

CURTIN IET ON CAMPUS

INSTITUTE OF ENGINEERING AND TECHNOLOGY

Certificate Maker Kickstart Manual

Harrison G. Outram
Membership Officer

July 3, 2021



The Institution of
Engineering and Technology

Contents

1	Introduction	1
1.1	Context	1
1.2	Problem	1
2	Solution Overview	3
2.1	Rundown	3
2.2	Prototype	4
2.3	Minimal Viable Product	5
2.4	Competitive Product	6
3	Prerequisite Knowledge	7
3.1	Python Language	7
3.2	OSI and TCP/IP Models	7
3.3	HTTP Requests and Responses	9
3.4	Object Orientation	12
3.4.1	Encapsulation	13
3.4.2	Abstraction	14
3.4.3	Inheritance	15
3.4.4	Polymorphism	16
3.5	Application Programming Interface	17
3.6	JSON	18
4	Attendee Management	20
4.1	Internal storage of attendees	20
4.2	Attendee File IO	20
5	Certificate Editing and Creation	21
6	MailChimp Authorisation	22
6.1	Using API and Server Keys	22
6.2	Using OAuth2	23
7	Certificate Uploading	25
7.1	Doing single POST requests	25
7.2	Doing multiple async requests with batches	26
7.3	Using callbacks to retrieve the status	28
8	Contact Updating	29

9 User Interface	30
9.1 Command Line Interface	30
9.2 Desktop GUI	31
9.3 Webapp	32
10 Version Control	33
10.1 Context	33
10.2 What is Git?	33
10.3 Using Github	35
10.4 Recording Library Requirements	36
10.5 Virtual Environments	37
11 Coding Standards	39
11.1 Readability and Maintainability	39
11.2 Documentation comments with docstrings	40
12 Testing Code	43
12.1 Testing tactics	43
12.2 Historical automated testing using asserts	43
12.3 Modern automated testing using pytest	45

List of Figures

1 Old solution flowchart.	2
2 New solution flowchart.	3
3 HTTP 1.1 and HTTP 2.0 visualised.	12
4 Encapsulation visualisation.	13
5 Inheritance UML example.	15
6 Association UML example.	16
7 OAuth2 simplified flowchart.	23
8 Advanced example of curses.	30
9 Simple example of Git workflow.	34
10 Contributions using Git forks.	36

1 Introduction

1.1 Context

Curtin IET On Campus (or “CIET” for short) provides multiple industry talks and workshops to STEM students (particularly engineering and computer related science). As part of CIET’s commitment to quality, attendees shall receive a certificate of attendance stating the name of the event, the club name, the president’s full name with a signature, the the attendee’s full name, and the number of approved CPD hours. This certificate of attendance must be done in a PDF document with vector graphics where possible, then emailed to attendees as soon as possible. This is particularly important for undergraduate engineering students, as one graduation requirement is to obtain a minimum of 16 weighted hours (or five and a third actual) in the PRES category (technical presentations and workshops by a professional body). By creating and sending out these certificates, CIET is holding itself to a high standard of attendee satisfaction and professional development.

1.2 Problem

Despite the necessity of creating and sending certificates, the process for doing so is extremely time consuming and error prone. As shown in Figure 1, this process is comprised of four stages: (1) the attendee record must be collected from the events team and checked for errors, e.g. multiple registrations from one person or missing but required information. (2) one certificate is made for each attendee via the template on CIET’s Canva account. Each certificate is then checked one by one for errors, being recreated if erroneous. (3) the certificates are uploaded to CIET’s MailChimp account, where MailChimp auto-generates a URI for each certificate, required later on. Unfortunately, these URIs involve a randomly generated hexadecimal string, meaning each URI must be recorded manually in the attendee record, then checked for errors. Once the attendee record has been updated with certificate URIs, it is uploaded to MailChimp. (4) The certificates are checked one last time by creating a mock campaign on MailChimp, going through each attendee, downloading the certificate, then checking the certificate for errors and fixing where erroneous. All up, this process takes at least two hours per 50 attendees for someone who has gone through it before, and **far** longer for someone who has not. Worse yet, despite every certificate being checked three times during this process, it is still possible for erroneous certificates to be sent out, which has happened on at least one occasion. Clearly, this process is extremely time consuming and error prone, leading to attendee frustration and a negative impact on CIET’s reputation.

