empty).
  - *metadataHeaders* (optional parameter): it is only used when the format parameter takes the punctual value of *'metadata'*. It is a string list where we have to insert the headers we want to be included.

  For knowing the required scopes for invoking this function refer to (Google, 2019a, /v1/reference/users/messages/get).
- *list(userId, includeSpamTrash = false, labelIds = None, maxResults = None, pageToken = None, q = None)*: returns a resource with the following structure:

```
{
'messages' : [ users.messages resource ],
'nextPageToken' : string,
'resultSizeEstimate' : unsigned integer
}
```

  As it happens with the *list* method of the labels resource (see Section 2.1.4), *'messages'* list does not contain all of a message information (for obtaining the full email data we can use *get* method). Each element of this list only contains the *id* and *threadId* field.

  The parameters of this method are:

  - *userId*: user's email address (we can use the special string value *'me'*).
  - *includeSpamTrash* (optional parameter): boolean parameter which determines if it includes messages with the labels *SPAM* and *TRASH* in the result of the operation.

- *labelIds* (optional parameter): it is a list which let us filter the messages by only returning emails with labels that match all of the identifiers that belong to this list.
- *maxResults* (optional parameter): an integer which determines the maximum number of messages to return.
- *pageToken* (optional parameter): string which specifies a page of results.
- *q* (optional parameter): string which let us do an specific query (with the same query format as the Gmail search box) and filter the messages by only returning emails that match with it.

For knowing the required scopes for invoking this function refer to (Google, 2019a, /v1/reference/users/messages/list).

- *send(userId, body = None, media_body = None, media_mime_type = None)*: it sends the given message to the email addresses specified in the *To*, *Cc* and *Bcc* headers. The first two parameters are the only ones we will use. The first (*userId*) is the user's email address (we can use the special string value *'me'*) and the second is the message we want to send in an RFC 2822 (Resnick, 2001) formatted. For knowing the required scopes for invoking this function refer to (Google, 2019a, /v1/reference/users/messages/send).

2.1.5.3. MIME

To be able to create messages and read the body of the emails, it is essential to understand what the MIME standard consists of. Hence, in this section we are going to give a general idea about this.

MIME, whose acronym stands for Multipurpose Internet Mail Extensions (contributors, 2019b; Freed and Borenstein, 1996), is an Internet standard for the exchange of several file types (text, audio, video, etc.) which provides support to text with characters other than ASCII, non-text attachments, body messages with numerous parts (known as multi-part messages) and headers information with characters other than ASCII. Each data type has a different name in MIME. These names follow the format: type/subtype (both type and subtype are strings), in such a way that the first denotes the general data category and the second the specific type of that information. The values the type can take are:

- *text*: means that the content is simple text. Subtypes like *html*, *xml* and *plain* can follow this type.
- *multipart*: indicates that the message has numerous parts with independent data. Subtypes like *form-data* and *digest* can follow this type.
- *message*: it is used to encapsulate an existing message, for example when we want to reply a email and add the previous message. Subtypes like *partial* and *rfc822* can follow this type.
- *image*: means that the content is an image. Subtypes like *png*, *jpeg* and *gif* can follow this type.
- *audio*: indicates that the content is an audio. Subtypes like *mp3* and *32kadpcm* can follow this type.

