STANDARD REST API FOR EMAIL

Kalana Guniyangoda

(118209x)

Dissertation submitted in partial fulfillment of the requirements for the degree Master of Science

Department of Computer Science & Engineering

University of Moratuwa
Sri Lanka

June 2015

DECLARATION

"I declare that this is my own work and this dissertation does not incorporate without acknowledgement any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief, it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to University of Moratuwa the non-exclusive right to reproduce and distribute my dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).

Signature: Date:

Name: Kalana Guniyangoda

The above candidate has carried out research for the Masters Dissertation under my supervision.

Signature of the supervisor: Date:

Name of the supervisor: Prof. Gihan Dias

ABSTRACT

Email has long been a most popular mode of electronic communication. Initially, email communication was between multi-user hosts using the SMTP protocol, and later on, with the popularity of client-server communication, protocols such as POP, IMAP and Submit were developed for connecting e-mail clients and servers. Today, the most popular method of e-mail access is via a web browser. However, there is still a lack of standard protocol defined for e-mail access via web browsers. All the current web-mail systems use proprietary communication between web interfaces and the backend server. Therefore, each web-mail system can only be accessed with its own web interface and vice versa. Therefore, it is opportune to develop a standard protocol for email servers and browser-based email clients harnessed with HTML5 capabilities to communicate over the HTTP protocol.

Representational State Transfer (REST) is a popular architectural style to implement applications using the HTTP protocol and offers many features such as scalability and loose coupling. This would be beneficial in implementing browser-based email clients and would make it possible to create an open standardised HTTP based protocol similar to SMTP.

In this dissertation, we analyse the major REST and non-REST HTTP-based e-mail protocols and APIs, starting from Paul Prescod's initial proposal, as well as other email protocols such as IMAP, and identify the set of features required of an http-based e-mail protocol. We then define a standard API for this purpose, combing the strong features of current systems and protocols. The REST API introduced in this dissertation provides the needed functionality of an e-mail system, including authentication, sending emails, reading emails and managing emails & attachments. Furthermore, we specify messaging formats, error codes and notification mechanisms for the system. We have also developed a server-side implementation which supports the API.

We have run the e-mail system under three scenarios, and show that it has acceptable functionality and performance.

ACKNOWLEDGEMENTS

I would like to dedicate my sincere thanks to my supervisor Prof. Gihan Dias for his dedicated support for the success of this research. This would not have become a success without your support from the initial stage of the research.

I would like to thank the entire academic and the non-academic staff of the Department of Computer Science and Engineering for their kindness extended to me in every aspect.

Last but not least, I thank my family and all my friends who supported me for the success of this piece of work. Your support was so precious.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

API	Application Programme Interface
HATEOAS	Hypermedia As The Engine Of Application State
HTTP	Hypertext Transfer protocol
HTTPS	HTTP over TLS
IANA	Internet Assigned Numbers Authority
IMAP	Internet Message Access Protocol
JSON	JavaScript Object Notation
MIME	Multipurpose Internet Mail Extensions
POP	Post Office Protocol
REST	Representational State Transfer
SMTP	Simple mail transfer protocol
TLS	Transport Layer Security
XML	EXtensible markup Language

LIST OF APPENDICES

1. INTRODUCTION

1.1. Background

Email is an essential part of modern communication. It was first introduced in 1961 in MIT's CTSS system[1]. It became an integral part of the Internet, and the protocols to govern the email communications were also gradually developed. The major protocols which were developed are Simple Mail Transfer Protocol (SMTP)[2], Post Office Protocol (POP)[3] and Internet Message Access Protocol (IMAP)[4]. These protocols cover most of the needs of email communication by providing a complex array of functionalities. Some other helper protocols were also developed alongside with these protocols and were standardized by the Internet Engineering Task Force. The traditional email systems follow a client-server architecture where, the mails are stored in servers and users access email storage using applications which support email protocols. With the popularization of World Wide Web over the Internet, the email systems were also designed with front-ends as web interfaces. However, the same traditional protocol stack was used as the backend for these web interfaces.

With the popularity of the Internet, and its integration to the business aspect, leads to the introduction of web services and related concepts such as service oriented architecture. A web service provides a set of well-defined functionalities which could be used by software programed as an abstract unit. The REpresentational State Transfer [5] was another architectural style which becomes popular as a way of providing abstract interfaces for software applications to communicate programmatically. This type of architecture led to the popularized the "machine to machine communication" and their benefits such as reusability, scalability and uniformity. Therefore the idea of offering email as a web service was designed and implemented accordingly. It is important to note that, even this type of implementations is based on HTTP protocol; it significantly differs from the functionality of a webmail interface. The ability to programmatically access the email store would allow developers to build mashup web applications, third party productivity applications and many more.

However, the APIs and email systems which were developed were not standardized. The implementations were differed significantly, depending on what vendor desired to achieve through the implementation. For example, while Google[6] and Microsoft[7] produced REST APIs to allow programmers to interact with their email facilities, the API calls, semantics and results vary significantly. And when we consider the APIs offered by products such as Sendinc, the functionalities are offered to serve the purpose of sending emails only with different semantics from either of Google or Microsoft implementations. The variations in the APIs would force developers to build client application which is specific to APIs. Moreover, the data exchanging formats which are used in the APIs differ and do not follow the REST constraints in most of the cases. This would result in tight coupling of client/servers which forces the use of proprietary technologies rather than open standards.

1.2. Objectives

In this dissertation, our goal is to introduce a standardized API which could follow by most of the applications which provides email facilities and to standardize the semantics of it. We have identified the required functionalities for a REST based email system by analysing the existing APIs and email systems. Based on the analysis and best practices, we have designed a REST API for interacting with an existing email store which was previously accessed using IMAP or related traditional Email protocols. We also introduce the JSON semantic to be used in for email communication over HTTP protocol.

2. LITERATURE SURVEY

2.1. Email

Email is the abbreviation of electronic mail, a method of exchanging messages electronically between connected computers. Email now has become an essential part of communication in every aspect. Therefore, it is interesting to study how emails evolve into its modern architecture, the methodologies it uses and the limitations.

2.2. The history and evolution

The first form of email communication wasn't between connected computers. The first form of email communication was host based where users would log into a central shared system and able to leave messages which could be read by another user. The first such system is considered to be MIT's Compatible Time-Sharing System (CTSS) [1] in 1961. Even though the email content did not get transferred from one machine to another, users could log into the system using remote terminal and read their emails conveniently. This system has evolved by providing various features as per user convenience, yet provided the basic model of email communication. Following the CTSS system many other host based email systems were developed such as UNIX [8] mail and Professional Office System (PROFs) by IBM[9].

With the popularity of connected computers via network, emails were able to share among several computers rather than keeping it in a shared system. With the research effort of ARPANET in 1971 [10], emails were being able to send across networks reaching remote sites and other organizations. Since then, emails protocols were formalized and it followed client/ server architectural mode. This architectural style inherently provides robustness and scalability which helped the growth of the Internet. However, even though the standard protocols offered common interface for interoperability, vendor specific functionalities and features lead to the development of variety of solutions based on requirement. The email store would vary from database based implementations to flat file system and endpoints vary from client side software to web based interface. Webmail is another innovative way of providing a web based interface to remote email store via Internet. This model offers easy access to email stores using a browser instead of different email clients. But due to vendor specific implementations, a browser is used to load web interface specific to vendor instead. Moreover, their internal architecture is also differs based on their requirements. For example, while Gmail uses database system to store and retrieve emails, the PHP based webmail access email store using generic protocols. With the popularity of mobile devices as a platform for the Internet usage and messaging[11], more client specific interfaces and software were developed to provide specific features.

While World Wide Web and the Internet rapidly grow, the need for system interoperability increased. With this in mind, a new aspect called web service was developed. Web services are used to connect existing software in such a way that application component can be reused. The paradigm was supported by technologies such as SOAP and REST. Service providers used both SOAP and REST to provide their own web services alongside with their other interfaces. Next section, we would discuss about the protocols and formats which are used for emailing.

2.3. Email Protocols

2.3.1. SMTP

SMTP stands for Simple Mail Transfer Protocol. It was defined in RFC 821[2] and was used as the Internet Standard from 1982 to 2008, which was then modified to consolidate old standards and introduce new additions, by the Standard RFC. SMTP was first developed based on Telnet and FTP protocols with the intention of transferring messages across connected computers. However the protocol has been modified to adapt the increasing demand and functionality requirements. Since the protocol was initially meant for message transferring only, to address issues related to authentication and etc., a new Internet Standard was introduced as 'RFC 6409'[12]. The protocol which used specifically mail submission was identified as 'submission protocol'.

The SMTP protocol has being implemented in many by vendor and open source projects to comply with the ' RFC 5321'[13] and is normally offered through port 25 of a server. The mail submission protocol generally offered through port 587. The basic mechanism of how SMTP protocol used in email system portrayed in figure 2-1.

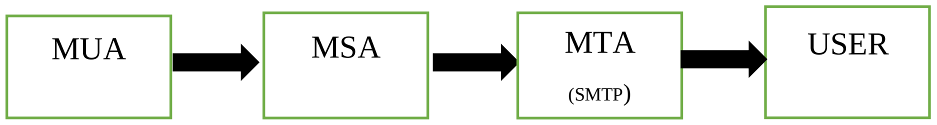


Figure 2-1: Mail Sending Process

First, the MUA (mail user agent) would connect to a MSA (mail submission agent) to submit a properly formatted email message. Then the message would be handed over to MTA (mail transfer agent) for transferring it to the receiver. The receiving MTA would keep the message on the server till the receiver would consume the message. The role of the MSA is not mandatory as many of the MTA's still accept formatted messages for transferring. The email message format is defined in draft standard 'RFC 5322' [14] as "Internet Message Format". The storage of such message is varied from MTA implementations. Generally the MTA's keep their emails in local file systems and some implementations keep it in special database structures. After the mail received by recipient mailbox, mail retrieval protocols such as IMAP and POP would come into play. We will discuss their functionality in the next chapter.

2.3.2. POP and IMAP

POP stands for Post Office Protocol and is used to access previously discussed mailboxes in SMTP servers. The protocol was first defined in 1984 with 'RFC 918' [3] and was later improved to add functionality and security. The current Internet Standard for POP protocol is defined in 'RFC 1939'[15] and is known as POP – version 3. The protocol is generally offered in port 110 and is one of the most common protocols used to access email stores besides IMAP protocol. POP3 is supported by most of the contemporary mail clients as well. Same as SMTP, the POP protocol was designed to listen for connections and act upon commands issued by the connected client.

IMAP stands for Internet Message Access Protocol. It was first defined in 1988 by 'RFC 1064'[4] as ' Interactive Mail Access Protocol'. However the protocol was then developed into the version which used today as ' Internet Message Access Protocol - Version 4' by 'RFC 3501" [16] in 2003. Same as POP, IMAP protocol is also used to retrieve mails from an email store and is generally offered through port 143 and is also a command driven service. Considering the functionality of both protocols, even though both of them are being offered in all modern email clients, IMAP has additional set of functionality and could be considered to supplant POP protocol. On an IMAP server, once logged in, the client could execute commands such as SELECT, CREATE, DELETE, RENAME, SUBSCRIBE, STATUS, LIST using the mailbox name as the parameter. A list of functional differences between the two protocols has been listed below.

1. POP connects to the server as per request and would get disconnected thereafter while IMAP would keep the connection. This would result in faster response time for IMAP protocol.
2. IMAP servers could be accessed by multiple mail clients while POP is only allowed per one client at one time.
3. Partial access to mail parts, such as MIME parts are allowed in IMAP and results in faster message access.
4. IMAP supports storing of state information per mails and would provide greater control to the client.
5. IMAP supports creation, renaming and deletion of mailboxes on the server.
6. IMAP could do server side searches for client requests
7. IMAP supports extensions to the base protocol which adds new functionalities.

2.4. Messaging Formats

2.4.1. Internet Message Format

Internet message format was first formalized as ARPA Internet Text Messages in 1977 in an effort to formalize the informal practices over text messages send across computers. Later this was standardized as “Internet Message Format” by draft standard RFC 5322[14]. Internet message format specifies the syntax for text messages or rather US-ASCII only and does not provide a provision for structured data such as audio or images. Each of character line should not exceed 998 characters to support common implementations. Email message generally consists with two parts; the body and the header. Header section has fields which are used to describe the message while the body part of the message contains the actual message. The header section contains several fields such as date/time and address which are specified by the standard.

2.4.2. Multipurpose Internet Mail Extensions

Apart from the text message exchange, which is specified by Internet message format, the MIME specification is used to encode data such as audio, video, image and other types of data into email as multi-part and non-textual message bodies. This was first formalized by N. Borenstein in 1992 by RFC 1341[17]. MIME is now specified in six linked RFC documents which together define the specification. Even though MIME was first introduced for messages transmitted over SMTP, it is now used as Internet media type. The server which does the MIME encoding adds MIME headers to the message which is then used by the client application to identify the appropriate application to use in decoding the message. IME has specified the following header fields;

- MIME version
- Content-Type: Indicate the Internet media type
- Content-Disposition: specify the presentation style

Since MIME formatted messages support multipart messages, the 'Content-Type' fields could be used to identify message boundaries for each part and decode them separately. The various encoding mechanisms offered by MIME to represent binary data are 7bit, quoted-printable, base64, 8 bits and binary.

2.5. Web Services

Web services are a way of communicating between connected computers over a network in a machine understandable way. Thus it could be automated among computer programs. The underlying implementation of the service is hidden from the consuming parties, yet a standard interface was used. Thus the web services facilitated to make interoperable systems by connecting heterogeneous computers. Web service model consist of three roles[18]: service provider, service registry and service requestor and three interactions: publish, bind and find operations. Typically the service provider would publish its service description to service registry and a service requester could search for web services. Once a service requester finds the required web service, it could bind itself to the service for future interactions. The above operations have being implemented by standards such as WSDL[19], UDDI[20], SOAP [21]and WS-Security[22].

Web Services Description Language (WSDL) is an XML format for describing the web service and it would be bound into a concrete network protocol and message format to define an endpoint. Universal Description Discovery & Integration (UDDI) is a set of services supporting the discovery of web service providers and their technical interfaces. Simple Object Access Protocol (SOAP) is a standardized XML based data formatting protocol used in message exchanging between web services. However the SOAP protocol uses another network protocol to carry their payload through the network. Mostly HTTP or SMTP is used for this task. SOAP message has three parts: an envelope, a header and the body. Even though the SOAP header is optional, it can be used to pass application specific instructions such as information related to processing of the message. The body contains the SOAP call and response data. Optionally, we could add “SOAP Fault” section to carry error information within a SOAP message. WS-Security is a specification for web services to ensure the integrity and confidentiality of messages. It also supports a variety of token mechanisms to support user identification and business certifications. WS-Security is an OASIS Standard document produced by the Web Services Security Technical Committee.

2.6. Service Oriented Architecture (SOA)

SOA is a design pattern which emphasis the services provided by the connected components over a network. Considering what we discussed under web services, in SOA, the components would exalt only in one of the functionalities than providing several. Components could act together to provide a complete software. SOA was driven by the need of fulfilling the Business to business (B2B) needs where one business can come up with software components which could be later outsourced to another[23]. SOA design pattern made it easy to reuse and combine the discrete business processes and services. SOA could be designed using open standards such as web services and would provide interoperable heterogeneous system.

2.7. Representational state Transfer (REST)

Representational state transfer (REST)[5] is an architectural style for hypermedia systems which was defined by Dr. Roy Fielding in his doctoral dissertation. It gives a set of principles which guides REST and a set of interaction constraints to retain those principles. Hypermedia is a concept which was formed as early as in 1965[24] when researches build systems with linked content where a view could traverse through using documents itself. Nowadays, REST is applied to HTTP protocol extensively. While it could be applied to any system, it aligns well with the concept of web's architecture. The REST architectural style has defined six constraints, without defining the design of each constraint. The constraints are as follows.

- Client-Server
  - Allows client server components to grow independently and provides scalability
- Stateless
  - The server would not maintain a state and thus provides visibility to the client site monitoring system, scalability in server side due to reduce resource usage.
- Cache
  - State whether the responses can be cached or not and thus provides communication efficiency and scalability.
- Layered System
  - Layers provide independence over each component and provide scalability.
- Code on demand style
  - REST allows client functionality to be augmented at run time by downloading and adding code on demand. This simplifies the initial design of the client.
- Uniform interface
  - It applies the software engineering principle of generality to the components of the REST architectural style and thus provides simplicity.

REST architectural style identifies 3 types of architectural elements, including Data elements, Connectors and Components. These three types of elements are an abstraction of hypermedia system. However, it only focuses on the role of each element plays in a hypermedia system, rather than the actual implementation of each.

Data Elements

The nature of the data element is a key aspect of the architecture. REST components communicate by transferring a representation of a resource in a standard data type. The data type is decided based on what a recipient desires or simply the nature of the resource. As defined by Roy Fielding, the REST data elements are as follows,

- Resource - the intended conceptual target of a hypertext reference
- Resource identifier - URL, URN
- Representation - HTML document, JPEG image
- Representation metadata - media type, last-modified time
- Resource metadata - source link, alternates
- Control data - if-modified-since, cache-control

Connectors

REST uses various connector types to enclose the activities of accessing the above mentioned resources and transferring the resource representations. By having a defined set of connectors provides an abstract interface for component communications. This simplifies the process and provides separation of concern for the implementation of resource and communication mechanism used. As defined by Roy Fielding, the REST connector elements are as follows.

- Client – libwww, libwww-perl
- server – libwww, Apache API, NSAPI
- cache – browser cache, Akamai cache network
- resolver – bind (DNS lookup library)
- tunnel – SOCKS, SSL after HTTP CONNECT

Components

REST components are the element of the architecture which forms architecture with above mentioned connectors and data elements. As defined by Roy Fielding, the REST Components are as follows,

- origin server – Apache httpd, Microsoft IIS

- gateway – Squid, CGI, Reverse Proxy
- proxy – CERN Proxy, Netscape Proxy, Gauntlet
- user agent – Netscape Navigator, Lynx

2.7.1. Application of REST

As we have discussed the REST architectural style is based on the hypermedia system which is governed by the state of the resource. REST architectural style can be explained easily with the use of HTTP protocol since it is considered to be that REST is the set of guidelines on how HTTP is to be originally used. The operations of HTTP performed through request methods such as GET, PUT, POST, DELETE and content negotiation is used to define different types of resource representations. To design a complete REST based solution the following steps should be followed[25].