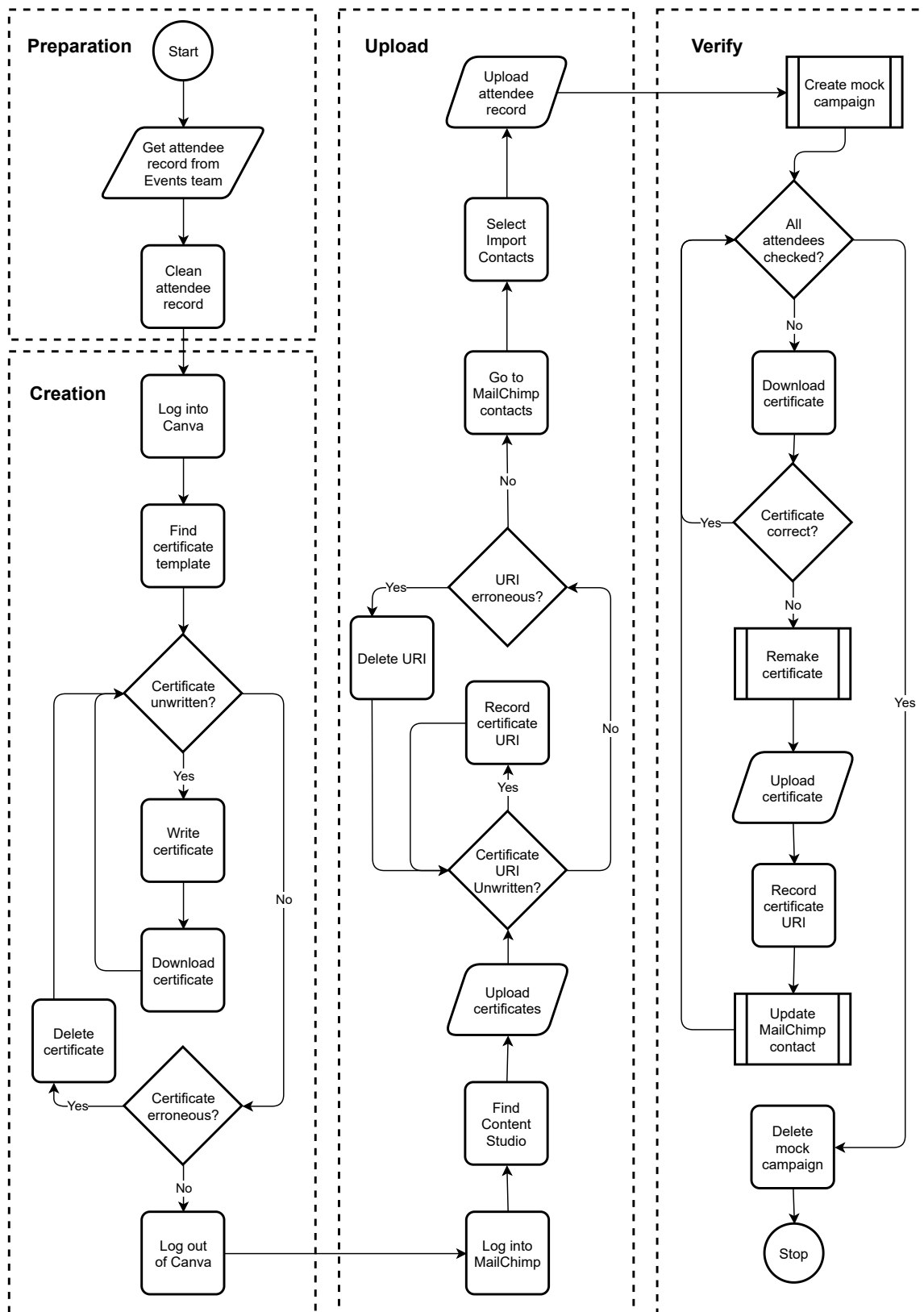


Figure 1: Old solution flowchart.

2 Solution Overview

2.1 Rundown

After some research, it has been discovered that stages 2, 3, and 4 of the old solution can be automated. For stage 1, instead of creating each certificate manually via Canva, the template can be created once and used to automatically create every certificate. Note that the template must be a Word document with merge tags, not a PDF. The MailChimp API can be used to automate stage 3, where each certificate can be uploaded and have its URI recorded. With the URIs known, the attendee record can be updated and uploaded to MailChimp. Since the certificates were created automatically, stage 4 becomes redundant. This process is summarised in figure 2. With such a script, CIET can be confident in its ability to deliver the correct certificates to its members on time with minimal effort.

Realising this goal, however, will require a well organised team and a plan. Due to the varied actions this script must perform, a moderately large codebase utilising several libraries is needed, demanding a significant time investment and expertise. This complexity is compounded by delegating the workload between a team, needing appropriate version control, task management, and enforceable coding standards. Furthermore, such a project could easily suffer from scope creep, especially with the user interface. Hence, the need for subsequent sections in this document.