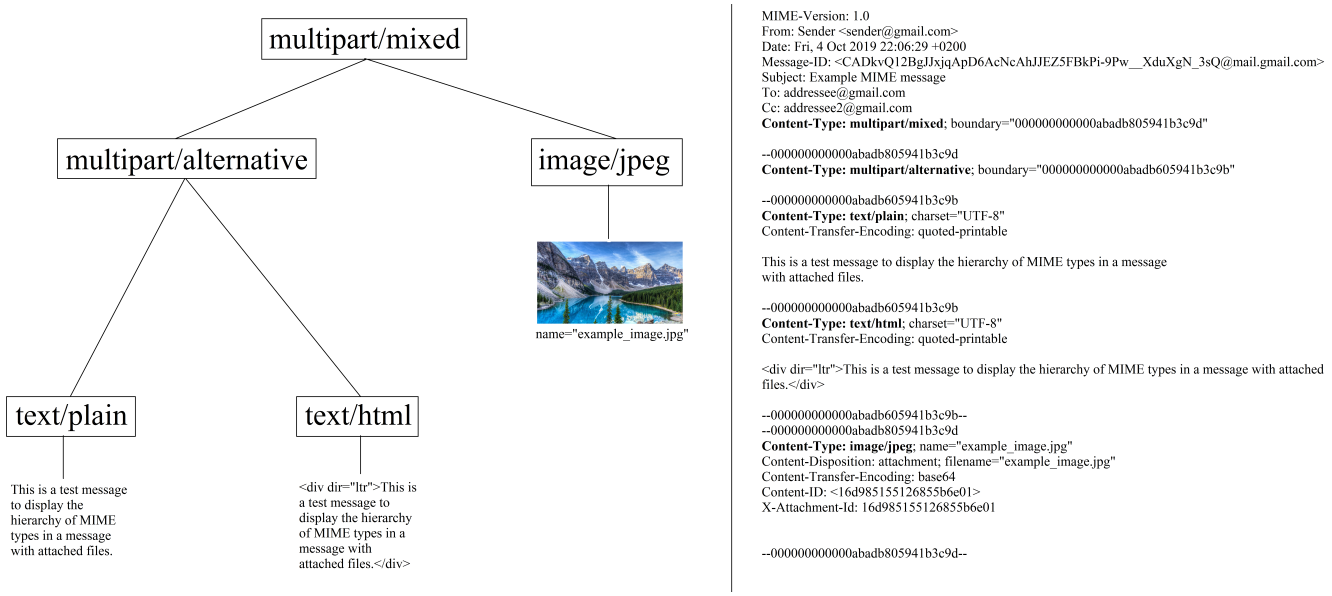

Figure 2.2: MIME types tree structure of an email example

- *video*: denotes that the content is an video. Subtypes like *mpeg* and *avi* can follow this type.
- *application*: it is used for application data that could be binary. Subtypes like *json* and *pdf* can follow this type.
- *font*: means that the content is a file which defines a font format. Subtypes like *woff* and *ttf* can follow this type.

MIME has several headers which appear in all emails sent with this standard. The most important of them are the following:

- *Content-Type*: the value of this header is the type and subtype of the message with the same structure that we have explained before. For example, if we have the header *Content-Type: text/plain*, it means that the message is a plain text. By using the type *multipart* make the creation of messages with parts and subparts organized in a tree structure (in which leaf nodes can belong to any type and the rest of them can belong to any multipart subtype variety) possible (Freed and Borenstein, 1992, Section 7.2). A feasible composition of a message with a part with plain text and other non-text parts could be constructed by using *multipart/mixed* as the root node like in Figure 2.2. Indeed, in the example of Figure 2.2, we can observe the use of *multipart/alternative* for a message which contains the body both in plain text and in html text. Other different emails constructions are possible (like forwarding with the original message attached by using *multipart/mixed* with a *text/plain* part and a *message/rfc822* part) thanks to the tree structure of the *Content-type* header.

  Another important detail, that we can observe in the example in figure 2.2, is the fact that each node of the tree structure of the emails is visited and showed following the pre-order traversal.

- *Content-Disposition*: this header is used to indicate the presentation style of the part of the message. There are two ways to show the part: *inline* content-disposition

(which means that the content must be displayed at the same time as the message) and *attachment* content-disposition (the part is not displayed at the same time as the message and it requires some form or action from the user to see it). Furthermore, this header also provides several fields for specifying other type of information about the content, such as the name of the file and the creation or modification date. The following example is taken from RFC 2183 (Troost et al., 1997) and, as we will explain after the example, it does not match with the syntax of this same header in the example that we can see in the last part of the example message of figure 2.2:

```
Content-Disposition: attachment; filename=genome.jpeg;
    modification-date="Wed, 12 Feb 1997 16:29:51 -0500";
```

As we have said, this syntax is different from the used in the email example of Figure 2.2. This results from the fact that, in HTTP, the header we find in that figure (*Content-Disposition: attachment*) is usually used for instructing the client to show the response body as a downloadable file. As we can observe, it has a *filename* field which is used for establishing the default file name when the user is going to download it.