- Identify resources

First, depending on the system, the resource should be identified. For example, if the system in consideration is a help desk, one of the resources would be customers. The designer should always abstract the resource types to simplify the design. Each of these identified resources are then should be made up the URI's of the hypermedia system. For example, the URI to list all the customers in a bookshop the following URL can be used.

```
http://booksforsale.com/resources/customers
```

- Link resources

It is a must to link all the resources together to form a true hypermedia system. The state of the client would transfer from clicking a hyperlink which would direct the user to another linked page or a new representation of the current resource.

- Select and create formats

The resources of the hypermedia system have to be represented in a suitable format, such as XML, which then could be then converted to HTML or Rich text format. The client could request the content-type of the resource by setting the header values of the request or either setting the URI to represent the desired content type.

- Identify method semantics

With the use of HTTP protocol, it defines 8 methods including GET, POST, PUT, DELETE, OPTION, TRACE, PATCH and HEAD[26]. The designer should think of the system interns of CRUD (Create, Read, Update and Delete) in order to understand all the required method for each resource. The usage of each HTTP method should be in-line with its defined action.

- Select response codes

The response codes stands for the response codes the client would receive during the interactions with hypermedia system. These response codes are helpful for system debugging and exception handling. HTTP response codes are well defined and cover most of the aspects in hypermedia communication.

2.7.2. Use of HTTP request methods

When REST architectural constraints are applied to HTTP, the uniformity of interacting with the resources is governed by HTTP methods. The methods require to be uniformly defined for all the resources and thereby intermediaries don't have to know the resource type in order to understand the meaning of the request. Therefore the method definition goes along with how it is defined in the HTTP protocol itself. The definition of HTTP methods could be found on RFC 7231[26]. The specification defines three method properties; safe methods, idempotent methods and cacheable methods. The safe methods request should not result in a state change on the origin server. A method is considered idempotent if multiple identical requests would have the same effect for a single such request. Request methods can be defined as cacheable to indicate that responses to them are allowed to be stored for future reuse. The following table summarizes the HTTP request methods which could be used to interact with resources in REST architecture.

Table 2-1: HTTP request methods

Method	Description	Properties
GET	Transfer the current selected representation of the target resource.	Cacheable, Safe, idempotent
HEAD	Identical to GET. But the server does not send the message body	Safe,

	in response.	idempotent
POST	Process the representation in the request according to the resource's own semantics. Can use to create and append new data. The server responds with status code 201 If resource newly created.	
PUT	Replace all current representations of the target resource with the request payload. The server responds with status code 201 If resource newly created, 200 if successfully modified.	Idempotent
DELETE	Remove all current representations of the target resource. The server responds with status code 202 for accepting requests, 204 when there is no response content and 200 when action taken and has response.	Idempotent
CONNECT	Establish a tunnel to the server identified by the target resource and thereafter restrict its behaviour to blind forwarding of packets, in both directions, until the tunnel is closed. Use in proxies.	
OPTIONS	Describe the communication options for the target resource. This would help clients to determine the options and requirements associated with a resource.	Safe, idempotent
TRACE	Perform a message loop-back test where the final recipient of the request would reflect the message received. The client must not send a message body with the request.	Safe, idempotent
PATCH[27]	Uses for partial resource modifications since PUT only allow a complete replacement of a resource.	

2.7.3. HATEOAS

HATEOAS is the abbreviation of Hypermedia as the Engine of Application State. HATEOAS is one of the sub constraints of “Uniform interface” as specified by Dr. Roy Fielding. This constraint requires a server application to provide hypermedia responses which should assist clients to understand the available interactions with that application. Even though there can be few fixed entry points to the application, for a well-designed application which abides to this constraint could be used by clients without making any assumptions or binding them to a previously known set of interactions.

The interactions that could perform depends on the current resource representation client had received and thus works as the “engine of the application state”. This would allow server application to evolve independently and thus assist client-server architecture and scalability. The responses should include resource URL’s to perform further actions with explicit ‘link rel’ attributes which describes the resource.

2.8. REST vs. SOAP

While discussing about REST it is important to note that, REST is not the first and the only web service architecture which is in use. Simple Object Access Protocol (SOAP)[21] is another protocol specification for web services. SOAP primarily depends on XML data format for communication and runs on top of another application layer protocol such as HTTP. While SOAP has its own advantages, as pointed out by Jakl[28], REST has features that favour the design of efficient hypermedia system. Table A summarizes the comparison of SOAP vs. REST by Jakl on various aspects of web services.

Addressing the low end network connection with high latencies, would not be a complicated issue due to the support of cacheability. The scalability aspect of REST architecture would support the fact that a mail system is distributed and highly connected and would grow rapidly as it would gain more customer base.

Table 2-2: REST vs. SOAP

Aspect	REST	SOAP
1. Protocol Approaches	A custom protocol using existing protocol HTTP	Create a framework which provides a basement
2. Standardization of ● Addressing ● Methods ● Messages	Only the message payload is not standardized. Other two have well defined standards.	None of the aspects are standardized in this style.

3. Cacheability	Define intermediaries which reduce the network load.	SOAP messages does not differentiate cacheability
4. Statelessness	Stateless, and thus provide better scalability	Does not define server side state.
5. Security	Uses predefined methods and firewalls can interpret.	Common firewall products do not identify messages.

2.9. XML

Extensible Mark-up Language [29] (XML) is a subset of Standard Generalized mark-up Language and was developed by an XML working group formed under World Wide Web Consortium in 1996. Their design goals for XML included it being directly usable over the internet, supports variety of applications, should be easy to create; it should be reasonably clear and etc. XML documents consist with units called entities which contain data. Even though XML was originally designed to handle large-scale electronic publishing, nowadays it used for data exchange. Therefore the XML document is designed and developed to carry data between computers and application and was not intended for display formatting.

The specification has defined a list of requirements which should adhere by XML formatting. A format which adheres to the specification is called ‘well formatted document’. A well-formatted XML document could be processed by XML parser and otherwise it will result in an error. A well-formatted XML document must contain a root element and other elements are nested within the root and its closing tag. After the introduction of XML, many schemas and formats have been introduced and used for varies purposes. RFC 7303[] defines the rules to use when constructing Internet Media Types when sending XML. An example of the document object formatted in XML is as below:

Code Snippet 2-1: Example XML document

```
<letter>
<to>Recipient</to>
<from>Sender</from>
<heading>Example</heading>
<body>XML formatted mail</body>
</letter>
```

2.10. JSON

JavaScript Object Notation (JSON) is a text-based open standard which is similar to XML. JSON primarily used as a lightweight data interchange format. It is derived from the JavaScript scripting language for representing simple data structures and associative arrays, called objects. The JSON format was specified in RFC 4627[30] by Douglas Crockford. JSON's design goals were for it to be minimal, portable, textual, and a subset of JavaScript. Thus the processing of JSON is much faster compared to XML and is more human readable compared to XML.

JSON message has two possible structures. One is name/value pairs which is realized as objects and the other is ordered list of values which is realized as an array. Inside these structures JSON can represent four primitive types of data: strings, numbers, Booleans, null and the two structured types itself. An object begins with {(left brace) and ends with} (right brace) while an array would begin with [(left bracket) and ends with] (right bracket). The example JSON message in code snippet 2.2 shows the JSON representation of an object that describes a person;

Code Snippet 2.2: Example JSON document

```
{
"email":{
"sender”: "Sender Name",
"senderemail" : "sender@sender.com",
"receiverEmail" : "receiver@receiver.com",
"subject" : "Test Email"
"body" : "This is a testmail",
"attachments": [
{
     "id":0,
     "name": "test.txt",
     "contenttype": "text/plain"
}
]}}
```

2.11. JSON vs. XML

JSON and XML are used extensively in contemporary systems for data interchange. Both of the formats have their own advantages and disadvantages based on the use cases. Therefore it is important to understand the pros and cons of each technology. Alen [31] et al have done an extensive comparison on XML and JSON. Based on their analysis, they have pointed out the following factors

- Code and data model

XML documents are ‘well-formed’ and the technologies used for manipulating and formatting of XML documents are standardized by the World Wide Web Consortium (W3C). Due to this XML could be used for data validation. JSON on the other hand is modelled based on Java Script objects and thus, easy to parse by the OOP languages. JSON data validation is not yet standardized. However, there is an Internet Draft called JSON Schema which could be used for data validation.

- Accessing and extracting data

XML document could be modelled as a tree and then could be accessed for data extraction. There are many models and tools developed for XML data parsing such as DOM, XSLT. On the other hand JSON documents could be converted into objects and could be accessed as an object.

- Extensibility

XML documents could be used to store varies types of data by expanding XML attributes and CDATA sections. Even though this makes the document harder to read, it could be used to transfer documents containing images, graphs, etc. JSON data representation is limited to data types such as strings, Booleans, floating point numbers, etc. Therefore while JSON is ideal for simple data representations, XML could be used to represent complex documents.

Nurzhan Nurseitov et.al[32] has done extensive testing on the performance difference of XML and JSON interchange formats. Their study focuses on measuring the transmission times and resource utilizations in a client/server environment. The first study has done by running a single time-consuming transmission of a large quantity of objects and the second study has conducted by gradually increasing the number of objects per test. The test result of the first study is shown in the table. Their study has shown that JSON is significantly faster than XML when considering the higher number of data objects. However the study also found that the CPU utilization is higher for JSON processing.

Study One:

Table 2-3: JSON vs. XML performance

	JSON	XML
Number of Objects	1000000	1000000
Total Time(ms)	78257.9	4546694.78
Average Time(ms)	0.08	4.55

2.12. Authentication mechanisms

Authentication is described in Internet Security Glossary as the process of verifying an identity claimed by or for a system entity. The process could be broken into two steps as identification step and verification step. When an identifier was presented to the security system, the system would confirm the binding between the identifier and the entity. After authentication, the authorization process would take place to grant permission to a system entity to access system resources. Many authentication protocols have been introduced and used in today’s applications. In this section, we are discussion some of the popular authentication mechanisms used in the APIs and applications which were considered under this study.

2.12.1. HTTP Authentication

HTTP protocol offers a general framework for access controlling and authentication via a set of challenge-response authentication schemes[33]. The authentication schemes have been published in the IANA Authentication Scheme Registry. Under challenge response mechanism, the server could challenge a request from the client and the client could then provide the necessary identification information. The authentication scheme uses ‘WWW-Authenticate’ header field to denote the authentication scheme and the related parameters used in the scheme. Once the user is authenticated, the ‘realm’ could be used to denote the resources that are authorized under the given identifiers.

One of the popular authentication schemes which use the above mechanism is basic authentication scheme. The basic authentication mechanism name is defined as “Basic” and the authentication parameter realm is set to be required. When a user tried to access a protected resource the server would challenge with ‘Unauthorized’ status code and ‘WWW-Authenticate’ header field. The client software then should obtain a password and username from the user and concatenate them with a single colon “:” character. Then the sequence would be encoded with base64 encoding. Since the basic authentication mechanism passes the password in clear-text, it is required to conduct the communication over HTTPS rather than HTTP. However, basic authentication mechanism follows the HTTP semantic and thus complies with the RESTful constraint of being stateless.

2.12.2. Query Based Authentication

Query based authentication[34] is another authentication mechanism used in RESTful services where each and every request to the API is signed with a private credentials. Whenever a user generates a request, the requested URI’s parameters are re-organized in lower case alphabetical order and then signed by a hashing algorithm. The resultant signature is appended to the URI as a parameter. To avoid security threats against this mechanism, the users could deploy the support of HTTPS connection.

Code Snippet 2-3: Query based authentication request example

```
GET
/object?timestamp=1261496500&apiKey=Qwerty2010&signature=abcdef0123456
789
```

2.12.3. OAuth 2.0 Authorization Framework

Traditional client server authentication model works well for scenarios where the client authenticates itself directly to the server. The authentication mechanisms discussed in section 2.9.1 and 2.9.2 follows this model. However, in a situation where a new third party provides services as an interface to the original server, the client has to share the credentials with the third party. Sharing of credentials force third party apps to store it indefinitely and causes security concerns. Moreover, the third party gets full access to the server on behalf of the client even though the access in only required for some of the resources on the server.

OAuth was designed and developed to address these issues. In OAuth 2.0[35], four roles have been identified to perform the authorization process; a resource owner, resource server, client and the authorization server. The resource server and the authorization server interactions are not covered in the specification and in fact it could be the same server. The process starts when the client requests authorization from the resource owner. The authorization could be granted to the client using one of four grant types. After obtaining the authorization, the client then requests for an access token from the authorization server. The access token can then be used to access resources from the resource server. The specification was designed to use with HTTP and any other protocol are considered to be out of scope of the specification.

The OAuth 2.0 protocol itself is not an authentication protocol. The granting of authorization would happen after proper authentication of resource owner or the client depending on the grant type. This allows systems to use existing authentication protocols such as OpenID Connect and SAML. OAuth 2.0 doesn’t provide confidentiality of information and thus forces to implement it over a secure connection such as TLS.

2.13. REST based email systems

2.13.1. “Reinventing Email using REST”

Paul Prescod has written an article on “Reinventing Email using REST”[36] with the idea of acting as a suitable basis of replacing the existing email system with a complete HTTP based system using REST architecture if HTTP becomes a dominant protocol. Paul Prescod describing the basic concept in his article; *“To get a representation from a resource we use the HTTP method GET. To overwrite a resource based on a representation we use the PUT method. To delete a resource we use the DELETE method. To modify a resource based upon its current state (extend it or mutate it) we use POST.”*

In his article, he has specified some generic cases of email usage, such as “Sending a Message”, “Adding a Queuing Mailbox”, “Managing Mailboxes” and “Delivering Mail to the End User”. The proposed system would have the following list of benefits:

1. Reuse of standard server side tools like Apache, Squid, standard web resource search and management tools. Compatibility with generic web client-side tools like browsers. Every

web browser becomes a mailbox browser "for free". Web management tools would become mailbox management tools.

2. Integration of email namespace with Web namespace means that web documents may refer to mail messages/mailboxes and vice versa.
3. Integration of email namespace with Web namespace would allow individuals to have one identifying URI per "persona" rather than a home page URI and a mailto: URI. This could be achieved today if mail programs could extract the appropriate metadata from a home page, but this kind of extraction is a core concept of REST: "hypermedia is the engine of application state."
4. Features of HTTP protocol become available to mail programs "for free" (e.g. Security, caching, reliability extensions, etc.)
5. Implementing a mail user agent requires knowledge not of three protocols, but of one: HTTP.

2.13.2. HTTP Access to Email Stores

Lisa Dusseault has submitted an Internet-Draft "HTTP Access to Email Stores"[37], in which she formulates a standard format and a standard navigation mechanism for accessing email stores via HTTP protocol. This would provide interoperable access to mailboxes and messages over HTTP. However, this internet draft has been expired and hasn't proceeded into a standard. This Internet draft does not try to replace the existing protocol stack of the traditional email system. Rather, it brings up the possibility of using HTTP features in accomplishing email use cases. Below features could be implemented by using HTTP for accessing an email store.

- A persistent URL to download emails.
- A URL and response formats (XML, PDF) to download the mailbox content.
- Discovery capabilities for the email store.
- Delete method to remove listing of an email from an email store.
- POST method to add emails to an existing email store.

She has proposed the RESTful ATOM protocol for the purpose of providing content listing of the email store. By using ATOM, it can be more easily integrated into clients and servers which do not support HTTP email access functionality off the shelf. The ATOM protocol provides direct functionalities which simplify the operations such as downloading the content in XML format via GET request and much more. The functionality can be easily provided by an existing feed reader and the feed reader then can be used as an HTTP email store browser.

The specification is only for the read-only access to the email stores and it has not covered the email sending, delivering or managing spam or functions such as creating mailboxes. Although with these limitations she has listed many of the use cases that this type of an email system would be helpful in the Internet. Below is a summary of possible use cases for when HTTP protocol access is available for email stores.

1. Fully fledged client utilities, which can get benefits from data in the email stores and augment the functionality of it. (Ex: Calendar client)
2. Third party services which would access the email content and augment their services (Ex: Twitter like service automatically fetching data from email store)
3. Mashups where several sites integrated together via the standardized API, and would be able to provide collective functionality. The sites might include mailing lists.

More than the functionalities offered by the proposed system, it is interesting to study the formatting which as being used to represent the email message body in XML. Since it is based on ATOM protocol the message body can be considered as a feed. The feed begins with 'feed' root element and 'entry' element marks the beginning of an email body. Four elements have been used to describe the feed itself.

1. title' element – mailbox name
2. 'id' element – universally unique mailbox id
3. 'author' element – owner of the mailbox
4. Link element – contains the URL to the feed and sections in the feed. The link element uses ‘rel’ attribute values to describe the link type.
   a. rel=self : the URL to the feed itself
   b. rel=alternative: Other URLs to the mailbox, such as IMAP
   c. rel=service : URL to the document describe server features
   d. rel=current: URL to the most recent entries in the feed
   e. rel=next: URL to the next section of entries in the feed
   f. rel=prev URL to the previous section of entries in the feed

Within the feed element, to represent the contents of each feed, an ‘entry’ element would be used and within that element, the following elements have been used to represent email body.

1. 'id' - depends on the message id given by IMAP and SMTP.
2. 'title' - contains the subject of the message.
3. 'updated' - date received or the last time it was changed.
4. 'published' - sent date
5. 'summary' - contains a piece of body of the email
6. ‘link’ - provides links to the entire message and the attachments. This has the following rel attributes.
    a. rel=self: human readable version of the message
    b. rel=alternative: machine readable format of the message
    c. rel=enclosure: attachments
    d. rel=related: maps directly to message in the “Reference” header
    e. rel=in-reply-to maps directly to messages in “In-Reply-To” header

2.13.3. Restful interface for database based email server

Karunarathna et al[38] have developed a restful interface to database based email server called DBmail[39] with the aim of providing users with a clean HTTP based interface which follows REST constraints. Their system also provides Restful interface to the offline mail client too. Authors have used DBmail, which stores all the emails in relational database and provides IMAP, POP3 and SMTP protocols to communicate with other email systems. However, DBmail community itself has shown interest in providing a RESTful interface which works as an event-driven daemon. The proposed daemon would return JSON responses and provide limited functionality as of now.