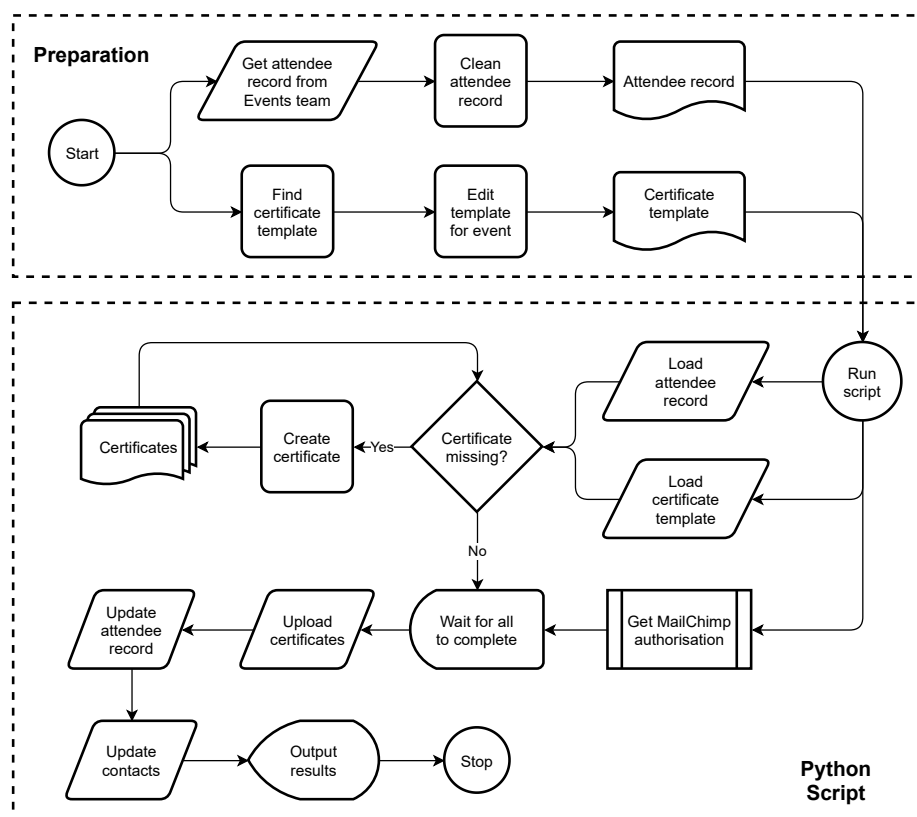


Figure 2: New solution flowchart.

2.2 Prototype

For the sake of time, a prototype can be implemented before scope creep becomes an issue. A prototype must include:

1. Command line interface to run program.
2. Must start program with certificate template path, attendee record path, and Mailchimp API and server keys as command line arguments
 - (a) Strongly recommended to use argparse library.
3. Must be able to load attendee records from a CSV file.
4. Generate a certificate of attendance for each attendee
 - (a) Must be saved into a folder called `/certificates` in the current working directory.
 - (b) Each certificate must be named based on attendee's full name.
5. Upload certificates to Mailchimp, keeping track of the file URIs.
 - (a) Uploading of certificates must be done as a batch, not individually uploaded.
6. Update the attendee record with the certificate URIs.
7. Update the Mailchimp contacts with file URIs.
8. Inform the user of what the program is currently doing.
 - (a) Can just be a simple sentence, e.g. "Uploading certificates to Mailchimp..."
9. Be able to process errors without crashing.
10. Must be able to continue processing certificates when one fails.
11. Must inform user of which step or certificates failed and why.
 - (a) Can be done once the program has finished everything else.
12. All public and protected classes, methods, and function must have Google style docstrings.
13. Must be able to run on Windows, Linux, and Mac systems capable of running Python 3.

For the sake of time and simplicity, the following assumptions can be made:

1. The certificate of attendance is known.
 - (a) The only field to be changed is the attendee name.
 - (b) The exact location of the attendee name field is known and constant.
 - (c) The placeholder text of the attendee name field is known and constant.
 - (d) The certificate is always valid.
2. The attendee record contains no errors.

2.3 Minimal Viable Product

Once a prototype is complete, a minimal viable product (MVP) can be written. The MVP has all the criteria of the prototype, excluding 2, and

1. Can be run via an executable.
 - (a) No command line interfacing should be necessary.
2. Should have a simple and initiative GUI.
 - (a) GUI does not need to be aesthetically pleasing.
3. Can select input files via file explorer.
4. Must log at least info, warning, error, and critical messages to a log file.
 - (a) Debug messages are optional.
 - (b) It must be known where the log file is.
 - (c) The GUI should have a button for viewing the log file.
 - (d) Strongly recommended to use the logging library.
5. Must use OAuth 2 to get Mailchimp authorisation.
 - (a) The program should not have the option to use API keys.
6. Should be able to inform the user of errors that occur in real time without interrupting background tasks.
7. Use appropriate colour coding and symbols to inform user of successes, warnings, and errors.
8. Must be able to instantly respond to user inputs without stuttering.
9. Multiprocessing and multithreading must be used to prevent the program from locking.
10. Must be able to check attendee record for errors.
 - (a) Repeated email addresses.
 - (b) Missing names or emails.
 - (c) Missing required fields.
11. Must be able to edit PDF document in GUI.
 - (a) Must be able to view PDF document within GUI.
 - (b) Must be able to select the field where the full name goes.
12. All code declared as public must be documented using Sphinx.