- *Content-Transfer-Encoding*: when we want to send some files in a message, sometimes they are represented as 8-bit character or binary data, which are not allowed in some protocols. On this account, it is necessary to have a standard that indicates how we should re-encoding such data into a 7-bit short-line format. The Content-Transfer-Encoding header (Freed and Borenstein, 1992, Section 5) will tell the client which transformation has been used for being able to transport that data. Therefore and for lack of a previous standard which states a single Content-Transfer-Encoding mechanism, the possible values, which specify the type of encoding are: *'base64'* (see Section 2.1.5.4), *'quoted-printable'* (see Section 2.1.5.5), *'8bit'*, *'7bit'*, *'binary'* and *'x-token'*. All these values are not case sensitive. If this header does not exist, we can assume that the value of this header is *'7bit'*, which means that the body of the message is already in a seven-bit mail-ready representation, in other words, all the body of the message is represented as short lines of US-ASCII data. Despite the *'8bit'*, *'7bit'* and *'binary'* indicate that the content has not been transformed, they are useful for knowing the kind of encoding that the data has. This header will generally be omitted when the Content-Type has the *multipart* or *'message'* type (as it happens in the message example of Figure 2.2), because it also admits the last three types we have mentioned.

  It is common to add another header (as we can see in Figure 2.2) called *charset*, which value represents the original encoding of data so the client is able to decode it.

2.1.5.4. Base64

As we have seen, in the field *raw* of the messages resource (see Section 2.1.5.1) the entire email message has to be encoded using base64url; and, as we have studied when we learnt how the MIME standard (see Section 2.1.5.3) is, we can find email whose content encoding is base64. Base64 (contributors, 2019a; Josefsson, 2006) is a group of reversible binary-to-text encoding schemes which represent binary data as a sequence of ASCII printable characters. It makes use of a radix-64 to translate each character, because 64 is the higher power of two than can be represented using only printable ASCII characters. Indeed all the Base64 variants (like base64url) utilise the characters range A-Z, a-z and 0-9 in that

order for the first 62 digits, but the chosen symbol for the last two digits are very different between them. In particular, the MIME (see Section 2.1.5.3) specification, established in RFC 2045 (Freed and Borenstein, 1996), describes base64 based on Privacy-enhanced Electronic Mail (PEM) protocol (contributors, 2019c; Josefsson and Leonard, 2015), which means that the last two characters are '+' and '/' and the symbol '=' is used for output padding suffix. In the same way, MIME does not stablish a fixed size for the base64 encoded lines, by contrast it specifies a maximum size of 76 characters.

If we try to apply standard base64 in a URL encoder, it will translate the characters '+' and '/' to its hexadecimal representation ('+' = '%2B' y '/' = '%2F'). This will cause a conflict in heterogeneous systems or if we use it in data base storage, because of the character '%' produced by the encoder (it is a special symbol of ANSI SQL). This is why modified Base64 for URL variants exists (such as base64url in Josefsson (2006)), where the '=' character has no usefulness and the '+' and '/' symbols are replaced by '-' and '_' respectively. Besides it has no impact on the size of encoded lines.

2.1.5.5. Quoted-printable

Other reversible binary-to-text encoding that could be used in the content of a MIME message is the quoted-printable encoding (contributors, 2019d; Borenstein and Freed, 1993). Making use of printable characters (such as alphanumeric and '=') proved capable of transmitting 8 bit data over a 7 bit protocol. Unlike base64, if the original message is mostly composed of ASCII characters, the encoded text is readable and compact.