Karunarathna et al has designed their RESTful interface to respond with XML. Since XML is a well-established mark-up language, the authors could re-use the existing frameworks and design their own tags to represent the email message content. The authors have given priority to implementing a set of well-established IMAP commands which cover most of the generic usage cases of users such as “login”, “logout”, “select”, “close”, “create” and “noop”. Using these commands and the resources identified during their study, the following capabilities are offered through their API:

- Host/restmail/user/lastLogin : Return the last login time
- Host/restmail/user/logout : Return the last login time
- Host/restmail/nap : Make sure server is connected or not for online offline feature.
- Host/restmail/user/inbox/All : Retrieve all mails
- Host/restmail/user/inbox/size : Size of all mails
- Host/restmail/user/inbox/delete : delete all mails
- Host/restmail/user/inbox/ 46576ac : Get specific mail
- Host/restmail/user/inbox/ 46576ac/size : Get the size of the mail
- Host/restmail/user/inbox/ 46576ac/Delete :Delete specific mail
- Host/restmail/user/inbox/ 46576ac/header : Get the header of the mail
- Host/restmail/user/inbox/ 46576ac/header : Get the header of the mail
- Host/restmail/user/inbox/46576ac/mailbody?mime=2432 : Get specific part of the mailbody
- Host/restmail/user/inbox/ 46576ac?mark=seen/unseen : Mark mail as seen or unseen
- Host/restmail/user/inbox/move/?uid=46576ac&from=folder1&to=folder2 : Move mail from folder to folder
- Host/restmail/user/inbox/Flags/All?Date=2010/02/23 : Retrieve all mails on given date
- Host/restmail/user/inbox/Flags (Seen, Unseen, Deleted, Recent) : Retrieve the mails relates to each flag
- Host/restmail/user/inbox/Flags? Date= 2010/02/23 : Retrieve the mails relates to each flag and given date
- Host/restmail/user/inbox/All? Amount=10& from= 2010/02/23&to=2010/02/25 : Received mails from 2010/02/23 to 2010/02/25 at inbox retrieve 10 by 10

However, the URL's are constructed by giving prominence to the actions rather than to the resources. For example, to delete all the mails, the URL contains the word 'delete' and the user is supposed to perform a 'GET' request to the targeted url. Thus the authors are limited to using only GET and POST request method for the entire API. The POST request method has used only when email sending functionality. Moreover, the XML responses have no notion of hypermedia concept built into it. Therefore, in my opinion, the API couldn't consider to be RESTful.

2.13.4. RESTMAIL by Marcin Bazydlo

Marcin Bazydlo has proposed and implemented a model of restful email system[40] for his master's thesis, taking the basic idea given by Paul Prescod[36] and improving it further into a complete email system. He has designed the email system with email sending and receiving functionalities. The mail sending functionality has implemented following the idea of Paul Prescod, where mail sending is changed from PUSH mechanism to PULL mechanism. The sender will create the email as a web resource in her own server and would then create a notification web resource in recipient mailbox. If the recipient wishes to download the content, she can pull the necessary email parts from sender's server.

Marcin has defined four major resources while designing his proposed email system. The resources are "message", "notification", "receipt" and "directory". Notifications are used to inform about new messages while receipts are used to convey the status of an operation. He has used YAML to represent the responses generated by his API. The responses are well documented and support hypermedia concept. Sample response for GET request to message resource is shown in code snippet 2.4.

Code Snippet 2-4: Example of GET request on Message resource of RESTMAIL system.

```
:type: message
:title: Msg two
:id: "0588533542101"
:updated: 2009-06-11T16:42:05+02:00
:published: 2009-06-11T16:42:05+02:00
:links:
- :type: text/html
  :href:
http://localhost:3000/kajko/box/0521@localhost/01869@localhost/
  :rel: related
- :href: http://localhost:3000/kajko/box/0588533542101@localhost/
:rel: self
:author:
 :name: Unknown
```

Marcin has used GET, PUT, POST and DELETE HTTP request methods properly on each resource to perform actions accordingly. Thus he has not used any actions in the URLs to safeguard the Restfulness of the API. Although this system offers the complete email solution via HTTP via restful API, the integration with the traditional mail system is not possible.

3. STUDY OF EMAIL APIs

Email has become a business critical technology in today's world. To support this, many free and commercial email systems have been developed since the introduction of email. In this chapter, we are doing a critical comparative study on the Restful APIs provided by some of the popular commercial products. In the next chapters, we would discuss the critical features which should be supported by an Email system and how each of them is done in the most effective way.

3.1. Gmail REST API

Gmail is a popular email service provided by Google Inc. from 2004, and has gained popularity among the webmail users. Gmail offers IMAP connections to its email stores apart from its standard web mail interface. With the popularity of mobile device usage and their applications, in June 2014, Google introduced a RESTful interface to Gmail[6]. By using the new API, programmers were able to develop applications for mobile devices which could communicate with Gmail without using their web mail interface or the IMAP protocol. The Gmail REST API supports functionalities such as reading, composing and sending mails after proper authorization. It also identifies email threads and labelling used by Gmail. The authorization and authentication is handled by OAuth 2.0 and it provides scopes and tokens to determine authorization. The API uses JSON payloads for communication and has introduced API client libraries to ease up programming against the API. The Gmail API, commonly used by application developers to do read-only mail extraction, custom labelling of emails, automated mail sending and to migrate email account from other providers. Gmail REST API has identified five primary resources and has built functionalities around it.

- Messages: Messages are a basic unit of a mailbox which holds the email message. Users are able to create or delete messages. However the messages can't be updated or changed.
- Labels: Labels are another basic unit which used to categorize emails depending on user and system requirements.

- Drafts: Contains draft email messages. Draft email messages can’t be modified; however, they could be replaced by a newer version. Sending a draft mail would result in deleting the draft and creating a new mail with SENT label.
- History: History resource is a collection of recently modified messages.
- Threads: Threads resource represents a conversation and it could be only deleted and updated with new messages. Users can’t create a thread resource.

Resources are represented as JSON objects. The “Message” resource has several unique parameters such as “threadId” and “labelIds[]” to support threading and labelling features. The “snippet” property contains a short part of the message for quick viewing while “payload” property contains all the parsed message parts. The parts include standard RFC 2822 header array, parts array representing container MIME message parts and message body parts. Code snippet 3-1 shows a sample representation of message resource response in Gmail REST API.

Code Snippet 3-1: Sample representation of Message resource of Gmail REST API

```
{
  "id": string,
  "threadId": string,
  "labelIds": [
    string
  ],
  "snippet": string,
  "historyId": unsigned long,
  "payload": {
    "partId": string,
    "mimeType": string,
    "filename": string,
    "headers": [
      {
        "name": string,
        "value": string
      }
    ],
    "body": users.messages.attachments Resource,
    "parts": [
      (MessagePart)
    ]
  },
  "sizeEstimate": integer,
  "raw": bytes
}
```

Apart from the five main resources, the API provides a sub resource for “messages” as “attachments”. This would allow clients to download attachments separately from the message body if the attachments are external to the mail body. In that case the message body contains the attachment ID’s, their filenames and MIME types. The clients could then download each attachment based on its ID. The attachment resource has ID, size and data parameters. The data are presented as base64 encoded string. Gmail REST API has used GET, POST, PUT, PATCH and DELETE HTTP request methods to interact with the above resources. The API uses POST for resource creation and PUT for resource update function as shown in below example:

```
• POST /userId/labels : creates a new label
• PUT /userId/labels/Id: Updates the specified label.
```

It is interesting to note the usage of the PATCH HTTP request method in the Gmail REST API. The patch method is used for partial updates of resources as shown in below example:

- PATCH /userId/labels/id: Updates the specified label. This method supports patch semantics.

Although, most of the URL’s of the API is constructed properly to give prominence to the resources and then to run the actions by the use of the appropriate HTTP request method, some of the URL’s are constructed with actions. For example the code snippet 3-2 has a list of URLs which has verbs as resources

Code Snippet 3-2

```
• POST /userId/drafts/send: Sends the specified, existing draft
• POST /userId/messages/id/modify:  Modifies  the  labels  on  the
  specified message.
• POST /userId/messages/send: Sends the specified message
• POST  /userId/messages/id/trash: Moves the specified message to
  the trash.
• POST  /userId/messages/id/untrash: Removes the specified message
  from the trash.
• POST /userId/threads/id/trash: Moves the specified thread to the
  trash.
```

3.2. Outlook Mail REST API

Outlook Mail REST API[7] is a part of Office 365 REST APIs developed by Microsoft to provide programmatically access to their Office 365 suite. This includes messaging platform, event & calendar, contact details and OneDrive files/folders. The REST API would help developers to build apps for mobile devices and cloud platforms easily. Apart from the REST API, Microsoft provides client libraries which could be used for programming. The Outlook Mail REST API could be used to perform mail activities such as reading, composing, sending messages & attachments and to manage folders.

Outlook API supports 4 HTTP request methods GET, POST, PATCH and DELETE. Microsoft has used JSON as the request/response format for their API and is using the Microsoft Azure Active Directory and OAuth to authenticate. The API specifies Folder, Message and User as major resources.

- User: A User in the system. Microsoft has used "me" to indicate current users' email address itself. User resources contain 'RootFolder', 'MailboxGUID', "folders" etc.
- Folder: Is a folder in the user's mailbox such as Inbox or Sent Items. This resource may contain resources such as other Folders and Messages. Users could perform Get, Create, Update, Delete and Move/Copy operations against this resource.
- Message: A mail message in mail directory. The 'attachments' come as a sub resource from Message resource and could be accessed by its id. The availability of attachments is indicated with "HasAttachments" parameter. The message resource has an email body preview property as well as the whole email body. The format of the 'body' of the email is denoted with 'ContentType' property. Even though the message body provides a WebLink property which could be used to open the message in the Outlook web app, the resource representations do not denote any notion of hypermedia. Message resource supports Get, Send, Reply, Forward, Update, Delete, Move/Copy, get attachments, create attachments and delete attachment operations.

Important factors to note in Outlook Mail API is the use of the PATCH request method instead of PUT and the schematics used in Move/ Copy operations of folders and messages. Following is an example of the PATCH request method to update the message resource. The data would be sent as JSON encoded and the server would reply with status code and resultant resource state.

```
PATCH https://outlook.office365.com/api/v1.0/me/messages/{message_id}
```

The Move and Copy operations have been designed by appending ‘move’ or ‘copy’ to the end of the URL used to denote the resource which wants to be moved or copied. The destination would be send as JSON encoded POST data to the server. Following is an example of a move operation for a message in Outlook API.

```
POST
https://outlook.office365.com/api/v1.0/me/messages/{message_id}/move
```

Message sending in Outlook mail API covers several possible scenarios such as “only the fly”, “sending drafts messages”, “creating new draft message”, “create a reply draft message”, “create a reply to all message” and forward messages. On the fly mail sending is performed by posting email content to the following URL.

```
        POST https://outlook.office365.com/api/v1.0/me/sendmail
```

While a creating draft message is simple as posting to the message resource, to send draft messages, the following URL is used. The message_id is the message ID of the draft message.

```
POST
https://outlook.office365.com/api/v1.0/me/messages/{message_id}/send
```

Similarly, replying and forwarding URL’s are as follows.

```
POST
http://outlook.office365.com/api/v1.0/me/messages/{message_id}/reply
POST
http://outlook.office365.com/api/v1.0/me/messages/{message_id}/forward
```

3.3. Zimbra REST API

Zimbra Collaboration server[41] is an open source collaboration server consisting with features such as email system, calendar service, file sharing, document management, etc. All of these features and administrative functionality is provided through a web interface [20]. Zimbra provides a desktop client which would allow its users to sync content from the server to the desktop and this in-turn provides mobility. Zimbra has provided a REST API and a SOAP API both. The REST API facilitates applications to access the data stored by a Zimbra server such as emails, calendars, and contact address. The URL template for the REST commands in Zimbra is as follows:

Code Snippet 3-3: Zimbra URL Template

```
{protocol}://{host}:{port}/home/{user}/{object}?{params}
protocol : transport protocol
host : zimbra server IP or hostname
port : zimbra server port number
user : user account
object : objects or parameters to act upon
```

Zimbra supports a variety of response formats based on the resource and what user specifies in the query URL. The users can specify the required format as a parameter value as shown below:

```
http://localhost:7070/home/john.doe/tasks?fmt=ics   : provides ics format
http://localhost:7070/home/john.doe/tasks?fmt=xml : provides xml format
```

The authentication process is supported by allowing cookie Auth token check or check for query parameter Auth token or prompting for basic authentication. Zimbra uses only the ‘GET’ and ‘POST’ request methods in their API and has defined the below set of tasks which could be performed against the API.

- Get folder : Get the items in the folder (Use GET)
- Import Messages : Imports a message to a mail folder (Use POST)
- Get Contacts: Gets the contacts in the designated folder. (Use GET)
- Import Contacts: Import contacts to a designated folder. (Use POST)
- Get Calendar: Gets the appointments from the calendar (Use GET)

- Get FreeBusy: Gets the calendar. (Use GET)
- Import Appointments: Import appointments. (Use POST)
- Get Tasks: Gets the calendar (Use GET)
- Get Item: Gets an item (Use GET)
- Get Briefcase: Gets the list of items in the briefcase folder (Use GET)
- Get Briefcase Item: Gets a specified item from the briefcase folder (Use GET)
- Export mailbox: Exports the entire contents of a mailbox.(Use GET)

3.4. Sendinc API

Sendinc [42] is a service which can be used to encrypt email messages without having to implement encryption mechanisms into the users own email system. The website of Sendinc provides an interface for email communication provided that the users have already registered with Sendinc. Both sender and the receiver should register into the site for sending email and receiving it. The keys used for encryption would not be stored by any third party or by Sendinc Therefore, only the recipients would be able to read the emails. The importance of this service would be that it provides a mechanism that users should not need to create their own keys or to publish and securely share the key information but still would be able to send and receive emails securely. Sendinc would generate the necessary random keys and would email it to the recipient in a form of a link without keeping a copy of it and following least privileged principle.

Sendinc provides an API which would provide a way to integrate secure email into user applications. Sendinc has provided two API types, SMTP API and REST API. While SMTP API facilitates existing email systems and client software to integrate Sendinc services, REST API provides a mechanism to send and receive encrypted messages for HTTP enabled application. Here we focus on the REST API provided by Sendinc to identify its capabilities. The Sendinc REST API uses GET and POST HTTP request methods for its operations and provides responses in XML or JSON depending on the requests extension. The authentication mechanism supported by Sendinc is basic authentication and the communication is encrypted using 256 bit SSL to provide transport encryption. It is interesting to look into the message formatting offered by Sendinc API. The code snippet 3-4 shows the API response to the account information in JSON format.

Code Snippet 3-4: Account resource representation

```
{
"account":
  {
    "email": "email@address.com",
    "first_name": "Bill",
    "last_name": "Lumbergh",
    "date_created": "2011-07-11 18:01:22",
    "type": "pro",
    "max_messages_per_day": 200,
    "max_recipients": 100,
    "max_attachment_size": 209715200
  }
}
```

Apart from the email, first name, last name and the date created, other information tags are of business interest to Sendinc. Similarly, to send emails, the API has specified to POST pre-defined set of parameters to “message.json” or “message.xml” depending on which format the client wants to consume. List of parameters which should include in the message as follows;

- email - Sender email. (Must match Sendinc user unless Corporate user)
- recipients - Recipients. Separated by commas.
- recipients_cc - CC recipients. Separated by commas.
- recipients_bcc - BCC recipients. Separated by commas.
- subject - Message Subject
- message - Body of message.
- copy_me - Send a copy of the message to the sender
- notify - Receive an email notification when a recipient opens your message.
- expires - Set an expiration date (in days) for your message to expire

Users could retrieve a message by accessing the correct URL and performing a GET operation. This results in a JSON or xml response based on user preference. The message body consists of parameters such as body, subject, sender_name, sender_email, created, expired, recipient and attachments. Attachments handling of the messages are done by using multipart/form-data formatted POST request. The retrieval of attachment could be performed by a GET request to the attachment identified by its ID. If the response could not be completed, the server would generate error messages with HTTP error code accordingly. It is interesting to note that Sendinc hasn’t followed any hypermedia convention and provide URL’s within the response.

3.5. Postmark REST API

Postmark[43] provides SMTP service that made possible its customers to send HTTP emails at it and Postmark would process the emails and then send it out to the designated recipients. It would keep a track of the email transactions and provides an administrative interface which facilitates its users to monitor volume, bounces, spam complaints, and send activity. Postmark also provides an inbound HTTP service which would accept various format types of emails and would output a JSON format email from the designated recipient. This functionality provides a way for web applications to communicate with traditional third party SMTP/POP email servers without implementing their own mail servers.

PostMark provides a REST API for their customers, which would accept emails in JSON format with the POST HTTP method. Authentication would happen through API a key which passes through HTTP headers. Keys could be bound to libraries for automation and further usages. The communication is secured by the use of HTTPS layer for transportation. PostMark API provides both HTTP response codes and API error codes. Below is a list of API error codes with regards to the interest of our study.

- 300 - Invalid email request - Validation failed for the email request JSON data that you provided.
- 402 - Invalid JSON - The JSON data you provided is syntactically incorrect.
- 403 -Incompatible JSON - The JSON data you provided is syntactically correct, but still doesn’t contain the fields we expect.
- 604 - You don’t have delete access -You don’t have permission to delete Servers through the API.
- 701 - Message doesn’t exist
- 813 - Not a valid email address or domain.

PostMark, through their API, provides functionalities such as sending mails, bounce mail monitoring and outbound mail statistic analysing. PostMark email sending is performed as POST request with the payload shown in code snippet 3-5.