2.4 Competitive Product

If the MVP is done, a competitive, market ready, version can be done. This version should have everything the MVP has, and

1. Must be able to load attendee records from an Excel file.
2. Has ultra-specific error messages.
 - (a) Informs the user of what exactly went wrong.
 - (b) Where possible, offers a solution.
 - (c) Where possible, offers links to online documentation.
3. Must include install wizard with a build machine.
 - (a) Must be able to choose where to install program.
4. Allows user to select where log files are saved.
5. Allows user to select what types of messages get logged.
6. Must be able to select multiple fields in template as placeholders.
 - (a) Must be able to select what each placeholder gets replaced with graphically.
 - (b) Must be able to select between attendee attribute, constant value (e.g. event name), or system value (e.g. timestamp).
7. Must be able to generate PDF report of what happened.
 - (a) Can select where report gets saved to.
 - (b) Can select what goes on the report.
8. Must have an aesthetically pleasing GUI.

3 Prerequisite Knowledge

3.1 Python Language

While Python was originally designed as a simple scripting language, it has since evolved into one of the most used and popular languages. The simplicity of Python lends itself well to rapid prototyping, so simple that Python's syntax is often considered to be pseudocode. Furthermore, the library support for Python, both first and third-party, makes it extremely easy to make complex tasks simple. The sheer abundance of libraries available creates a compounding effect, as existing libraries makes it easier to wrote other libraries, leading to an exponential growth in library support. Whenever a new protocol or standard becomes prominent, chances are someone has already made a Python library to make it trivial to implement the protocol or standard within other programs. Additionally, the popularity of Python means there is abundant support online for both known issues and asking online for assistance. For the sake of prototyping, it is difficult to find a better language than Python.

Considering the abundance of support for Python, learning the language is best done with interactive tutorials foudn online. Such examples include W3 Schools, Codecademny, and Learn Python. Numerous other tutorials exist, such as those on Tutorials Point and the countless YouTube videos; however, these tutorials lack an interactive component, leading to reading or watching without learning. When learning a new language, the source is less important than the interaction between the material and the student.

Where Python's weakness lies is in its performance. Since Python is a weak typed language, datatype checking must be done during runtime, greatly reducing performance. Far worse, the interpreted nature of Python means compiler optimisations cannot be performed, slowing down Python to a crawl. Compared to optimised compiled languages, such as C, Rust, and C++, Python performs dozens of times worse when comparing equivelant programs. This can be greatly mitigated by taking advantage of scientific libraries, such as NumPy and Pandas. These libraries are written in C and C++, allowing for speeds within the ballpark of C, C++, and Rust. However, certain conventions must be followed to take advantage of vectorised data, leading to data needing to be structured in vectors or matrices. Unfortunately, this will only result in a performance increase when doing mathematical operations on numeric data, such as vector dot products or summing an array. Ergo, Python is a poor choice for performance critical applications.

3.2 OSI and TCP/IP Models

To understand how programs interface with each other over the internet, the Open Systems Interconnection (OSI) model must be introduced. Starting in the late 1970s, the OSI model details how computers, or "nodes", communicate with each other over a network. The model is split into seven layers, each with their own purpose and protocols, shown in table 1. Each layer only needs to know about the adjacent layers, decoupling the responsibilities as much as possible. When sending messages over a network, the sender goes through the layers from top-to-bottom,

Table 1: OSI Model for computer networking

No.	TCP/IP Model	OSI Model	Protocols
7	Application	Application	HTTP, FTP, WS, RTP
6		Presentation	JPEG, TLS, SSL
5		Session	NFS, SQL, PAP
4	Host-to-Host	Transport	TCP, UDP
3	Internet	Network	IPv4, IPv6
2	Network Access	Data Link	ARP, CDP, STP
1		Physical	Ethernet, Wi-Fi

whereas the receiver goes the layers bottom-to-top. Despite the significant learning curve, the OSI model successfully delegates the numerous responsibilities of network communication across seven mostly decoupled layers.

1. Physical Layer

What physical devices are used to connect the nodes within a local area network (LAN). By extension, also dictates the bit rate, what receives data, and what sends data. Does not understand data beyond a sequence of 1s and 0s.

2. Data Link Layer

Establishes and terminates a connection between two physically connected nodes. Data is broken into packets and organised into frames. Uses a logical link control (LLC) to identify network protocols, check for errors, and synchronise frames. Also uses Media Access Control (MAC) addresses to uniquely identify nodes and setup connections, including permissions.