Each byte could be represented via two hexadecimal character. On this basis, the '=' symbol followed by two hexadecimal digits are enough to encode all the characters except the printable ASCII ones and the end of line. For example, if we want to represent the 12th ASCII character we can encode it as '=0C' or if the equality symbol (whose decimal value is 61) is in our original message, it could be encoded as '=3D' (note that despite being a printable ascii character it must be encoded as it is a special character in this encoding). This is how quoted-printable encodes the different characters.

In respect of the maximum line size, as it happens with the MIME specification of the base64 (see Section 2.1.5.4), it is 76 characters each encoded line. To achieve this goal and still be able to decode the text getting the original message, quoted-printable adds *soft line breaks* at the end of the line consisting of the '=' symbol and it does not modify the encoded text.

2.1.6. Threads resource

When we access to our inbox, we are actually seeing the inbox threads instead of the messages resource. Every message, even if it is an only email without a reply, is enclosed in a thread resource (Google, 2019a, /v1/reference/users/threads) which is essentially a list, perhaps unitary, of messages resources. In fact, as we can observe in the following resource representation, in its dictionary structure it has a list of messages resources:

```
{
    'id' : string, # The identifier of the thread
    'snippet' : string, # A short part of the text
    'historyId' : unsigned long,
    'messages' : [ users.messages resource ]
}
```

The most important methods of this resource are:

- *get(userId, id, format = 'full', metadataHeaders = None)*: if successful, this method returns the requested threads resource. In respect of the parameters, they are defined in the same way as in *get* messages resource method (see Section 2.1.5.2) with the exception of the parameter *format*, whose only difference is that it does not accept the *'raw'* value. For knowing the required scopes for invoking this function look up in (Google, 2019a, /v1/reference/users/threads/get).

- *list(userId, includeSpamTrash = False, labelIds = None, maxResults = None, pageToken = None, q = None)*: if successful, it returns a dictionary structure analogous to the view in the *list* message resource method (see Section 2.1.5.2). Needless to say, instead of returning a messages resource list it will give us a threads resource list, which does not contain the complete information of each thread (for example each element of the list has not a list of messages resource). Full thread data can be fetched using the previous method. The parameters of this method are defined in the same way as the *list* messages resource method. For knowing the required scopes for invoking this function refer to (Google, 2019a, /v1/reference/users/threads/list).

2.1.7. Drafts resource

The last Gmail API resource we will study is the most easy to understand after knowing all the structures related with emails that we have explained in the above sections: the drafts resource (Google, 2019a, /v1/reference/users/drafts). Its representation is very simple:

```
{
    'id' : string # The immutable identifier of the draft
    'message' : users.messages resource
}
```

As we can observe, a draft is virtually a messages resource with an identifier. Indeed, in order to create a new draft with the *DRAFT* label we must create a MIME message (see Section 2.1.5.3) as we have to do when we want to send a new email by using the *send* messages resource method. Then it is necessary to invoke the drafts resource method *create(userId, body = None, media_body = None, media_mime_type = None)* by giving the value *'me'* and the message created to the first to parameters.

2.1.8. API Usage Limits

One factor to be taken into account is the limitations of the Gmail API (Google, 2019a, /v1/reference/quota) which could become a drawback in the application development. It has a limit on the daily usage and on the per-user rate. In order to measure the usage rate, "quota units" are defined depending on the method invoked. In Table 2.1 we can consult the value of some methods in quota units (we have selected the more important methods for our purpose, for the quota units of other methods it is recommended to refer to (Google, 2019a, /v1/reference/quota)).

However, both daily usage limit and per-user rate limit are acceptable for the type of software we want to build: 1,000,000,000 quota units per day and 250 quota units per user per second. Therefore there are no constraints (for our purpose) that avoid us to use this API.