Code Snippet 3-5:

```
{
  "From": "sender@example.com",
  "To": "receiver@example.com",
  "Cc": "copied@example.com",
  "Bcc": "blank-copied@example.com",
  "Subject": "Test",
  "Tag": "Invitation",
  "HtmlBody": "<b>Hello</b>",
  "TextBody": "Hello",
  "ReplyTo": "reply@example.com",
  "Headers": [
    {
      "Name": "CUSTOM-HEADER",
      "Value": "value"
    }
  ],
  "TrackOpens": true,
  "Attachments": [
    {
      "Name": "readme.txt",
      "Content": "dGVzdCBjb250ZW50",
      "ContentType": "text/plain"
    },
    {
      "Name": "report.pdf",
      "Content": "dGVzdCBjb250ZW50",
      "ContentType": "application/octet-stream"
    }
  ]
}
```

PostMark is using base64 encoding for attachment encoding and would pass it with the message itself. PostMark hasn't included URL's in their responses and has no notion of hypermedia in the API. It is interesting to study the implementation of outbound/inbound message search functionality implemented by PostMark. For this, they have used query string parameters in URL, where they have specified various parameters such as message count, offset, recipient, fromemail, tags and subject. Apart from email sending, for all the other functionalities, PostMark uses the HTTP request method GET. Example outbound message search query is as follows:

```
"https://api.postmarkapp.com/messages/outbound?recipient=john.doe@yaho
o.com&count=50&offset=0&tag=welcome" \
```

3.6. Email Yak REST API

Email Yak[44] is a service which can be used by web applications to send and receive emails. A user can create an account and then register a domain which is used as the domain that the emails are send out and receives and then use the REST API with JSON message formatting. The users can change the MX records in their domain name server to point to the domain name created in Email Yak service. For sending emails through Email Yak service, the users have to POST JSON formatted messages to a specified URL by the API. The JSON message contains parameters such as FromAddress, ToAddress, Subject, TextBody and Attachments. The attachments are base64 encoded and would be sending in line with the message itself. Code snippet 3-6 shows email sending request in Email Yak API.

Code Snippet 3-6: Email Sending Request for Email Yak.

```
POST  https://api.emailyak.com/v1/private_api_key/json/send/email/
{     "FromAddress" : "from@example.com",
      "FromName" : "Sam Jones",
      "SenderAddress" : "sender@example.com",
      "ToAddress": "receiver1@example.com, receiver2@example.com",
      "ReplyToAddress": "replyto@example.com",
      "CcAddress": "receiver3@example.com, receiver4@example.com",
      "BccAddress": "receiver4@example.com, receiver5@example.com",
      "Subject" : "Test",
      "HtmlBody" : "Hello",
      "TextBody" : "Hello",
      "Headers" : [{"Name" : "CUSTOM-HEADER-1", "Value" : "Header Value"},
                   {"Name" : "CUSTOM-HEADER-2", "Value" : "Header Value"}]
      "Attachments" : [{"Filename" : "File1.txt", "Content" : "SG93ZHkh"},
                       {"Filename" : "File2.txt", "Content" : "SGV5IEhleSEh"}]
}
```

Similarly the GET request method on API specified URL could be used to retrieve the email message content. Email ID and a Boolean value of whether to retrieve headers or not would be passed in as parameters in the URL and it would output JSON message similar to the above example. Example of mail retrieving URL is as follows:

```
https://api.emailyak.com/v1/private_api_key/json/get/email/?EmailID=Ra
ndomEmailID&GetHeaders=True
```

Similarly, the user could use API to obtain a list of all the emails or new emails from the email store. Email Yak has featured an email delete functionality which could be used to purge emails and their attachments. For this, the user has to POST the required email ID to the API. For this the API has specified the following URL:

```
https://api.emailyak.com/v1/private_api_key/json/delete/email/
```

Analysing the URLs used by Email Yak API, it is apparent that rather than identifying resources, they have given prominence to actions such as ‘get’, ‘delete’, ‘send’. This way, we would not be able to use HTTP request methods properly since a URL is bind to one single action. The Email Yak API has not covered hypermedia concepts in their API too.

3.7. Context.IO Email REST API

Context.IO [45] is a web service which provides an API, that when connected to an email store, would provide instant access to data such as email content, attachment, etc. This would turn the mailboxes into online data stores and would provide developers to make applications which would enable web-based previews for the attachments, get version history of attachment files, get a list of modifications to the two versions of attachments, etc. API would return JSON messages and this would allow a web application to access email store without IMAP protocol support. Context.IO offers two API versions; Lite and 2.0 API. The Lite version has lesser features and focused on providing lightweight operations while it receives emails. The 2.0 API is focused on providing a much more complete feature set where the clients would be able to use contacts lists, directories, threads and historical data.

Context.IO API provides authentication through OAuth2. The user should need a consumer_key which is sent with the request and a consumer_secret which should be kept as a secret. Using the secret, a signature is generated for the request and the signature is included in the request itself. When Context.IO receives the request, it would check for the validity of the signature and would authenticate the request. For now Context.IO provides support for HMAC-SHA1 for signature generation and sending the data would be done through HTTP AUTH headers. Context.IO provides much functionality via HTTP POST, GET and DELETE request methods.

Context.io 2.0 API has implemented five major resources; messages, threads, files, contacts, and web hooks. It has no notion of hypermedia content and have used HTTP request methods PUT and POST both for content creation & modification. Even though there is a ‘message’ resource, sources and contact resources too have messages as a sub resources. This would helpful in filtering messages based on contact name or the folder name without setting other parameters. The 2.0 API allows clients to move messages between directories and delete messages. These features are not present in Lite version. However the API does not support email sending functionality. The API also provides customized error messages which enable clients to figure out whether the error is server side or client side. Example of email body retrieval in Context.IO is shown in code snippet 3-7.

Code Snippet 3-7: Email body retrieval in Context.IO 2.0

```
GET https://api.context.io/2.0/accounts/id/messages/message_id/body
[
  {
    "type": stringMIME type of message part being fetched,
    "charset": stringencoding of the characters in the part of message,
    "content": stringthe actual content of the message part being pulled,
    "body_section":  string  indicating  position  of  the  part  in  the  body
structure,
  }
]
```

3.8. Mailgun REST API

Mailgun [46] is a programmable email platform which allows programmers to build email capabilities into their own web applications easily. The functionality includes sending and receiving of messages, tracking messages, forwarding messages and storing messages and event handling. For this the API has defined four resources; domains, messages, stats and events. The API uses GET, POST, PUT and DELETE request methods for its functionality. The POST request method is used for new content creation while PUT has been used for content update. Their REST API provides JSON response messages while error messages are passed using standard HTTP error codes. Mailgun also has implemented wrapper libraries which would help developers accessing the API.

Mailgun API offers basic authentication over SSL/TLS connection for security. The mail sending functionality is a major part in this API and uses could send both MIME formatted messages and individual parts of an email. The API has specified a set of parameters including “from”, “to”, “subject”, “text” and “attachment”. Successful message sending would result in a

JSON response. Mail receiving, forwarding and storing would uses routes defined by the user. Email retrieval format depends on the content header parameter of the request. The API returns error codes using the following HTTP response codes:

Table 3-1: Mailgun customized error codes

Code	Description
200	Everything worked as expected
400	Bad Request - Often missing a required parameter
401	Unauthorized - No valid API key provided
402	Request Failed - Parameters were valid but request failed
404	Not Found - The requested item doesn't exist
500, 502, 503, 504	Server Errors - something is wrong on Mailgun's end

3.9. PostageApp API

PostageApp[47] is a service which facilitates to design, send and analyse emails easily. The users can have their own templates for the email for reuse using HTML and CSS, and can customize the templates with variables to personalize the emails and analytical capabilities. PostageApp has provided several plugins and frameworks for simplified usage of their API. A user of PostageApp could access the API using one of those plugins or using their own code. PostageApp API depends only on POST HTTP request method and responds with JSON messages. The POST request content type has to be set to JSON for correct handling of the request at the API endpoint. The code snippet 3-8 shows account information retrieval example.

Code Snippet 3-8: PostageApp account information request

```
curl -v \
-H "Content-type: application/json" \
-X POST \
-d ' { "api_key" : "PROJECT API KEY" } ' \
https://api.postageapp.com/v.1.0/get_account_info.json
```

Error messages are handled by the API and would provide detailed error information via JSON response. PostageAPP uses API key for user authentication and this has to be used in each and every message and SSL/TLS is used to provide security between web application and API. When sending messages, the attachments are embedded in JSON messages in base64 encoding with the content type tag. PostageApp does not have any notion of hypermedia content. Since it does only support POST requests, it couldn't be considered as a restful API.

3.10. Yahoo! Mail Web Service

Yahoo! is one of the first free email service providers which became popular with their web mail interface. The Yahoo! Mail Web Service gives programmatically access to the developers to build web application which could use to interact with user mail accounts. It is important to note that this is a web service and it is not Restful. However, we have included this API to get a better understanding of the differences of the two methodologies.

Yahoo! mail web service has recently introduced an OAuth authentication mechanism over their previous BBAuth mechanism. Yahoo has introduced libraries which should be used for programming and it covers JSON endpoints and OAuth authentications. The libraries have implemented the JSON - RPC specification on top of the web service and the requests follow the specific serialized JavaScript object. The formatted JavaScript object has the following properties.

- Method: Method indicated the name of the method called by the request. At present there are twenty three methods supported by Yahoo! API. The methods included operations for manipulating mail Folders, Message manipulation, Attachment handling and SPAM filtering.
- Params: an array containing the method parameters.
- Id (optional): the ID of the request, allows asynchronous clients to match a response back up with the original request. The service responds with serialized JavaScript as well. Once again, the JavaScript object follows a specific data format:
- Result: the return from the API method must be null if there was an error.
- Error: an error object resulting from the call being made (like an exception), must be null if there was no error.

3.11. Summary of Commercial API

Most of the commercial APIs are focused on providing a specific set of features and has been built around those features. However, fully featured email clients such as “Gmail” and “Outlook” have developed to cover most of the generic requirements of an email user. Considering the features offered by Even though the implementation and feature additions differ from these email systems, core functionalities remain the same.

Table 3-2: Summary of Commercial API

Service	Purpose	Protocols	Data Formats	Authentication
Gmail	Full email system	REST/IMAP /SMTP	JSON	OAuth 2.0
Outlook	Full email system	REST/IMAP/PO P/SMTP	JSON	OAuth 2.0
Zimbra	Full email system	REST/SOAP	XML, JSON	Basic HTTP
Sendinc	Send and receive secure messages	REST/SMTP	XML, JSON	Basic HTTP
Postmark	Transactional Mail	REST/SMTP	JSON	Token in Auth Header
Email Yak	Email for web apps	REST	XML, JSON	Query Auth
Context.io	API for email store	REST	JSON	OAuth 2.0
Mailgun	Transactional Mail	REST/SMTP	XML	Basic HTTP
PostageApp	Transactional Mail	REST	JSON	Basic HTTP
Yahoo!	Full email system	JSON-RPC	JSON	OAuth 2.0

3.12. Comparison of HTTP methods uses in APIs

The table 4-1 summarizes the use of HTTP request methods of the REST email systems discussed in section 2.13 and the APIs studied in chapter 4. A tick sign indicates that the request method has been used as explained in section 2.7.2. A cross sign indicates that the method has been used for function outside of the specification.

Table 3-3: Summary of HTTP method use in API

EMAIL API	GET	POST	PUT	DELETE	PATCH
PRESCOD [36]	✓	✓	-	-	-
HTTPMAIL [37]	✓	✓	-	✓	-
REST/Offline Mail[38]	X[1]	✓	-	-	-
RESTMAIL[40]	✓	✓	✓	✓	-
Gmail[6]	✓	X[2]	✓	✓	✓
Outlook[7]	✓	X[3]	-	✓	✓
Zimbra[41]	✓	✓	-	-	-
Sendinc[42]	✓	✓	-	-	-
Postmark[43]	✓	✓	✓	✓	-
Email Yak[44]	✓	X[4]	-	-	-
Context.IO[45]	✓	✓	✓	✓	-
Mailgun[46]	✓	✓	✓	✓	-
PostageAPP[47]	-	X[5]	-	-	-

[1] GET method has used for resource deletion/ moving

[2] POST method has used for resource trashing

[3] POST method has used for resource moving/copying

[4] POST method has used for resource deletion

[5] POST method has used to get resource/ list resources

4. COMPARATIVE ANAYLSIS OF API FUNCTIONS

Depending on the study on APIs in previous chapter, we have identified common functions of a generic email system which is suited for the majority of the day-to-day email activities. In this chapter, we would analyse each of these functionalities and their implementation. With that knowledge, we would design our email system after choosing the most suitable features and their relevant resource representations.

4.1. Common functions of a generic email system

4.1.1. Login into email system

Considering the systems studied in the previous chapter, most of the APIs either used HTTP basic authentication or OAuth2 authentication mechanism. Postmark and EmailYak has used different methods, respectively Token based and Query Authentication. In Postmark the client has to use the header 'X-Postmark-Server-Token' and if the header is missing or wrong, the API would respond with an HTTP Response 401 (Unauthorized). Postmark uses two types of authentication tokens; "Server token" and "Account token". The required authentication header to be used will be specified by each reference page to the API endpoints.

Microsoft outlook REST API and Google Gmail API use Oauth2 as their authentication and authorization mechanism. Google has implemented Oauth2 mechanism to comply with "OpenID Connect" specification to provide authentication. For Outlook REST API, the applications should first register themselves with the "Azure Active Directory" before using the OAuth2 protocol with "Authorization Code Grant Flow". Both of the APIs uses 'scopes' extensively to specify the authorization granted to application. The 'scope' specifies the resources that are accessible using the granted permissions.

The most popular mechanism used by the studied APIs is the "HTTP basic authentication" mechanism. Basic authentication is simple to implement, but limited in functionality. It is a must to use a secure channel to transport data between the client and the server if this method is used.

4.1.2. Listing email directories available in the account

Even though this is not a common feature of studied APIs, Context.IO, RESTMAIL and Outlook API have developed this functionality into their email systems. From a REST architectural perspective, here we are accessing a resource which would result in a representation of that resource. In the case of Context.IO Lite API, they have identified ‘folders’ as the resource and a ‘GET’ request on folders resource would result in listing all the folders under a particular mail account. For example:

```
 GET https://api.context.io/lite/users/id/email_accounts/label/folders
```

In RESTMAIL, this resource has been identified as the first ‘/’ of the URL and thus omitting the resource shown in the URL. For example:

```
GET   http://localhost:3000/
```

In the Outlook API, this functionality has been identified as “Get folders” and it supports GET HTTP method. The resulting resource would contain ID’s of each folder in the user’s mailbox. For example:

```
         GET https://outlook.office365.com/api/v1.0/me/folders
```

4.1.3. Listing mailbox content

Listing particular mailbox content is a common functionality offered by the studied APIs. Some implementations have considered messages to be a sub resource of a folder while others specifically construct the URL to denote the message collection as a resource. APIs including Outlook, Sendinc, Postmark, Context.IO 2.0 API, Mailgun and PostageApp has used “messages” resource to denote the collection of emails in a particular folder. For example:

```
GET https://outlook.office365.com/api/v1.0/me/messages
GET domains/<domain>/messages
GET https://api.context.io/2.0/accounts/id/messages/message_id/folders
GET https://api.postageapp.com/v.1.0/get_messages.json
```

Gmail has identified a resource called threads to denote conversations and thus have introduced the “thread” resource which could be used to get a list of threads.

```
GET https://www.googleapis.com/gmail/v1/users/userId/threads
```

For the Zimbra API the email collection is returned when a particular folder has been accessed using the GET method. This is the same URL template used in RESTMAIL. The folders could be the default set of folders or any user defined folder. For example:

```
GET http://localhost:7070/home/john.doe/inbox?fmt=xml
GET {mailbox_name}/{box}/[{path}]/
```

The Email Yak API has specified a special URL which could be used to list the emails. However, in every API has used ‘GET’ HTTP method for obtaining a representation of the email messages.

4.1.4. Renaming mailboxes

Even though the IMAP commands supports renaming of a folder in the mailbox, only two of the APIs from the study has implemented this feature. Outlook API has the ability to rename the folder name using the PATCH HTTP method. For example:

```
PATCH https://outlook.office365.com/api/v1.0/me/folders/{folder_id}
```

The folder ID could be a well-known folder name such as Inbox or a user given ID. The body of the PATCH request passed in as a JSON formatted object. For example:

```
{
  "DisplayName": "Business"
}
```

Since Google has used “Labels” to denote the folders, to rename the folder name, we should change the Label name. Gmail supports both ‘PUT’ and ‘PATCH’ HTTP methods for this purpose. For example:

```
PATCH https://www.googleapis.com/gmail/v1/users/userId/labels/id

PUT https://www.googleapis.com/gmail/v1/users/userId/labels/id
```

The PUT method would require user sending the request with the new name and a few other settings, the PATCH method supplies the relevant potion of the Label resource. In both cases, the server responds with the new version of the resultant Label resource.

4.1.5. Deleting mailboxes

Mailbox deletion is supported by three of the studied APIs including Outlook API, Gmail API and RESTMAIL. Since Gmail API uses 'Labels' to denote the folder, deleting the label would remove the label from all the messages tagged under that label. Here the messages won't get affected by this action. For example:

```
DELETE https://www.googleapis.com/gmail/v1/users/userId/labels/id
```

However, Outlook API and RESTMAIL has treated messages and sub folders as the content under the folder in subject and a deletion of parent folder would result in a deletion of its content too. For example:

```
DELETE https://outlook.office365.com/api/v1.0/me/folders/{folder_id}
DELETE /{mailbox_name}/{box}/[{path}]/
```

4.1.6. Display mail headers

Most of the APIs support displaying of the mail header for a particular email. While some of the APIs embedded header information in the message itself, other APIs has designed the header retrieval via special URL's or by specifying parameters to obtain the header. For example, the following has used descriptive URL in retrieving headers:

```
GET
https://api.context.io/lite/users/id/email_acc/label/folders/fol
der/messages/m_id/headers

GET
https://api.postmarkapp.com/messages/inbound/:messageid/details
```

Outlook Mail API and Email Yak have used parameters to request the header only. The Outlook API uses OData in Accept-header to specify the parameters while Email Yak specifies it as a GET variable. Gmail and Mailgun would return the full headers within the message body, when requests for a particular message.