3. Network Layer

Uses routers to sent segment of data from layer 4 and pair them with an address, typically an IPv4 address. Each segment with sent between routers to the correct LAN, where layer 2 takes control. By extension, also responsible for determining the fastest path to sent the segments through routers. Without layer 3, communication between LANs (e.g. the entire internet) would not be possible. Sometimes known as a “connectionless” protocol, as messages are sent without checking they were successfully sent.

4. Transport Layer

Takes data being sent from layer 5 and breaks it into segments to be sent over layer 3. Also responsible for reassembling segments back into the full message on the receiving end. Transmission Control Protocol (TCP) prioritises reliability over speed, ensuring that segments (called packets) are received intact. If a segment is corrupt or not sent within a time limit, the receiver asks for

the segment again. The receiver must also send back a message. If the response is incorrect, the sender resends the message. Alternatively, User Datagram Protocol (UDP) treats segments as datagrams, sending them without any error checking, prioritising speed over reliability. Just like layer 3, UDP is also known as connectionless. The former is used for internet browsing and uploading to a server, whereas the latter is used for streaming videos, music, and playing games online.

5. Session Layer

Manages communication channels, or “sessions”, between devices. Responsible for opening sessions, ensuring sessions remain open during data transfer, and closing sessions. Can also setup “checkpoints”, ensuring that if the data flow is interrupted, it can be continued later without restarting.

6. Presentation Layer

Prepares data from layer 7 before being sent via layer 5. Performs compression, encoding on the sending end, decoding on the receiving end, and encryption. The ‘s’ in “https” stand for secure and is handled by the Transport Layer Security (TLS) protocol, often mistaken for the outdated and hacked Secure Socket Layer (SSL).

7. Application Layer

The programs that are generating the data being sent and received. Examples include web browsers, email clients, servers, and streaming services such as Netflix and Twitch. Also responsible for presenting the data in a meaningful way to end users.

The observant reader would also notice that there is a simplified TCP/IP model, only comprised of four layers. In it, the bottom two layers of OSI are combined into the Network layer, where frames are transmitted between nodes on a LAN via mapping IP addresses and port numbers to MAC addresses. The next layer, called the “Internet layer”, uses the assigned IP addresses to direct packets to the correct LAN. The internet layer also uses Internet Control Message Protocol (ICMP) to inform hosts about network issues, encapsulated within the IP dataframes. To determine the physical address of a node, layer 2 also uses Address Resolution Protocol (ARP). Layer 3 in the TCP/IP model is identical to the transport layer in the OSI model.¹ Lastly, the top three layers of OSI are combined into the application layer in TCP/IP model. Due to the reduced complexity, the TCP/IP model is often used over the OSI model for internet communications.

3.3 HTTP Requests and Responses

Of interest to interfacing with internet-based servers is the hypertext transfer protocol (HTTP). Invented circa 1990, HTTP is a request-response protocol, transferring data in simple text documents.

¹Despite the name, UDP can be used in the TCP/IP model

As of the time of writing, HTTP 1.1 is the most commonly used version, with 2.0 slowly replacing 1.1. In all versions, the first step is for the client to connect to the server via a TCP handshake. Once a connection has been established, the client can proceed to send a “request”, which the server processes then replies with a “response”.

The request is a text document made of three, sometimes four parts. The first line is always the request header, formatted as the method type, followed by the path to the resource, then the HTTP version. Afterwards, each line then comprises the one key-value pair of the header, used to specify metadata, such as the host, accepted languages, authorisation codes, connection type, and expected content type(s). To end the header, an empty line must then follow. For some requests, a body may also be necessary, such as the content being uploaded to the server. Below is an example of such a request.

```
1 POST /boards/baking_for_life/comments HTTP/1.1
2 Host: www.myforum.example:8080
3 Authorization: QWxhZGRpbjpvcGVuIHNlc2FtZQ==
4 Connection: keep-alive
5 Content-Encoding: gzip
6 Content-Type: application/json
7 Date: Mon, 12 June 2005 10:12:58 GMT
8 User-Agent: Mozilla/5.0
9
10 {
11     "username": "John_Smith_Official",
12     "board": "Baking for Life",
13     "comment": "what are the best tools for making a layered cake?"
14 }
```

Once the server has finished processing the request, it sends back a response, formatted very similarly to a request. The response’s first line is always the status line, formatted as the HTTP version, the status code, then the status message. The status code is a three digit code informing the client of how the server handled the request. The first digit is the broad type of code, then the next two digits specifies the exact type of code. The status message is simply a humanly readable message version of the status code. After the status line, the header fields are then declared line-by-line, ending with an empty line. For some requests, a body will also be included after the empty line. See the example below.

```
1 HTTP/1.1 200 OK
2 Age: 205699
3 Connection: keep-alive
```

```
4 Content-Encoding: gzip
5 Content-Length: 103
6 Content-Type: text/html
7 Date: Mon, 12 June 2005 10:13:26 GMT
8 Server: Apache/2.4.1 (Unix)
9
10 <html>
11 <body>
12 <h1>Hello, Curtin IET On Campus!</h1>
13 </body>
14 </html>
```



HTTP Status Codes For long messages split into multiple requests, a 1xx status code means the server successfully received the segment and the client should send the next. 2xx means the message was received, understood, and processed successfully. 3xx means the server does not have the resource being requested but does know another server that does.^a 4xx means the client did something wrong, such as the classic 404 file not found error. Lastly, 5xx means the server did something wrong.