Method	Section where the method is explained	Quota units
getProfile	2.1.3	1
labels.get	2.1.4	1
messages.get	2.1.5.2	5
messages.list	2.1.5.2	5
messages.send	2.1.5.2	100
threads.get	2.1.6	10
threads.list	2.1.6	10
drafts.create	2.1.7	10

Table 2.1: Main methods' quota units

2.2. Electronic Mail Protocols

Apart from Gmail API (see Section 2.1), there is another way to read, send emails and access to the user's email data and this is by directly using the communication protocol for electronic mail transmission and the internet standard protocol to retrieve email messages from a mail server over a TCP/IP connection. In this section we are going to study the main email management protocols, both electronic mail transmission protocol (such as Simple Mail Transfer Protocol, which is explained in Section 2.2.1) and message access protocol (such as Internet Message Access Protocol and Post Office Protocol, which are studied in Sections 2.2.3 and 2.2.2, respectively).

In spite of being a mail server-independent solution, as we will see, we are going to find security issues which are going to hinder our user's email data access. These trials come from the automatic server access. The assessment of the advantages and disadvantages of making use of the email protocols or the Gmail API is discussed in Section 2.2.4.

2.2.1. Simple Mail Transfer Protocol

Simple Mail Transfer Protocol (also known as SMTP) is a network connection-oriented communication protocol used for the exchange of e-mail messages. It was originally defined in Postel (1982) (for the transfer) and in Crocker (1982) (for the message). It is currently defined in Klensin (2008) and Resnick (2008). However, this protocol has some limitations when it comes to receiving messages on the destination server. For this reason, this task is intended for other protocols such as the Internet Message Access Protocol (see Section 2.2.3) or the Post Office Protocol (refer to Section 2.2.2), and SMTP is used specifically to send messages.

Making use of SMTP, an email is "pushed" from one mail server to another (next-hop mail server) until it reaches its destination. The message is not routed according to the message recipients specified during the client's connection to the SMTP server, but from the destination mail server. Thanks to the fact that this protocol has a feature to initiate mail queue processing, intermittently connected mail server can extract messages from another remote server when necessary.

2.2.2. Post Office Protocol

Post Office Protocol (also known as POP) is an application protocol (in OSI Model) for obtaining e-mails stored in a remote Internet server called POP server. It was originally defined in Reynolds (1984) (it was POP version 1, also known as POP1). Current POP

version (POP3, in general when we talk about POP we refer to this version) is detailed in Myers et al. (1996).

POP was designed for receiving emails. Using POP, users with intermittent or very slow Internet connections (such as modem connections) can download their email while online and check it later even when offline. The general operation is: a client using POP3 connects, gets all messages, stores them on the user's computer as new messages, deletes them from the server, and finally disconnects. However some mail clients include the option to leave messages on the server. They use the order UIDL (Unique IDentification Listing) which, unlike most POP3 commands, does not identify messages depending on their mail server ordinal number, due to the fact that this creates problems when a client tries to leave messages on the server, since messages with numbers change from one connection to the server to another. Accordingly, server which makes use of UIDL, assigns a unique and permanent character string to each message. Thus, when a POP3-compatible mail client connects to the server, it uses the UIDL command to map the message identifier. This way the client can use that mapping to determine which messages to download and which to save at the time of download.

Like other old Internet protocols, POP3 used a signature mechanism without encryption. The transmission of POP3 passwords in plain text still occurs. Nowadays POP3 has various authentication methods that offer a diverse range of levels of protection against illegal access to users' mailboxes.

The advantage over other protocols is that between server-client you do not have to send so many commands for communication between them. The POP protocol also works properly if you do not use a constant connection to the Internet or to the network that contains the mail server.

2.2.3. Internet Message Access Protocol

Internet Message Access Protocol (also known as IMAP) is an application protocol, designed as an alternative to Post Office Protocol (se Section 2.2.2) in 1986, which allows the access to stored messages in an Internet server. As with the Post Office Protocol, with IMAP you can access your e-mail from any computer with an Internet connection. The current version of IMAP (IMAP version 4 review 1 or IMAP4rev1) is defined in Crispin (2003).