4.1.7. Mail retrieval

Email display is a must have function for all the APIs and email systems to support mail receive and viewing functionality. Two of the common features of all the APIs are the use of GET HTTP method for message retrieval and the use of a "message id" to denote the message to be downloaded. The following URL structures have been used for message retrieval.

```
GET https://www.googleapis.com/gmail/v1/users/userId/messages/id
GET https://outlook.office365.com/api/v1.0/me/messages/{message_id}
GET http://localhost:7070/home/john.doe/?id=288
GET https://rest.sendinc.com/message/{X-Sendinc-Message-Id}.json
GET https://api.postmarkapp.com/messages/inbound/:messageid/details
GET
https://api.emailyak.com/v1/private_api_key/json/get/email/?EmailID=Em
ailID
GET https://api.mailgun.net/v3/domains/mydomain.com/messages/messageid
```

The response for a successful message request would result in a response generally containing the message id, email headers, subject, body parts, email size, mime types etc. In Context.IO API, the mail content has been separated as sub resources of the message resource. Thus only the mail body would be delivered. Most of the APIs follows MIME RFC2822[14] format for correctly denoting email body fields.

4.1.8. Deleting Email messages

Email deletion is supported by four of the studied APIs. Gmail, Outlook, Email Yak and Mailgun has provided the functionality by sending DELETE HTTP method request to a URL specifying the message ID of the mail to be deleted. It is interesting to note that Gmail and Outlook APIs would respond for a successful deletion with HTTP status code '204' which stands for "No Content" and would not provide any other feedback. For an example of a deletion request in Gmail API, the following format is used:

```
DELETE https://www.googleapis.com/gmail/v1/users/userId/messages/id
```

4.1.9. Retrieval of attachments

Attachment retrieval is another common function supported by the studied APIs. APIs such as Google, Outlook, Sendinc, Zimbra and Context.IO has designed their APIs to be able to download attachments as a sub resource based on their attachment ID, under the particular message. For example the URL structure of Gmail is as follows.

```
GET
https://www.googleapis.com/gmail/v1/users/userId/messages/messageId/at
tachments/id
```

However Email Yak and Mailgun APIs have designed their API to return URL's of the attachment in the message body when requested in the email. The Email Yak does not specify the details of the attachment; however Mailgun specifies the size and the content type of the attachment. The Postmark API would return the attachment with the message Body.

4.1.10. Email flag handling

Email flagging is an important concept which helps in manipulating and managing email messages. IMAP supports flags such as "Seen", "Answered", "Flagged", "Deleted", "Draft" and "Recent". However, only two of the APIs studied handles email flagging. Gmail has designed their system to support the label concept which works as the flag system. To update the flags, a Gmail API either uses PUT or PATCH HTTP methods on the specified message or thread label. Apart from common flags Gmail supports flags such as "Inbox", "Spam", "Trash", "Important", "Sent" and a set of special categories such as "personal", "social", "promotions" etc. Context.IO handles flags as a sub resource of the message resource. However the API only allows retrieving of the flag and does not support to update it.

```
GET
https://api.context.io/lite/users/id/email_acc/label/folders/folder/me
ssages/m_id/flags
```

The Context.IO API provides a way of marking messages as read or unread. The following URL could be requested by either POST or DELETE methods to set email message read or unread respectively.

```
POST
https://api.context.io/lite/users/id/emailacc/label/folders/folder/mes
sages/message_id/read
```

4.1.11. Copy/Move directories/ emails within directories

Email or folder copying/moving is only facilitated in Outlook REST API. The IMAP standard[16] has specified “rename” for folder name changes and “Copy” for email copying. Email moving is supported by an extension[48] to the IMAPv4. The Outlook REST API has similar URL structure for copy/move operations. The following URL is used for moving a message to a destined folder.

```
POST
https://outlook.office365.com/api/v1.0/me/messages/{message_id}/move
```

Here the POST request body contains the “DestinationId” parameter which has the destination folder ID. Similarly the moving and copying of message folders could be performed using POST method and specifying the destination in the request body. It is important to note that the URL itself contains the action which is going to be performed. Following is an example request with the request body for folder copying.

```
POST
https://outlook.office365.com/api/v1.0/me/folders/{folder_id}/copy
{  "DestinationId": "inbox" }
```

4.1.12. Email Searching and filtering

Email searching is another functionality supported by IMAP standard. Few of the studied APIs supports email searching functionality. Gmail API support query parameters in-order to perform searching and filtering of email messages. API users could filter messages by properties such as the sender, date, or label. Following is an example of a query which retrieves all messages sent by the user between Jan 1, 2014 and Jan 30, 2014:

```
GET    https://www.googleapis.com/gmail/v1/users/me/messages?q="in:sent
after:2014/01/01 before:2014/01/30"
```

The Outlook API supports email filtering and searching by the use of OData [] query parameters. The users could use actions such as filtering, selecting, ordering, count, etc. The following example URL would filter the messages which are in “Unread” state.

```
GET   https://outlook.office365.com/api/v1.0/me/messages?$filter=IsRead
eq false
```

4.1.13. Email Sending

Email Sending is another important functionality of an email system. Apart from Context.IO API others have provided mail sending functionality using different methodologies. In Gmail API, the message to be sent should be encoded in base64 and then would be included in the request body with the parameter raw. Users could either send mail directly or could save it as a draft message before sending. The following requests are performed for the mail sending functionality in Gmail API. The Gmail API has imposed a maximum file size limit of 35MB.

```
POST https://www.googleapis.com/upload/gmail/v1/users/userId/messages/send

POST https://www.googleapis.com/upload/gmail/v1/users/userId/drafts/send
```

In the Outlook REST API, the email sending could be done directly or could send an email already in the draft folder. The sender should create a JSON "Message" object as specified by the API and then make a POST request to the following URL

```
POST https://outlook.office365.com/api/v1.0/me/sendmail
```

Users could choose to save the email message in send folder by setting parameters in the request body. For a draft message which is already in the email box as a message, a POST request would be performed for the following URL.

```
POST https://outlook.office365.com/api/v1.0/me/messages/{message_id}/send
```

Outlook API users could also create a draft message before sending the mail. The API also facilitates message forwarding, reply and reply to all functionalities via special URL's. In all the cases, the API requires to use POST method. A similar method to Outlook API is followed by Postmark, Email Yak, Mailgun and PostageApp APIs for email sending. The email content is formatted as a JSON message with parameters specified by the API and then the content would be posted into a specified URL. The Gmail API requires the attachment content to be base64 encoded with the email body; other APIs specify the attachment array in JSON message separately. The attachment array generally has the content type property and the attachment content would be base64 encoded before embedding to the message.

4.2. Summary of Functions

Table 4-1 : Functionality Analysis of Vendor specific APIs

Function	Gmail API	Outlook API	Zimbra API	Sendinc API	Postmark API	Context.IO API	Email Yak API	Mailgun API	PostageApp API
Login	Oauth2	Oauth2	Basic HTTP	Basic HTTP	Token in Auth Header	Oauth2	Query Auth	Basic HTTP	Basic HTTP
List directories	GET request for Labels	GET request for folders	x	x	x	GET request for folders	x	x	x
List emails	GET request for messages	GET request for messages	GET Request for folder	x	GET request for messages	GET request for messages	GET request for messages	GET request for messages	POST request for messages
Create directories	POST request on labels	POST request to create child folders	x	x	x	PUT/POST request on threads	x	x	x
Rename directories	PUT or PATCH the label name	PATCH the directory name	x	x	x	PUT and POST to update message folder	x	x	x
Delete directories	DELETE request on a label	DELETE request on a directory	x	x	x	POST to remove folder affiliation from messages	x	x	x
Display Headers	GET request on	GET request on	GET request on	GET request on a	GET request on a	GET request on a header	GET request on a message	GET request on a message	x

	a message	a message	a message	message	message				
Display mailbody	GET request on a message	GET request on a message	GET request on a message	GET request on a message	GET request on a message	GET request on a message body	GET request on a message	GET request on a message	POST request on a message
Delete mail	DELETE request on message ID	DELETE request on message ID	x	x	x	DELETE request on message ID	POST request on message id	DELETE request on message id	x
Get attachments	GET request on attachment ID	GET request on attachment ID	GET request on attachment ID	GET request on attachment ID	GET request on a message	GET request on attachment ID	GET request on a message(gives URL for attachment)	GET request on a message(gives URL for attachment)	x
Flag Handling	PUT or PATCH the label name	x	x	x	x	GET request to check flags and POST to update	x	x	x
Copy/Move folders	PUT or PATCH the label name	POST folder ID to copy and move methods	x	x	x	POST to message ID with necessary parameters	x	x	x
Search & filter	GET query parameters	GET query parameters	GET query parameters	x	GET query parameters	GET request on folder sub resource in message resource	x	x	x
Send email	POST messages to send method	POST messages to sendmail method	x	POST to message resource	POST to email resource	x	POST to send/email	POST to messages resource	POST to send_message.json

5. THE SYSTEM DESIGN

In this chapter, we are presenting our design considerations for a REST API for email system. The design is based on the comparative study of APIs done in chapter 04 and the REST architectural constraints and other technical concerns discussed in chapter 03.

5.1. Architecture

Considering the APIs and the past work on the REST based email systems, we have identified two major branches of designs, based on the backend they have used. While backend is not visible to the clients who use the API, we believe it has a strong impact on the feature set that could be offered by the system. The Email systems proposed in [37], the author specifies a design which would provide an HTTP interface to an existing email store. While email sending functionality is not specified, many other features could be implemented with the API. The design specified in [40] offers fully functional email system with its own backend. In this approach, we have to implement the functionalities such as mail routing, notification and email storing. Noticeably in this architectural model, we could achieve email pull model rather than the traditional push model where the server needs to push the email to the correct servers. However, unless the system implements capabilities to communicate with traditional email protocols, integrating the new mail system to the existing environment would be a challenging task. Figure 5-1 depicts a generic architectural diagram of fully RESTful email system.

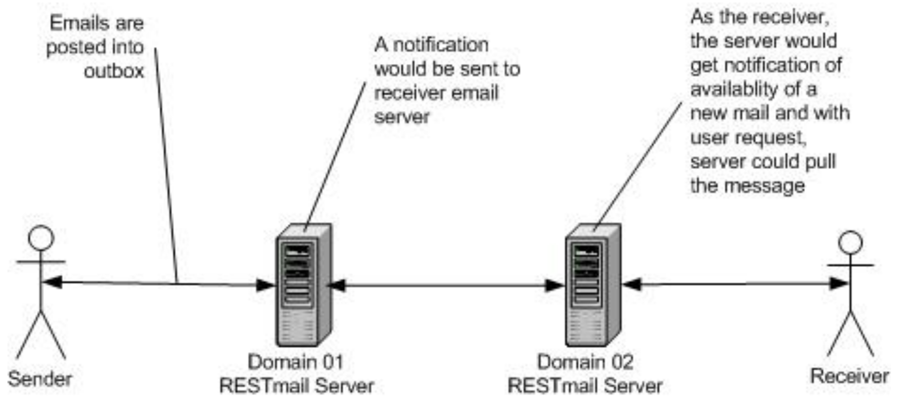

Emails are posted into outbox
A notification would be sent to receiver email server
As the receiver, the server would get notification of availablity of a new mail and with user request, server could pull the message
Sender
Domain 01 RESTmail Server
Domain 02 RESTmail Server
Receiver

Figure 5-1: Fully REST based email system

The approach taken by [38] addresses this problem by using a backend which supports traditional protocols and has a database which could be used to implement the RESTful API. Similar approaches have been taken by some of the APIs and vendor systems which were studied in chapter 4. For example, both Gmail and Yahoo support the traditional protocol stack alongside with their new REST API and the webmail interface. This architectural design allows seamless integration with the existing technology and does not require to completely abandoning the already existing and well established email infrastructure. Another important factor on using this model is that the mail receiving functionality has been handled by the underlying protocols rather than the HTTP itself or the API. However API such as [43] has implemented inbound webhooks which handles the incoming mails which are posted to special URL.

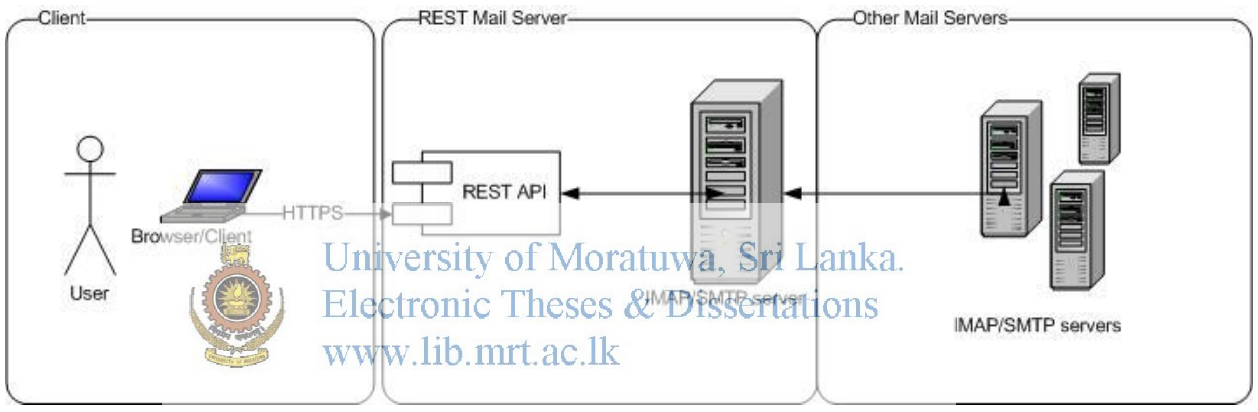


Figure 5-2: Hybrid REST mail system

Even though there are proposed email servers which follows the first architectural model which was discussed, the industry has adopted the second model for their implementations. It can be supported by the fact that it co-exist with the existing email infrastructure while providing the most of the benefits of having a RESTful interface to the email store. Noticeably the REST API could be reused for any standard email servers and thus would make it easy to be integrated and deployed. Due to the above facts, we have designed our system to follow the second architectural model where the REST interface connects to a traditional email store/ server and utilize the capabilities provided by email server to communicate with other traditional email servers.

5.2. HTTP Methods

Accurate use of HTTTP method is important for providing uniformity over the resources in our email system. Even though the HTTP specification[26] has defined nine HTTP request methods, according to the study, only GET, POST, PUT and DELETE are frequently used by the systems. As we have shown in table 3-12, some of the email systems and APIs who has confined their HTTP request methods to GET and POST methods have constructed their API URI's to indicate the action by adding verbs. This would limit the usability and clarity. The other APIs have used HTTP request methods appropriately. However, it is important to discuss about the different usages of POST, PUT and PATCH. Considering the systems which use both POST and PUT methods, both Gmail and Mailgun has used POST for new resource creation and PUT for a complete update of an existing resource. This is in-line with the HTTP specification where PUT is used when the client has knowledge on resultant URL and POST is used when the server is in charge of designating the URL for the new resource. However Postmark has used both POST and PUT in different scenarios which make it hard to map the action to the HTTP request method as defined in the specification.

In the case of the PUT and PATCH usage, only Gmail API supports both of the request methods and Outlook API only supports PATCH favouring over PUT. Implementing PATCH method support for JSON object, conferring to the PATCH semantics[49] would help to reduce the amount of data which should be passed by the client than when using the PUT request method. This is due to the fact that the PUT method requires the client to send the complete resource back to the origin server; while PATCH only requires sending the parts which were changed. Considering the above factors and generic usage based on HTTP specification, we have decided to use the following HTTP request methods in our API.

Table 5-1: Proposed HTTP request methods

Method Name	Functionality
GET	Used to obtain a representation of a resource
POST	Used to create a new resource.
PATCH	Used to update, rename & modify a resource.
DELETE	Delete a resource.

5.3. Data exchange language

Resource representation data exchanging is another important factor for the API. The study revealed that the majority of the APIs are using JSON as the primary data exchange format for representing resource. Even though some of the APIs offer XML as another data exchange format based on the analysis in section 2.8, we have concluded that JSON has better performance over objects with smaller size. Considering an email system, unless the email has attachments the communication mostly consist with smaller size messages. The responses generated for application communication could be easily designed to be smaller in the size too. Therefore, in most of the scenarios, the use of JSON could be justified for its better performance and the ease of processing at client end.

5.3.1. Hypermedia Format

To adhere to REST constraint of being HATEOAS, the media format must use hypermedia formats. Since JSON does not inherently supports hypermedia formats, various formats have purposed to augment JSON to handle hypermedia content. For example, JSON-LD[50], HAL[51] and Collection JSON[52] are some of the formats which are in the process of standardization. Many other formats have also been introduced, based on the requirements of the API.

Since the hypermedia formats are not standardized, we have designed our own format to represent the resources in the REST mail system to comply with HATEOAS constraint. However, this practice may lead to tight coupling between the client and the server unless the hypermedia format wasn't standardised alongside with the API. We have followed the IANA link relation registry [53] to describe the hyperlinks whenever possible. The proposed hypermedia format is a JSON object with following mandatory parameters.

- Type : Type of the resource
- Set of parameters which are unique to each representation of resources.
- Entities Object: Collection of sub resources. Has type parameter to identify items in the entity collection.
- Links Object: Contain links to the resource itself or a starting point and if paging available the links to the previous and next page is also available in this "Links" object.
- Actions array: The actions which could be performed on the resource by using HTTP request methods. The actions array could have several action objects, starting from the

action name. The first parameter in action object is the type of action. If the action involves GET query parameters, the type would be set to “query” and if the action is performed on a resource without additional parameters, the type would be set as “resource”. The action object also has “href” and “method” properties to denote the link, the HTTP request method which should be performed to execute that action. If there are any parameters to be passed to the action, it will be set in fields array.