^aResponses with status codes of 3xx should always include the server connection details in the body

As of HTTP 1.1, there are nine accepted methods. Introduced in HTTP 0.9 (the first public version), the GET request is a simple read only request to retrieve a copy of a resource from the server. As of HTTP 1.0, the POST request allows for clients to upload new resources onto the server. Also as of HTTP 1.0, the HEAD request works identically to a GET request, expect that the body is not returned. When HTTP 1.1 was made available, two important methods were added: A PUT request allows a client to replace an existing resource on a server with the one in the request body, whereas a DELETE request simply tells the server to delete the specified resource. Similar to a PUT request, a PATCH request tells the server to partially update a resource instead of replacing it. While not commonly used, HTTP 1.1 also introduced the TRACE and OPTIONS request types. The former performs a message loop-back test from the path of the specified resource, whereas the latter requests the server to describe all available communication options for the specified resource. Lastly, the CONNECT request allows clients to connect to a server through another server, useful for setting up proxy servers or websites using TSL (i.e. HTTPS).

While HTTP 1.1 remains the predominant version as of the time of writing, HTTP 2.0 will eventually replace it. HTTP 1.1 introduced vast performance improvements over 1.0, most notably the ability to remain connected to the server after each request-response cycle instead of having to reconnect to the server. However, in 1.1, the server can only process one request from any client at a time, meaning the client must wait until the current request has been responded before

sending another. This has changed in 2.0, allowing for multiple requests to be sent, processed, and responded to after a connection has been established, shown in figure 3. For now, this project can ignore HTTP 2.0, but future versions may need to consider the inevitable obsolescence of HTTP 1.1.