In contrast to Post Office Protocol, IMAP allows multiple clients to manage the same mailbox. This fact results from the main differences between these two protocols: IMAP does not remove email from server until the client specifically requests it (as POP removes them by default, it is impossible to accessing them from another device which has not the downloaded messages) and it does not download the messages to the user's computer (clients may optionally store a local copy of them). This last property gives raise to several advantages with regard to Post Office Protocol: the immediate notification of the arrival of a mail (due to it works in permanent connection mode) while POP checks if there are new e-mails every few minutes (which causes an appreciable rise in traffic and in the time the user has to wait to send a request to the server, because it is necessary to complete the download of all new messages first), it is possible to create shared folders with other users (it depends on the mail server), the e-mails do not take up memory in the user's local device while POP downloads them regardless of whether they are going to be read or not (effectively IMAP has to download a message when it is going to be read, but they are temporary files and only the e-mail headers are downloaded to manage the mailbox) and it allows the user to manage folders, templates and drafts in server in addition to be able

to search a mail from keywords.

2.2.4. Advantages and disadvantages of email protocols versus the use of Gmail API

Chapter 3

Work Description

3.1. Style Analyser

In order to generate messages with the user's writing style, it is necessary to define parameters which will determine and describe it. For this purpose, we have developed a style analyser that extracts the messages written by the user and obtains the value of various metrics from them. Then it will be useful for analysing different user's emails and drawing conclusions about what parameters describe the writing style of each person more accurately. Besides most of the developed code will be reusable in the final application for analysing the user's messages.

In this section we are going to explain the architecture of this analyser (see 3.1.1) and each of the modules that compose it (they are explained in sections 3.1.2, 3.1.3, 3.1.4 and 3.1.5). Finally, we are going to discuss the obtained results and analyse them for drawing a conclusion (this discussion can be looked up in 3.1.6).

3.1.1. Architecture

The first step when we are designing a system's architecture is to know its input and output. In this case, we want to implement a simple natural language processing system that analyses the writing style of emails. As we have previously mentioned, the writing style analysis will be represented through chosen metrics. Therefore, our system's output is going to be that chosen metrics (they are explained in section 3.1.5).

En respect of the system's input, because of the nature of the problem we face, it is reasonable to think that it must be a single email. However, we do not have the corpus of emails to analyse. For this reason, our first step will be to extract the emails that will be analysed. Hence, our system's input is going to be the Gmail user for accesing to the information that we are interested in.

3.1.2. Extracting Module

3.1.3. Preprocessing Module

3.1.4. Typographic Correction Module

3.1.5. Measuring module

3.1.6. Results and conclusions

Chapter 4

Conclusiones y Trabajo Futuro

Conclusiones del trabajo y líneas de trabajo futuro.

Antes de la entrega de actas de cada convocatoria, en el plazo que se indica en el calendario de los trabajos de fin de máster, el estudiante entregará en el Campus Virtual la versión final de la memoria en PDF. En la portada de la misma deberán figurar, como se ha señalado anteriormente, la convocatoria y la calificación obtenida. Asimismo, el estudiante también entregará todo el material que tenga concedido en préstamo a lo largo del curso.

Bibliography

And thus I clothe my naked villany
With old odd ends stolen out of holy writ;
And seem a saint, when most I play the devil.

Richard III, Act I Scene 3
William Shakespeare

Borenstein, N. and Freed, N. Mime (multipurpose internet mail extensions) part one: Mechanisms for specifying and describing the format of internet message bodies. Tech. Rep. RFC 1521, Internet Engineering Task Force (IETF), 1993.

contributors, W. Base64 — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Base64`, 2019a. [Online; accessed 17-October-2019].

contributors, W. Mime — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=MIME&oldid=916901691`, 2019b. [Online; accessed 23-September-2019].

contributors, W. Privacy-enhanced mail — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Privacy-Enhanced_Mail`, 2019c. [Online; accessed 23-September-2019].

contributors, W. Quoted-printable — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Quoted-printable`, 2019d. [Online; accessed 23-September-2019].

Crispin, M. Internet message access protocol - version 4rev1. Tech. Rep. RFC 3501, University of Washington, 2003.