Code Snippet 5-1 : Proposed Hypermedia Format

```
{
    "type": (Resource Type),
    "parameter01": "(value01)",
    "parameter02": "(value02)",
    "entities": [
        {
            "type": "(Resource Type)",
            "parameter01": "(value01)",
            "link": {
                "href": (URL for an item) ",
                "rel" : "self"
            }
        }
    ],
    "link": [
        "href": "(URL for the resource)",
        "rel": "self"
    ],
    "actions": [
           "(Action name)" : {
                "type": "(query or resource)",
                "href": "(URL for the resource/query)",
                "method": "(http request method)",
                "fields": [
                {"name":"type","value":"value"}
           ]

           }
    ]
}
```

5.4. Resources

As we have discussed in the section 2.4.1, one of the first steps of designing the REST API includes the identification of resources in our proposed system. This would help us in building the functionalities and URI’s accordingly. For our system, we have identified four main resources as “mail account”, “mail directory”, “mail” and “attachments”. Apart from this, entry point to the system is identified as the base URL. The error and successful message for API request are represented by special resource type “status”. The resources and their hierarchical structure have been depicted in figure 5-3.

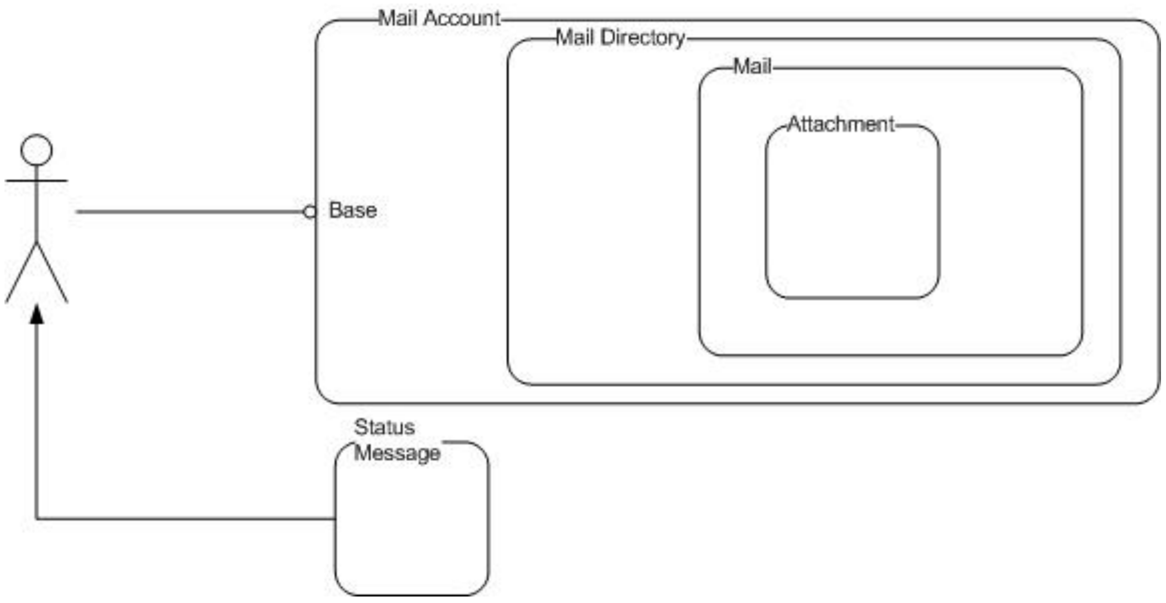


Figure 5-3: Resources

5.4.1. Base URL

Base URL is the entry point to our proposed system. The base URL is used for authentication to the system and once the client is properly authenticated, the system provides the details related to the mail account with the hypertext linking to the next resource the user could access. If properly authenticated, the API would respond with HTTP status code 200 while failure in authentication would result in 401. For example, code snippet 5-2 shows the response message for a successful authentication by a client.

Code Snippet 5-2 : Successful Authentication message

```
{
    "type" : "Status",
    "status": "success",
    "message": "User successfully authenticated",
    "link": {
        "href": "http://restmail.lk/{username}",
        "rel": "start"
    }
}
```

5.4.2. Mail account

Mail Account resource represents the mail directories for the account. When the user visits the web site with a given domain name, users would be replied with the Mail account representation. Thus the mail account resource is considered as our starting resource for the API. For example the resource representation when accessing a mail account is as follows;

Code Snippet 5-3 : Mail account resource representation.

```
{
    "type": "MailAccount",
    "name": "username",
    "entities": [
        {
            "type": "Mail Directory",
            "title": "Directory Name",
            "link": {
            "href":"https://domain/username/dir_name",
                "rel": "self"
            }
        }
],
"link": {
            "href": " https://domain/username
            "rel": "start"
    },
    "actions": [
            "(creat_dir)" : {
                "type": "resource",
                "href": " https://domain/username",
                "method": "POST",
                "fields": [
                      {"name":"type","value":"Directory"},
                      {"name":"name","value":"NewName"}
                ]
            }
    ]
}
```

5.4.3. Mail directory

Mail directory resource represents one of the mail directories and is a sub resource of the mail account. The list of emails in this selected directory has been represented as a JSON array. The following example shows the resource presentation of 'mail directory'.

Code Snippet 5-4: Mail directory resource representation.

```
{
    "type": "Directory",
    "name":"Directory Name"
    "MailCount": "Number of mail in directory",
    "PerPageMails": “Mails per page”,
    "entities": [
        {
            "type": "mail",
            "msgno": message sequence number,
            "from": "Sender <sender@domain>",
            "to": "Receiver <receiver@domain>",
            "subject": "Subject",
            "bytesize": Message size,
            "date": "Message send date",
            "flags": {
                "recent": 0,
                "unseen": 1,
                "flagged": 0,
                "answered": 0,
                "deleted": 0,
                "draft": 0
            },
            "link": {
                "href": " https://domain/username/dir/m_ID",
                "rel": "self"
            }
        }
      ],
    "link": [
      {"href":"https://domain/username/dir", "rel": "self"},
      {"href":"https://domain/username/dir?page=pid","rel": "next"},
      {"href":"https://domain/username/dir?page=pid","rel": "prev"},
      ],
    "actions": [
           "(rename_dir)" : {
                "type": "resource",
                "href": " https://domain/username/dirname",
                "method": "PATCH",
                "fields": [
                      {"name":"op","value":"replace"},
                      {"name":"path","value":"/name"},
                    {"name":"value","value":"newName"}
                ]
           },
           "(post_mail)" : {
                "type": "resource",
                "href": " https://domain/username/dirname",
                "method": "POST",
                "fields": [
                    {"name":"type","value":"mail"},
                    {"name":"m_uid","value":"<m_uid@domain>"},
                    {"name":"date","value":"unixdate"},
                    {"name":"from","value":"<from@domain>"},
                    {"name":"to","value":"<to@domain>"},
                    {"name":"cc","value":"<cc@domain>"},
                    {"name":"bcc","value":"<bcc@domain>"},
                    {"name":"subject","value":"subject"},
                    {"name":"bodyHTML","value":"html encoded"},
                    {"name":"bodyPlain","value":"plain text"},
                    {"name":"attachmentName","value":"name"},
                    {"name":" attachmentType","value":"MIME type"},
                    {"name":"        attachmentData","value":"base64
encode"}
                ]
           },
           "(delete_dir)" : {
                "type": "resource",
                "href": " https://domain/username/dirname",
                "method": "DELETE"
           }
    ]
}
```

The resource starts with its resource type name “Directory”. Inside the directory we have a collection of mail items in entity array. Each mail item contains the type of the item, its message number and other related information in the mail. Since multiple pages are expected in the response, the resource supports pagination via ‘page’ attribute. The link relation types ‘next’ and ‘prev’ has been used to denote the next and the previous pages while ‘relation’ type ‘self’ denotes the resource itself.

5.4.4. **Mail**

Mail resource represents an email within a mail directory. It is a sub resource of the mail directory. The resource type name would be set to mail in this instance. Previous and next emails in the mailbox have been linked with hypertext. For an example, following is a mail resource representation.

Code Snippet 5-5: Mail resource representation.

```
{
    "type": "mail",
    "msgno": {int},
    "m_uid": "<uid@domain>",
    "from": "sender@domain",
    "to": "receiver@domain",
    "cc": null,
    "bcc": null,
    "subject": "subject",
    "size": "byte size",
    "date": "Unix date",
    "flags": {
        "recent": 0,
        "unseen": 0,
        "flagged": 0,
        "answered": 0,
        "deleted": 0,
        "draft": 0
    },
    "bodyHTML": "HTML encoded body",
    "bodyPlain": "Plain text body.",
    "hasAttachments": "true",
    "entities": [
         {
           "type": "attachment",
            "attachmentID": "a_id",
           "attachmentName": "name",
           "attachmentType": "MIME type",
           "attachmentSize": "byte size",
              "link": {
                  "href": https://domain/user/dir/m_id/a_id",
                  "rel" : "self"
              }
           }
     ]
    "link": [
      {"href":"https://domain/username/dir/m_id ", "rel": "self"},
      {"href":"https://domain/username/dir/next_id ","rel": "next"},
      {"href":"https://domain/username/dir/prev_id","rel": "prev"},
      ]
    "actions": [
           "(delete_mail)" : {
                "type": "resource",
                "href": " https://domain/username/dir/m_id",
                "method": "DELETE"
           },
           "(update_flags)" : {
                "type": "resource",
                "href": " https://domain/username/dir/m_id",
                "method": "PATCH"
                "fields": [
                     {"name":"op","value":"replace"},
                    {"name":"path","value":"/flags/{0-5}"},
                    {"name":"flag name","value":"0 or 1"}
           }
    ]
}
```

The parameters specified in Table 5-3 are used for detail representation.

Table 5-2: Parameter list for mail resource

Parameter	Intention
Type	Resource Type
msgno	Message sequence number
m_uid	Unique mail ID
from/ to	Sender and Receiver email addresses
cc/bcc	CC and BCC email addresses
subject	Email Subject
size	Email message size in bytes

date	Date of email sent
flags	Email Flag if set 1 and if not 0
bodyHTML	HTML formatted mail body
bodyPlain	Plain text mail body
hasAttachments	True if has attachments, false if not
attachmentID	ID of the attachment
attachmentName	Name of the attachment
attachmentType	Attachment MIME type
attachmentSize	Byte size of the attachment

5.4.5. **Attachments**

The attachments have been identified as a separate resource from ‘mail’, since the size of the attachment might affect the loading of whole ‘mail’ resource. The attachment resource would contain its ID, name, MIME type, size and the base64 encoded version of the attachment data. Based on the attachment ID, the clients could access multiple attachments separately. The link array includes a link in the email message which the attachments belong to with the relation type ‘up’.

Code Snippet 5-6: Attachment resource representation

```
{
 "type": "attachment",
 "attachmentID": "a_id",
 "attachmentName": "name",
 "attachmentType": "MIME type",
 "attachmentSize": "byte size",
 "attachmentData": "base64 encoded data",
 "link": {
   {"href":"https://domain/username/dir/m_id/a_id ", "rel": "self"},
   {"href":"https://domain/username/dir/m_id/prv_id ","rel": "prev"},
   {"href":"https://domain/username/dir/m_id/nxt_id ","rel": "next"},
   {"href":"https://domain/username/dir/m_id/","rel": "up"},
 }
}
```

5.5. Functionality

After the analysis of the proposed REST email systems and commercial REST APIs we have compiled a list of functionality which could be considered as compulsory for having a functional RESTful email system. Starting from section 5.5.1, we would discuss the designing of each of those functionalities in our proposed system.

5.5.1. Login to mail system

The authentication to the mail system was designed by using basic authentication mechanism. Even though OAuth2 would have provided further functionality and support more use cases, the complexity of implementing OAuth2 functionality to the existing email servers would be an additional burden for an integration process. This would become a drawback for our proposed API. In our design, the API would require clients to authenticate when they reach the “root resource” and the credentials given here would be used to authenticate the user to the existing mail system. Since the credentials are passed in plain text format, it is required that the clients to the API and the API to mail server to use a secure communication channel. Assuming our API is exposed via the domain name “www.restmail.lk”, the following request would prompt the user to provide credentials for the authentication.

```
GET https://www.restmail.lk/
```

Otherwise for the machine to machine communication would be done by passing the credentials in the HTTP authorization header itself.

```
GET https://username:password@www.restmail.lk/
```

An unsuccessful login attempt would result in HTTP status code ‘401’ and would be responded with login failure. The logout functionality could be implemented at client side based on user agent used for authentication. Generally in browsers, the client would provide erroneous username/passwords intentionally to reset the authentication headers.

5.5.2. Getting a list of mail directory

Listing of directories under a particular mail account is important for account owners to organize emails and for easy access. This feature is implemented in [6] as labels while [45] and [7] provides folder resource to represent the folders in the mailbox. In our design, the clients would be able to retrieve a list of available mailboxes by following request.

```
GET https ://{domain}/{username}
```

This URL would be automatically available in the response after a successful authentication. The API would respond with a list of mailboxes in the mail account with HTTP status code ‘200’. In the case of [7] & [45], the ‘folders’ resource would requires additional resources to denote the hierarchical structure of the system. For example in [45], the ‘folders’ should be followed by the ‘folder name’. However, in our design, the username could be followed by the mailbox names discovered by this representation. Therefore, in our design, the URL structure would be shorter, clearer and would provide more emphasis on the hierarchical structure in mail box.

5.5.3. Creating new mail directory

Apart from the default or already existing mail directories for a mail account, the owner could create more directories for mail organization. Since we are creating a new resource in the server, the HTTP request method POST has been chosen for this. The new directory name is required to be passed through the POST request as JSON object as shown below.

```
POST https://{domain}/{username}
{ “type”: “Directory”, “name”:”new_dir_name”}
```

The “Directory” parameter is mandatory and the client could specify any string as the new directory name. Successful execution would result in a response with status code ‘201’ and a success message with the URL to the newly created directory and the mail account. The design of this functionality is similar to the design of [7] & [40] apart from the URL structure.

5.5.4. Rename a mail directory

The directory renaming functionality has been implemented in API such as [6] has [7]. The directory renaming could also be considered as ‘moving’, only if it happens within the directory. In our system we have only designed the directory renaming functionality by the following request.

```
PATCH https://{domain}/{username}/{dir_name}

{ "op": "replace", "path": "/name", "value": "new_name" }
```

Here, the URL denotes the directory, which is subjected to the renaming. We are performing a PATCH HTTP request against the resource and the request should follow the patch semantics. A successful execution of this request would result in HTTP status code ‘200’ and JSON object with the success message and the resultant URL. A failed execution would occur when the mail account has a directory with a similar name. In that situation, the system would respond with HTTP status code of ‘409’ and the error status message.

5.5.5. Delete a mail directory

The directory deletion functionality has been implemented in several APIs with DELETE HTTP request method. The following request could be performed to delete a directory in our system.

```
DELETE https://{domain}/{username}/{dir_name}
```

The URL denotes the directory which is going to be deleted and the successful execution of this would result in HTTP status code 200 and the JSON formatted success status message and in the case of URL does not exist, the system would respond with the HTTP status code 404.

5.5.6. Searching mails within a directory

Email searching is a basic requirement for mail system. However, other than exposing a controller resource, it is difficult to create a RESTful URL for the searching function. Therefore, in most of the APIs, GET query parameters have been used. Therefore, in our design, similar to the way we have set parameters for paginations; two separate query parameters could be set to filter out the mails in a directory. First parameter is the “filter”. The “filter” depends on the back-end implementation and it is recommended to implement the criteria defined by IMAP protocol. In our design, we are supporting several criteria such as “from”, “to”, “subject”, “all”. The second parameter is “string” and it is used to set the string value given by user for searching. The search results would be displayed as a directory resource with pagination parameters. However, only the matching mails would be available through the resource representation.

```
GET
https://{domain}/{username}/{dir_name}?filter={all}&string=”hello
”
```

5.5.7. Listing emails in a directory

The emails are considered to be a sub resource of the directories. Therefore, to list emails in a particular directory, the client could perform a GET request against the directory resource.

```
GET https://{domain}/{username}/{dir_name}
```

Based on the given directory name, this will return a collection of emails as a JSON array with HTTP status code 200. If the directory is empty, the API would return an empty JSON message linking back to the root resource. If the directory does not exist the API would return HTTP status code 404. As we have discussed in section 4.1.3, this functionality has been implemented by most of the systems which were studied. It is important to note that we have added pagination support for this resource due to the possible high volume of emails in a mail store. The page number is set by GET parameter “page” and the resource representation would provide hyperlinks for the previous and next page. Both [6] and [7] has used the parameters to denote the pagination information.

5.5.8. Displaying email

Displaying the email message is an important feature of an email system. As per the study on APIs email retrieval was done by performing a GET request on message ID. Following request could be used to retrieve the full message in our system.

```
GET https://{domain}/{username}/{dir_name}/{m_id}
```

In our design the message ID is set to be the message sequence number in the IMAP mail store. More details on resource representation have been discussed in section 5.4.4. The message body is formatted as a JSON object and the parameters discussed in table 5.3 could be obtained via that representation. The mail resource has been separated from mail attachment data. This was done to control the load time increase due to attachment size. The links to download the attachments are available in the mail resource itself. Unlike in [6] and [7] where the client requires to obtain the attachment ID and then construct the request URL, the method we have used here would support HATEOAS constraint. A successful request for message would result in HTTP status code ‘200’.

5.5.9. Retrieving email attachments

As we discussed in section 5.5.7, after successfully retrieving the email message, the client could work through the parameters and identify whether there are any attachment for the particular mail. The attachment ID is used to identify the attachment and thus the multiple attachments act as sub resources of the mail. The URL pointing to the attachments could be used to perform a GET request to obtain the attachment resource as shown below;

```
GET http://{domain}/{user}/{dir}/{m_id}/{attachment_id}
```

Successful retrieval of the attachment would result in a JSON object containing the information related to the attachment and the base64 encoded attachment data. The HTTP status code for successful retrieval is ‘200’ while for non-existing URL the HTTP status code has been set to 400’.

5.5.10. Posting/Creating email

The API allows clients to POST emails into directories. This functionality is available in IMAP as appending email and could use to implement many other features such as email sending, copying and moving. A special JSON object which is similar to the JSON object which would client receive when accessing the 'mail resource' should be generated prior to posting the mail. The email addresses, dates and the mail ID follow the RFC 2822[14] specification. Code snippet 5-4 shows sample request for mail creation. Here, the 'attachmentData' field should contain the base64 format of the attachment data.