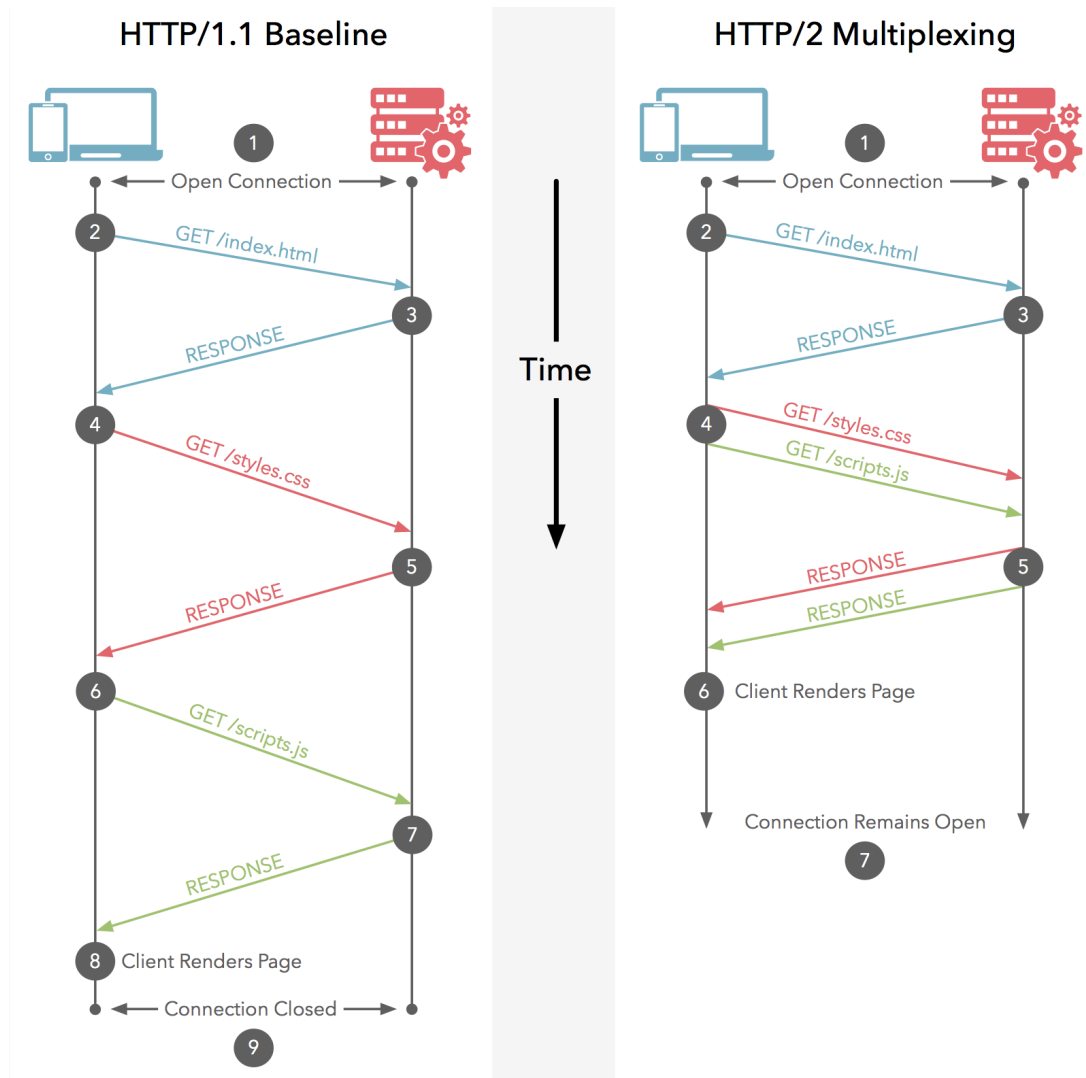


Figure 3: HTTP 1.1 and HTTP 2.0 visualised.

3.4 Object Orientation

Critical to software engineering, object orientated programming (OOP) has proven invaluable to large scale and collaborative code-based projects. This is possible via a sophisticated application of the broad divide and conquer approach, splitting complex programs into simple and decoupled modules and submodules. OOP achieves this through four main principles: encapsulation, abstraction, inheritance, and polymorphism.

3.4.1 Encapsulation

Consider the logistics in running a store: the stock available, the stock reserved for customers yet to buy them, preorders, cancellations, the stock on its way, the customers who have signed up as members, newsletters, purchases, overhead costs, etc. How does a business maintain such logistics? Who records the data? Who files and organises the data? Who has access to what data? Who is authorised to fix errors in the data and what process must they go through to make these fixes? Perhaps most importantly, who or what ensures the data is valid *before* it is placed with existing data? Such problems can be at least mitigated using the OOP paradigm.

Since objects contain data as variables, the objects can be made responsible for protecting the data from corruption. Instead of making the variables directly accessible, the object can have getters and setters (also known as mutators) to access the data. Getters should always return a copy of the variable, whereas setters must validate the new value before setting the variable. For example, it is impossible for a jar to have a negative number of marbles, hence if outside code attempts to remove more marbles from a jar object than what is available an error can be raised and the jar's number of marbles remains unchanged. Similarly, the jar will probably also have a capacity, so that if too many marbles are attempted to be inserted a different error should be raised. Some object variable may not even have getters, such as the length of a collection (e.g. an array or a linked list). Instead, the length variable should automatically change as elements are inserted or removed. Even further encapsulated, other variables may not even have getters, as they may be variables that should only be used by the object internally, such as an array inside of a hash table. Protecting data like this greatly mitigates the data from being corrupted or otherwise useless.

In Python, encapsulation is implemented very differently to other OOP languages. Languages like Java, C#, and C++ have the keywords `public`, `protected`, and `private` for deciding how hidden a variable or method is. `public` simply means anything that has access to the class or object can access the member, be it a variable or method. `private`, however, means only the class or object can access the member, completely hiding it from all other code. While useful for data protection, it was quickly found that `private` is too restrictive for many applications. Instead, `protected` has replaced `private`, as it allows subclasses to access the member, discussed further

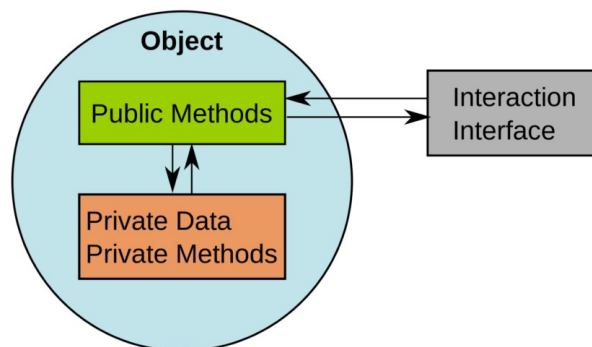


Figure 4: Encapsulation visualisation.

in the Inheritance subsection below. Python does not have these keywords, instead opting for using underscores: no underscores declares the member (or “attribute” as it is known in Python) as public, a single underscore means protected, and a double underscore, or “dunder”, means private. Protected and private variables can then be read using the `@property` decorator, and set using the `@x.setter` decorator, where `x` is the variable being set. Note the example below and the lack of dunder.

```
1 class MarbleJar:
2     def __init__(self, material, capacity, num_marbles = 0):
3         self._material = material
4         self._capacity = capacity
5         self._num_marbles = num_marbles
6
7     @property
8     def material(self): return self._material
9
10    @property
11    def capacity(self): return self._capacity
12
13    @property
14    def num_marbles(self): return self._num_marbles
15
16    @num_marbles.setter
17    def num_marbles(self, new_num_marbles):
18        if new_num_marbles < 0:
19            raise ValueError("Number of marbles cannot be less than 0")
20
21        self._num_marbles = new_num_marbles
```