Crocker, D. H. Standard for the format of arpa internet text messages. Tech. Rep. RFC 822, Dept. of Electrical Engineering, University of Delaware, 1982.

Freed, N. and Borenstein, N. Mime (multipurpose internet mail extensions). Tech. Rep. RFC 1341, Internet Engineering Task Force (IETF), 1992.

Freed, N. and Borenstein, N. Multipurpose internet mail extensions (mime) part one: Format of internet message bodies. Tech. Rep. RFC 2045, Internet Engineering Task Force (IETF), 1996.

Google. Gmail api | google developers. `https://developers.google.com/gmail/api`, 2019a. [Online; accessed 17-October-2019].

Google. google_auth_oauthlib.flow module. https://google-auth-oauthlib.readthedocs.io/en/latest/reference/google_auth_oauthlib.flow.html, 2019b. [Online; accessed 23-September-2019].

Google. google.auth.transport package. https://google-auth.readthedocs.io/en/stable/reference/google.auth.transport.html#google.auth.transport.Request, 2019c. [Online; accessed 23-September-2019].

Google. google.oauth2.credentials module. https://google-auth.readthedocs.io/en/stable/reference/google.oauth2.credentials.html#google.oauth2.credentials.Credentials, 2019d. [Online; accessed 23-September-2019].

Google. Oauth 2.0 scopes for google apis. https://developers.google.com/identity/protocols/googlescopes, 2019e. [Online; accessed 23-September-2019].

Google. Using oauth 2.0 to access google apis. https://developers.google.com/identity/protocols/OAuth, 2019f. [Online; accessed 23-September-2019].

Gregorio, J. googleapiclient.discovery. https://googleapis.github.io/google-api-python-client/docs/epy/googleapiclient.discovery-module.html#build, 2019. [Online; accessed 23-September-2019].

Josefsson, S. The base16, base32, and base64 data encodings. Tech. Rep. RFC 4648, Internet Engineering Task Force (IETF), 2006.

Josefsson, S. and Leonard, S. Textual encodings of pkix, pkcs, and cms structures. Tech. Rep. RFC 7468, Internet Engineering Task Force (IETF), 2015.

Klensin, J. Simple mail transfer protocol. Tech. Rep. RFC 5321, Internet Engineering Task Force (IETF), 2008.

Myers, J., Mellon, C. and Rose, M. Post office protocol - version 3. Tech. Rep. RFC 1939, Dover Beach Consulting, Inc., 1996.

Postel, J. B. Simple mail transfer protocol. Tech. Rep. RFC 821, Information Sciences Institute, University of Southern California, 1982.

Resnick, P. Internet message format. Tech. Rep. RFC 2822, Internet Engineering Task Force (IETF), 2001.

Resnick, P. Internet message format. Tech. Rep. RFC 5322, Qualcomm Incorporated, 2008.

Reynolds, J. K. Post office protocol. Tech. Rep. RFC 918, Information Sciences Institute, 1984.

Troost, R., Dorner, S. and Moore, K. Communicating presentation information in internet messages: The content-disposition header field. Tech. Rep. RFC 2183, Internet Engineering Task Force (IETF), 1997.

Appendix A

Título del Apéndice A

Contenido del apéndice

Appendix B

Título del Apéndice B

Este texto se puede encontrar en el fichero Cascaras/fin.tex. Si deseas eliminarlo, basta con comentar la línea correspondiente al final del fichero TFMTeXiS.tex.

–¿Qué te parece desto, Sancho? – Dijo Don Quijote –
Bien podrán los encantadores quitarme la ventura,
pero el esfuerzo y el ánimo, será imposible.

Segunda parte del Ingenioso Caballero
Don Quijote de la Mancha
Miguel de Cervantes

–Buena está – dijo Sancho –; fírmela vuestra merced.
–No es menester firmarla – dijo Don Quijote–,
sino solamente poner mi rúbrica.

Primera parte del Ingenioso Caballero
Don Quijote de la Mancha
Miguel de Cervantes