Code Snippet 5-4: Email Creation in a given mail directory

```
POST https://{domain}/{user}/{mail_dir}
{
 "type": "mail",
 "msgno": {int},
 "m_uid": "<uid@domain>",
 "from": "sender@domain",
 "to": "receiver@domain",
 "cc": null,
 "bcc": null,
 "subject": "subject",
 "size": "byte size",
 "date": "unix date",
 "bodyHTML": "HTML encoded body",
 "bodyPlain": "Plain text body.",
 "hasAttachments": "true",
 "attachmentName" :"small.gif",
 "attachmentType":"application/octet-stream",
 "attachmentData":"base64"
}
```

5.5.11. Deleting email

Mail deletion has been implemented in few APIs such as [6], [7], [46] and [40]. The standard way to perform the deletion is to perform a DELETE HTTP request against the mail resource. The following is an example request;

```
DELETE https://{domain}/{username}/{dir}/{m_id}
```

As we discussed in 4.1.8, the APIs such as [6] and [7] would responds with 'no content' HTTP status code. This is the logical response after a successful deletion of resources by a REST API. However, since we are trying to achieve HATEOAS system, the 'no content' response would leave the client in a state where they no longer have any link to follow. Therefore, we have designed our system to respond to a successful email deletion with a JSON formatted success message and HTTP status code of 200.

5.5.12. Flag manipulation

The API supports email flags, including 'recent', 'unseen', 'flagged', 'answered', 'deleted' and 'draft'. The flags could be updated by performing a PATCH HTTP request to the targeted mail resource. The PATCH semantics should be followed for the request body. The JSON pointer to the flag should be derived from the 'Mail' representation that the client may download prior to flag manipulation. For example, recent flag would have the path variable as /flag/0 since it is the first element of flags object. To set the flag, the value should be set to 1 and to clear, set the flag value to 0. For a successful flag update, the API would respond with a JSON formatted success message and HTTP status code of 200. Sending an unsupported flag would result in an error message with HTTP status code of 400.

```
PATCH https://{domain}/{user}/{mail_dir}/{mail_id}

{"op": "replace", "path": "/flags/{0-5}", "value": "0 or 1" }
```

5.5.13. Email sending

Email sending an important feature supported by our system. In general, the APIs we have studied have used POST HTTP request method to create the resource against a special URL which would indicate the API that client requires the mail to be sent. For example [7] would perform the request against a resource named 'sendmail'. Such methods would utilize the ‘action’ described by the resource rather than giving priority to the HTTP request method. However, unless we are following the email sending mechanism suggested in system such as [40], performing a POST request against the ‘Sent’ directory is the most common practice followed by the APIs.

The client could perform the POST request which was described in section 5.5.9 against the sent directory and the system would automatically send the email to the receiver end. After sending the mail, the system would save a copy in the sent directory. If the client requires sending a mail which is already in ‘Draft’ directory, first, a copy of the mail should be obtained by a GET request and then POST the message into ‘Sent’ directory after formatting. DELETE request should be made against the mail message in the Draft directory when the API responds with an operation success message. Successful message sending would result in HTTP status code 200 message while erroneously formatted request would result in HTTP status code 400. The following POST request and the payload should be followed to perform ‘send’ operation.

```
POST https://{domain}/{user}/Sent
{
 "type": "mail",
 "msgno": {int},
 "m_uid": "<uid@domain>",
 "from": "sender@domain",
 "to": "receiver@domain",
 "cc": null,
 "bcc": null,
 "subject": "subject",
 "size": "byte size",
 "date": "unix date",
 "bodyHTML": "HTML encoded body",
 "bodyPlain": "Plain text body.",
 "hasAttachments": "true",
 "attachmentName" :"small.gif",
 "attachmentType":"application/octet-stream",
 "attachmentData":"base64"}
```

5.5.14. Moving/Copying email

Moving and copying email messages within directories are only supported in API [6] and [7]. Since [6] uses labels to identify the directories, manipulating the labels would have the same effect of performing a move operation. However in [7], as discussed in section 4.1.11, the authors have construct the URL by adding the move verb and thus limiting the unified interface which considered as a constraint in RESTful systems. Therefore, to protect the REST constraints, we have designed our API to perform a copy or move operations by obtaining a copy of the original mail and creating a new mail message on desired location. Even though this method requires two operations, the server does not have to keep the status of the pervious request to track the client. However, if the mail account is used by multiple clients, this design may run into a race condition situation.

The client could first obtain the original mail which needs to be copied or moved by the same way discussed in section 5.5.7 and 5.5.8. Then the retrieved email content has to be re-formatted and then append to the desired mail directory as shown in section 5.5.9.

If the client performing a move operation, the client has to delete the original mail as a third step. Mail deleting would be discussed in section 5.5.10. A successful message creation would result in a JSON formatted success message and HTTP status code 201.

5.6. Summary of Design

Comparing the common functionalities identified in the section 4.1 with the functionalities we have designed for our API, apart from the searching and filtering functionality, all the other functionalities were designed into our system. With this design, our goal of designing an API which gives priority to REST constraints and open standards have been achieved and the API could be used in already established traditional email infrastructure which gives it added advantage.

In functionality perspective the move/ copy function has been changed significantly to avoid having verbs in the URL. This could be argued by defining such verbs as a “controller resources” in the API. However, in our design, the move and copy operations could be performed without sending additional parameters and could perform in a stateless manner.

With regards to our design, we have proposed substantial deviations from other APIs for URI structures by following hierarchical structure and for data exchange formats by adding hypertext links extensively. A hierarchical URI structure may lead to tight coupling of client and server. However, since the representations are linked with hypertext, the HATEOAS constraint if fulfilled and thus the hypertext could be used to traverse through the system without following the hierarchical structure.

6. IMPLEMENTATION

In this dissertation, our goal is to study existing APIs and come up with a standard specification which could standardize as an open protocol. To achieve this goal, we have implemented server side scaffolding which follows the design metrics discussed in chapter 5. The API could be tested by generic client side tools. Thus, we have not implemented any specific client side tools for testing.

As per the discussion in section 5.1, we have followed the architectural model where the API would connect to existing email store which supports IMAP and SMTP protocols. As shown in figure 5-2, the REST interface to the email store could be either hosted on a different system or the same system where the email store is hosted if there is a web server which supports our API implementation. In this chapter, we would discuss the use of technologies to implement the model API which was designed based on the chapter 5 design requirements.

6.1. Architecture

As we have discussed in section 5.1, we have chosen to base our implementation of the REST API on an architectural model depicted in figure 5-2. In our case, the communication channel between the REST API and the traditional email store was chosen to be IMAP/ SMTP protocols. The diagram 6-1 depicts the architectural diagram which was followed during the implementation.

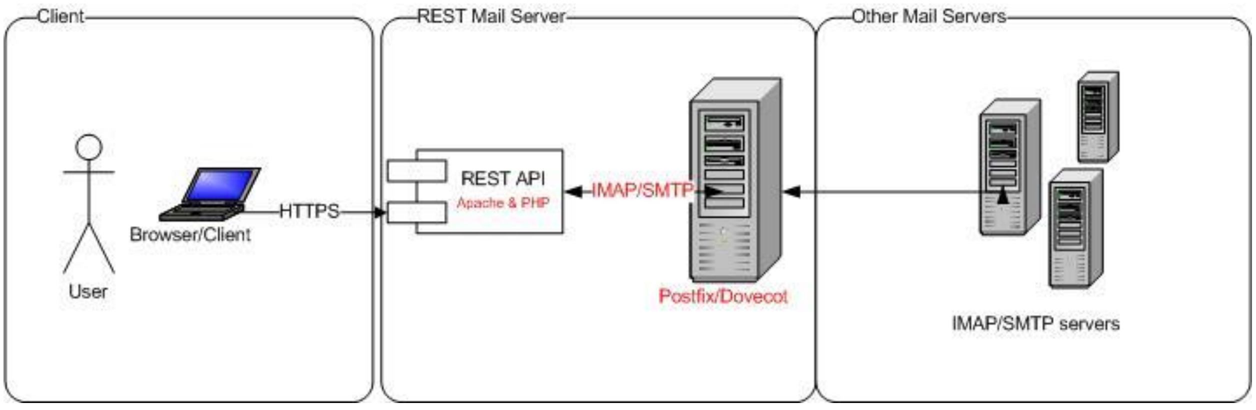


Figure 6-1: Proposed Architecture

6.2. Development environment

The API was developed using PHP 5.5.9 language and was hosted on Apache 2.4.7 server. PHP was chosen due to its popularity and simplicity in programming. This would help users quickly adapt or understand the implementation details to come up with their own implementation. Apache is the most commonly used web server and hence most of the users would be able to seamlessly integrate our API into their existing systems. The email store which was used for the development was a dovecot version 1.2.15 and postfix version 2.7.

6.3. Library usage

The libraries are a collection of resources used by computer programs to develop software. These libraries contain pre-written code, classes, methods and documentation. Using a library would help us to reduce the development time and reduce the erroneous code. In our implementation, we have used “PHP:IMAP”[54] and “Swift Mailer”[55] libraries to facilitate us in communicating with the IMAP and SMTP servers which was mentioned in previous section.

6.3.1. PHP: IMAP

The IMAP library provides a set of functions which could use to interact with IMAP protocol, as well as the NNTP, POP3. However, in our implementation, we have not provided support for other protocols than IMAP. The IMAP library was used in implementing all the functionality related to accessing the IMAP server. Some of the functions in the library have provided us one to one matching for our functions, requiring less program effort. Table 6-1 list down the PHP: IMAP functions used by our API.

Table 6-1: List of PHP:IMAP functions used

Function	Functionality
imap_open	Open an IMAP stream to a mailbox
imap_createmailbox	Create a new mailbox
imap_fetch_overview	Read an overview of the information in the headers
imap_renamemailbox	Rename an old mailbox to new mailbox
imap_deletemailbox	Delete a mailbox
imap_append	Append a string message to a specified mailbox
imap_fetchstructure	Read the structure of a particular message
imap_body	Read the message body
imap_list	Read the list of mailboxes
imap_headers	Returns headers for all messages in a mailbox
imap_uid	This function returns the UID for the given message

imap_clearflag_full	Clears flags on messages
imap_setflag_full	Sets flags on messages
imap_expunge	Delete all messages marked for deletion
imap_base64	Decode BASE64 encoded text
imap_qprint	Convert a quoted-printable string to an 8 bit string

Even though features such as copying and moving emails are directly supported by the PHP: IMAP library, we have given priority to protecting REST constraints over the functionality. Therefore, some of the functions in our implementation require additional work by the client side software to achieve the same effect.

6.3.2. Swift Mailer

Swift mailer is a library developed for the purpose of sending email from PHP 5 applications. Even though it has begun as a one-class project in 2005 by Chris Corbyn, the library now has developed into fully fledged email sending library by providing more functionality than the inbuilt mail() function of PHP. Swift mailer is now maintained by Fabien Potencier. Another alternative library which was considered for our use is PHPMailer. However considering that Swift mailer is licensed under MIT license[56] and PHPMailer has licenses under LGPL 2.1[57], it was decided to use Swift mailer avoid licensing issues.

6.4. Apache Configurations

In order to maintain URL structure for the API and to provide secure communication channel, the ‘mod_rewrite’ and ‘mod_ssl’ models were used. For the testing purposes, the HTTPS connection was supported using a ‘self-signed’ certificate with key length of 2048. All the communication to the API and from the API to the mail server is mandatory go through a secure connection to protect credentials which are passed through during the communication. Since the URL structure is important in denoting the resource hierarchy and to provide clear URL’s the following mod_rewrite parameters were set in ‘.htaccess’ file in the apache ‘DocumentRoot’ directory.

```
RewriteEngine on
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^(.*)$  index.php/$1
```

6.5. API Configurations

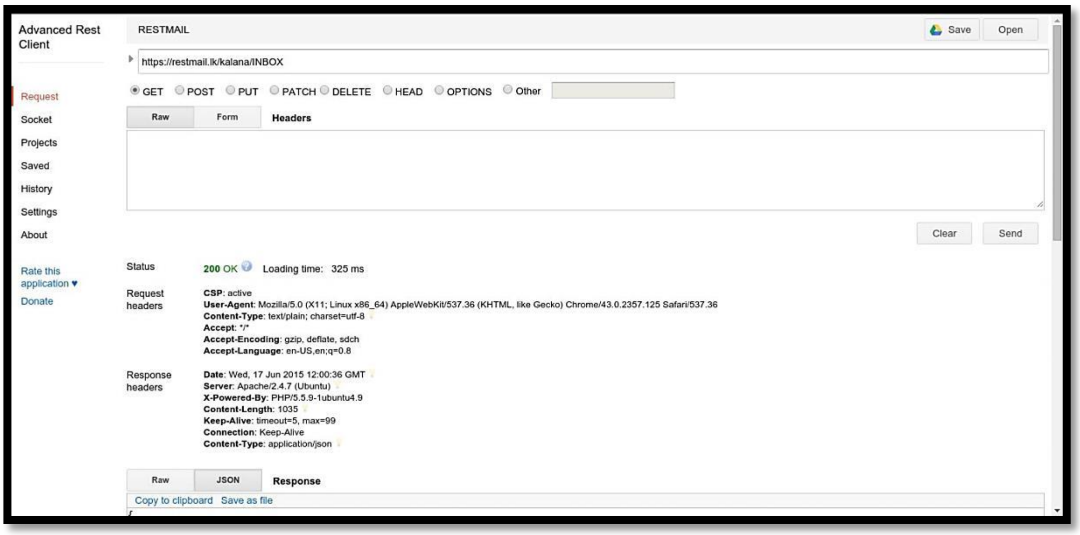
The API was designed for easy deployment by any users who can fulfil the basic requirements of having a PHP supported Apache web server with the modules and libraries mentioned in section 6.3 and 6.4. The mandatory configuration fields for the API can be configured in the simple text file located in the project directory. The following parameters have to be configured in 'restmail. conf' file before using the API

- IMAP mail server / port
- IMAP server transport security
- SMTP server / port
- SMTP server transport security
- Domain name of the API hosted server.
- Number of mails per page.

6.6. User agent/ Client

As we have pointed out in section 5.2, the API was designed to support only four out of nine HTTP request methods. Even though this may restrict our capabilities to perform actions against the resources, HTML 4[58] & HTML 5[59] specifies support for only GET and POST request methods. Thus making the browsers restricted to those two methods. However, the java script object 'XMLHttpRequest'[60] could be used to construct application which could generate request methods other than GET and POST. Therefore, using client side JavaScript libraries, it is possible to access the API via generic web browsers which supports JavaScript.

Since implementation of the client is out of the scope of our work, for the testing purposes of the API, we have used a browser extension named "Advance REST client" for Google Chrome browser and CURL software. In both of these tools, we can set the HTTP request method to be used and the relevant payload where it is necessary. The browser extension would request the password for the basic authentication via browser and would use the browser session thereafter, it would represent a typical browser based client for the API. For curl software, we have to manually set every parameter, including the credentials for each request, which might resemble machine to machine communication. Figure 6-2 and 6-3 shows a sample request performed using each of the above client software against our API.



Advanced Rest Client

RESTMAIL

https://restmail.lk/kalana/INBOX

[✓ GET] POST PUT PATCH DELETE HEAD OPTIONS Other

Status: 200 OK — Loading time: 325 ms

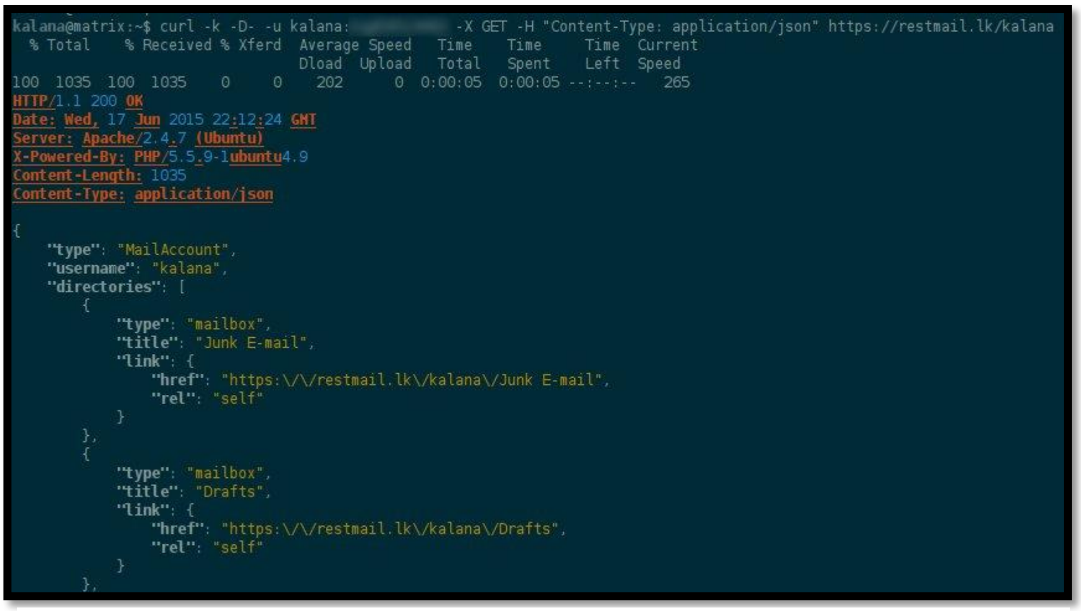
Request headers:

```
CSP: active
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/43.0.2357.125 Safari/537.36
Content-Type: text/plain; charset=utf-8
Accept: */*
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
```

Response headers:

```
Date: Wed, 17 Jun 2015 12:00:36 GMT
Server: Apache/2.4.7 (Ubuntu)
X-Powered-By: PHP/5.5.9-1ubuntu4.9
Content-Length: 1035
Keep-Alive: timeout=5, max=99
Connection: Keep-Alive
Content-Type: application/json
```

Figure 6-2: Google Chrome – REST client extension



```
kalana@matrix:~$ curl -k -D- -u kalana:[redacted] -X GET -H "Content-Type: application/json" https://restmail.lk/kalana
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  1035  100  1035    0     0    202      0  0:00:05  0:00:05 --:--:--   265
HTTP/1.1 200 OK
Date: Wed, 17 Jun 2015 22:12:24 GMT
Server: Apache/2.4.7 (Ubuntu)
X-Powered-By: PHP/5.5.9-1ubuntu4.9
Content-Length: 1035
Content-Type: application/json

{
    "type": "MailAccount",
    "username": "kalana",
    "directories": [
        {
            "type": "mailbox",
            "title": "Junk E-mail",
            "link": {
                "href": "https:\/\/restmail.lk\/kalana\/Junk E-mail",
                "rel": "self"
            }
        },
        {
            "type": "mailbox",
            "title": "Drafts",
            "link": {
                "href": "https:\/\/restmail.lk\/kalana\/Drafts",
                "rel": "self"
            }
        },
```

Figure 6-3: CURL command line tool

6.7. Testing of the API

Testing of the API was conducted in different environments to assess its correctness and performance in each environment. Although the correctness is an important factor to validate our proposed API, the performance would depend on external factors which are out of the scope of API design. However, we have tested our API against the following three scenarios:

1. Client, web server and the mail server in same computer. (Figure 6-6)
2. Client, web server and the mail server in same local area network (Figure 6-5)
3. The client connects via the Internet to the web server and mail server which is in same local area network. (Figure 6-4)

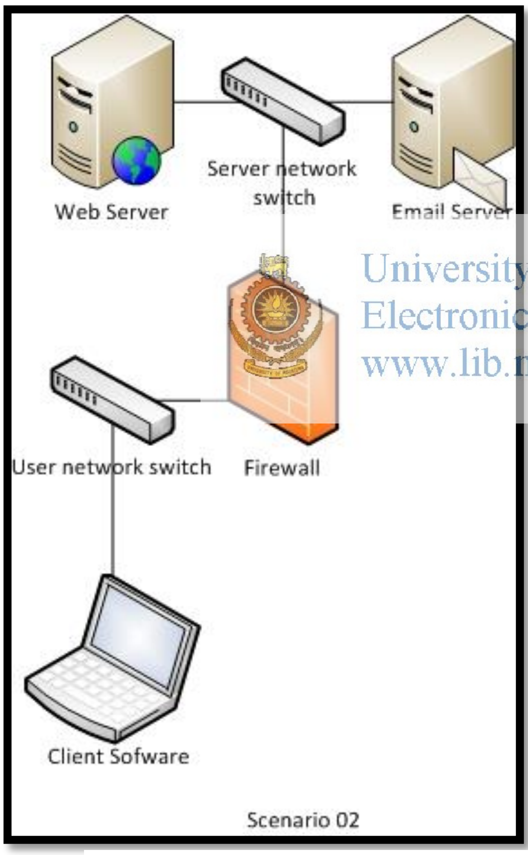


Figure 6-5 Test Scenario 02

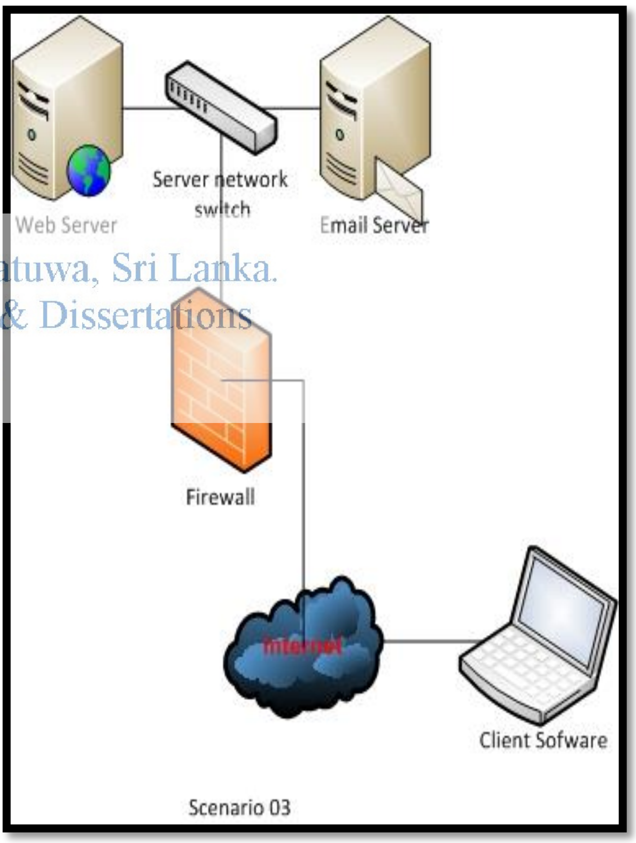


Figure 6-4: Test Scenario 03

Even though the functionalities worked as expected, the time consumed for a request to complete varied based on the scenario. Table 6-3 contains time consumption for functionalities complete in above three scenarios. Time was calculated using the ‘curl’ and ‘time’ Linux utilities. Same parameters and emails were used for all test cases. The results show a significant increase of time for functionality to complete in scenario 3. This may be due to the number of intermediate nodes the request has to pass when it connects through the Internet. Intermittent network issues may have affected the scenario 2 results because some of the results show more time consumption than scenario 3.

Table 6-3: Performance analysis

Functionality	Scenario 1 (ms)	Scenario 2 (ms)	Scenario 3 (ms)
1. Login to system	108	134.33	2202.33
2. Getting list of mail directories	176.67	219	6930.33
3. Creating a new mail directory	170.67	200.33	6262
4. Rename a directory	165.33	223.33	6488
5. Deleting mail directory	170	228.33	5086.33
6. Email Searching	178.4	236.2	5561
7. Listing email in a directory	181.33	244.67	6299.67
8. Displaying email	175.67	241.67	5507
9. Displaying attachment	175	237.33	6322.33
10. Post mail to a directory	203	243	7089.67
11. Deleting email	257.67	354.67	6929.67
12. Flag manipulation	235.33	357.33	7531.67
13. Email Sending	2654	5524	21313.67

6.7.3. Observations

- The results show a significant increase of time for functionality to complete with scenario 3. This may be due to the number of intermediate nodes the request has to pass, when the system connects through the Internet.
- The time taken for email sending operation significantly increase for all scenarios. Since the same content which was used in the function 9 was used in for this, the additional time taken should be the SMTP server processing time for the email sending and its reply.
- Email deletion had a slight increase in processing time compared to other operations in each scenario. This may be due to the fact that the API performs deletion as two separate processes; marking the message for deletion and then deleting the mail by expunging the mailbox.
- Email move and copy operation were not tested as it is the accumulated time for email retrieval, posting and deleting an email.

7. CONCLUSION & FUTURE WORK

In this dissertation, we have designed and implemented a RESTful API for email stores which supports IMAP protocol. The design was based on the comparative study conducted on commercial REST mail APIs in the market and analysis of previous attempts on building REST email systems. Relative technologies used with the above systems were also studied to identify the appropriate supporting technologies for the system. In our design we have given priority to protecting the REST architectural concepts while trying to maintain the usability of the API. It would aid us to use our design as a guideline for standardizing REST email API.

In this design we have identified a required set of resources which could represent a mail account and has defined a URL structure, which could use to access the resources. Moreover, we have identified a subset of HTTP request methods which could be used in identifying resources to achieve common functional requirements for a mail system. While designing, we have striven to maintain a resource representation format which would safeguard HATEOAS constraint. Therefore a client would be able to interact with the email system as a hypermedia system after entering to our API through the root resource. Our model implementation covers the functionalities which are proposed by our design. Since we are following an architectural model where our API could be directly used with an existing email infrastructure, it would easier for community test and improve the concept. With this design, we hope the community would be able to standardize the RESTful API design requirements rather than going for vendor specific designs and thereby providing more uniform, standardized interfaces where the client side application would be easier and would help to decouple the client and server applications and allow them to grow independently.

As future work on this matter, data exchange formats and resource representation could be improved and register it as a media type by IANA registration procedures. The API could be improved by standardizing the email search and filtering functionality as we have discussed in section 5.6. It is also possible to implement other back ends for the API where emails may store is in a database and is independent of traditional email servers and resides on the web server itself. Another area of improvement is adding mail receiving functionality to the API via special URL, where email receiving could be performed on HTTP.

REFERENCES

[1] T. Van Vleck, “Electronic mail and text messaging in CTSS, 1965-1973,” *Ann. Hist. Comput. IEEE*, vol. 34, no. 1, pp. 4–6, 2012.

[2] J. Postel, “Simple Mail Transfer Protocol”, RFC Editor, RFC0821, Aug. 1982.

[3] J. K. Reynolds, “Post Office Protocol,” RFC Editor, RFC0918, Oct. 1984..

[4] M. R. Crispin, “Interactive Mail Access Protocol: Version 2,” RFC Editor, RFC1064, Jul. 1988.

[5] R. T. Fielding, “Architectural styles and the design of network-based software architectures,” University of California, Irvine, 2000.

[6] Google Inc., “Gmail REST API,” *Google Developers*. [Online]. Available: https://developers.google.com/gmail/api/. [Accessed: 08-Apr-2015].

[7] Microsoft Corporation, “Outlook Mail REST API reference.” [Online]. Available: https://msdn.microsoft.com/en-us/office/office365/api/mail-rest-operations. [Accessed: 22-Apr-2015].

[8] K. Thompson and D. M. Ritchie, *UNIX Programmer’s Manual*. Bell Telephone Laboratories, 1975.

[9] A. Hunter, “PROFS for the Scientist,” *Central Computing Division*, Jan-1987. [Online]. Available: http://www.chilton-computing.org.uk/ccd/literature/ccd_newsletters/forum/p87a.htm#s1. [Accessed: 22-Apr-2015].

[10]T. V. Vleck, “The History of Electronic Mail.” [Online]. Available: http://www.multicians.org/thvv/mail-history.html. [Accessed: 22-Apr-2015].

[11]J. van Rijn, “The ultimate mobile email statistics overview,” *Emailmarketing consultant | Emailmonday*, Jun-2015.

[12] R. Gellens and J. Klensin, “Message Submission for Mail,” RFC Editor, RFC6409, Nov. 2011.

[13] J. Klensin, “Simple Mail Transfer Protocol,” RFC Editor, RFC5321, Oct. 2008.

[14] P. Resnick, “Internet Message Format,” RFC Editor, RFC5322, Oct. 2008.

[15] J. Myers and M. Rose, “Post Office Protocol - Version 3,” RFC Editor, RFC1939, May 1996.

[16] M. Crispin, “Internet Message Access Protocol - version 4rev1,” RFC Editor, RFC3501, Mar. 2003.

[17]N. Borenstein and N. Freed, *MIME (Multipurpose Internet Mail Extensions): Mechanisms for Specifying and Describing the Format of Internet Message Bodies*. IETF, 1992.

[18]S. Dustdar and W. Schreiner, “A Survey on Web Services Composition,” *Int J Web Grid Serv*, vol. 1, no. 1, pp. 1–30, Aug. 2005.

[19]E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, *Web services description language (WSDL) 1.1*. W3C, 2001.

[20]T. Bellwood, L. Clément, D. Ehnebuske, A. Hately, Y. L. Husband, and M. Hondo, “UDDI V3 Specification.” [Online]. Available: http://www.uddi.org/pubs/uddi-v3.00-published-20020719.htm. [Accessed: 22-Apr-2015].

[21]N. Mitra and Y. Lafon, “SOAP Version 1.2 Part 0: Primer (Second Edition),” W3C, W3C Recommendation, Apr. 2007.

[22]K. Lawrence, C. Kaler, A. Nadalin, R. Monzillo, and P. Hallam-Baker, “Web services security: SOAP message security 1.1 (WS-security 2004),” *OASIS OASIS Stand. Feb*, 2006.

[23]A. Arsanjani, “Service-oriented modeling and architecture,” *IBM Dev. Works*, pp. 1–15, 2004.
[24]F. Kappe, G. Pani, and F. Schnabel, “The architecture of a massively distributed hypermedia system,” *Internet Res.*, vol. 3, no. 1, 1993.
[25]R. Ekblom, “Applied Representational State Transfer", Umeå University, Department of Computing Science, 2011.
[26] R. Fielding and J. Reschke, “Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content,” RFC Editor, RFC7231, Jun. 2014.
[27] L. Dusseault and J. Snell, “PATCH Method for HTTP,” RFC Editor, RFC5789, Mar. 2010, 2010.
[28]M. Jakl, “REST Representational State Transfer,” 2008.
[29]T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, “Extensible markup language (XML),” *World Wide Web Consort. Recomm. REC-Xml-19980210 Httpwww W3 OrgTR1998REC-Xml-19980210*, p. 16, 1998.
[30]T. Bray, *The JavaScript Object Notation (JSON) Data Interchange Format*. IETF, 2014.
[31]A. Šimec and M. Magličić, “Comparison of JSON and XML data formats,” in *CECIIS - 2014*, 2014.
[32]N. Nurseitov, M. Paulson, R. Reynolds, and C. Izurieta, “Comparison of JSON and XML Data Interchange Formats: A Case Study.,” *Caine*, vol. 2009, pp. 157–162, 2009.
[33] R. Fielding and J. Reschke, “Hypertext Transfer Protocol (HTTP/1.1): Authentication,” RFC Editor, RFC7235, Jun. 2014.
[34]A.Bouchez, “How to implement RESTful authentication,” 24-May-2011. [Online]. Available: http://blog.synopse.info/post/2011/05/24/How-to-implement-RESTful-authentication. [Accessed: 07-May-2015].
[35] D. Hardt, “The OAuth 2.0 Authorization Framework,” RFC Editor, RFC6749, Oct. 2012.
[36]P. Prescod, “Reinventing Email using REST,” 11-Aug-2002. [Online]. Available: https://web.archive.org/web/20020811103430/http://www.prescod.net/rest/restmail/. [Accessed: 10-May-2015].
[37]L. Dusseault, HTTP Access to Email Stores. IETF, Internet Draft, 2008. Available: http://tools.ietf.org/html/draft-dusseault-httpmail-00. [Accessed: 23-Jun-2015]
[38]G. Dias, D. G. . Karunarathna, G. P. D. . Udantha, J. A. I. . Gunathilake, P. S. . Pathirathna, and R. A. T. . Rathnayake, “Database based and RESTful email system with offline web based email client,” in *2011 International Conference on Advances in ICT for Emerging Regions (ICTer)*, 2011, pp. 127–127.
[39] M. Frydrych and W. Horzelski, “DBMAIL – database architecture e mail system,” Studia Informatica, vol. 31, no. 2B, pp. 439–448, Jun. 2010.
[40]M. Bazyd\lo, “RESTmail–Design and Implementation of E-Mail System as a RESTful Web Service,” Master’s thesis, Institute of Computing Science, Poznań University of Technology (September 2009).
[41]“Zimbra REST API Reference - Zimbra :: Wiki,” *Zimbra: Email and collaboration for the Post-PC era*. [Online]. Available: http://wiki.zimbra.com/wiki/ZCS_6.0:Zimbra_REST_API_Reference. [Accessed: 22-Apr-2015].
[42] Sendinc, “ REST API - Developer - Sendinc Email Encryption,” *Sendinc Email Encryption*. [Online]. Available: https://www.sendinc.com/solutions/developer/rest. [Accessed: 22-Apr-2015].
[43] Postmark, “Introduction | Postmark Developer Documentation.” [Online]. Available: http://developer.postmarkapp.com/. [Accessed: 22-Apr-2015].

[44] Email Yak , *Email Yak Documentation*. [Online]. Available: http://docs.emailyak.com/. [Accessed: 22-Apr-2015].
[45]Context.IO, "Context.IO | lite," *Context.IO Email API*. [Online]. Available: https://context.io/docs/lite. [Accessed: 22-Apr-2015].
[46]Mailgun, "API Reference - Mailgun REST API 2.0 documentation." [Online]. Available: https://documentation.mailgun.com/api_reference.html. [Accessed: 22-Apr-2015].
[47] PostageApp, "API Overview / API / Knowledge Base - PostageApp Support." [Online]. Available: http://help.postageapp.com/kb/api/api-overview. [Accessed: 22-Apr-2015].
[48] A. Gulbrandsen and N. Freed, "Internet Message Access Protocol (IMAP) - MOVE Extension," RFC Editor, RFC6851, Jan. 2013.
[49] P. Bryan and M. Nottingham, "JavaScript Object Notation (JSON) Patch", RFC Editor, RFC6902, Apr. 2013.
[50] M. Lanthaler, M. Sporny, and G. Kellogg, "JSON-LD 1.0," W3C, W3C Recommendation, Jan. 2014..
[51] M. Kelly, *JSON Hypertext Application Language*. IETF, Internet Draft, 2013. Available: https://tools.ietf.org/html/draft-kelly-json-hal-06. [Accessed: 23-Jun-2015].
[52] M. Amundsen, "The Item and Collection Link Relations," RFC Editor, RFC6573, Apr. 2012.
[53] M. Nottingham, "Web Linking," RFC Editor, RFC5988, Oct. 2010.
[54] "PHP: IMAP - Manual." [Online]. Available: http://php.net/manual/en/book.imap.php. [Accessed: 11-Jun-2015].
[55] C. Corbyn, *Swift Mailer*. 2005.
[56] Open Source Initiative ,"*The MIT License (MIT) | Open Source Initiative*", [Online]. Available: http://opensource.org/licenses/MIT. [Accessed: 11-Jun-2015]
[57] GNU Operating System, "*GNU Lesser General Public License, version 2.1*",[Online]. Available: https://www.gnu.org/licenses/old-licenses/lgpl-2.1.en.html. [Accessed: 11-Jun-2015].
[58] A. L. Hors, D. Raggett, and I. Jacobs, "HTML 4.01 Specification," W3C, W3C Recommendation, Dec. 1999.
[59] E. D. Navara et.al., "HTML 5.1," W3C, W3C Working Draft, Jun. 2014
[60] H. Steen, J. Aubourg, A. van Kesteren, and J. Song, "XMLHttpRequest Level 1," W3C, W3C Working Draft, Jan. 2014.

APPENDIX A: SOURCE CODE

The source code and the libraries which were used with the software have been included in the attached compact disc. A guide on how to install the software for testing has been included with the software source code.