3.4.2 Abstraction

As a program become more and more complex, it becomes increasingly difficult to manage the functionality of the program. Trying to track where the functions of a program are organised and located turns impractical. Even trying to determine if a function has been implemented becomes time consuming and infeasible. This can be partially mitigated by organising functions into thematically narrow and consistent libraries; however, even this approach leads to too many functions to keep track of what has and has not been implemented. For complex programs, especially those requiring extensive interaction with other programs, a smarter approach is needed.

Abstraction attempts to solve this issue by hiding functionality within objects. Abstraction is simply the process of hiding the implementation by only exposing what the object can do. Hence,

outside code only needs to concern itself with what the object can do and not how it does it. Since objects can contain other objects, complex tasks involving several steps using numerous variables can be split across a tree-like hierarchy of objects. Consider the OSI or TCP/IP model detailed above: with OOP all that is needed is a single `NetworkManager` that handles how an application sends and receives data without code outside the `NetworkManager` needing to know about layers below the application layer. Therefore, outside code only needs to call a single method of one object and let it handle how the task is completed.

3.4.3 Inheritance

When similar entities need to be modelled, code repetition becomes an issue. Consider a pet tracker for a veterinary that needs to store various types of pets. Obviously, snakes and rabbits are two very different animals that require very different code to model. However, cats and dogs are similar enough that the code modelling these pets would at least somewhat repeat. Repeated code makes it more difficult to maintain, as each repeat must be individually tested, debugged, and modified due to changing environments or requirements, leading to the risk of some repeats not being debugged or changed properly. By extension, this also makes testing more expensive, time-consuming, and error-prone, as each repeat code block being tested will lead to repeat test cases. Clearly, such repetition needs to be eliminated.

OOP programming provides such a solution in the form of inheritance. Simply put, inheritance is the ability of some classes to inherit the code of other classes. For the pet tracker example, a base class called `Pet` or `Animal` can be made that contains code common to all pets. E.g., values like owner, pet ID, and date of birth. Further classes can be written that eliminate code repetition

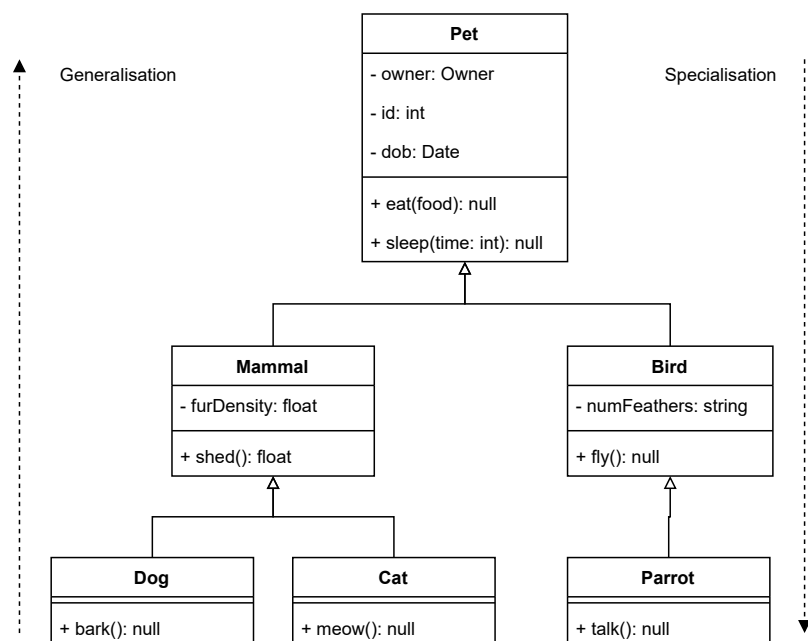


Figure 5: Inheritance UML example.

for groups of types of pets, e.g. Mammal, Reptilian, Bird, and Amphibian, each of which should inherit from the Pet base class. Then, more specific pet types can be written that inherit from these broad pet types, such as Dog, Cat, Turtle, Snake, Parrot, and Frog. Such inheritance is known as the *is-a* relationship, as a dog *is a* Mammal, which *is a* animal. Designing a codebase like this completely eliminates repetitive functions, greatly speeding up the process of writing the codebase and simplifying test cases.

Additionally, inheritance also covers one object being contained within another. This type of relationship is known as the *has-a* relationship or association. For example, a car is *not* a type of steering wheel and vice-versa. However, a car *has a* steering wheel. Consider a car manufacturer that needs to design and manufacture cars. While different car models are not identical, they may share identical steering wheels, gearboxes, transmission systems, fuel tanks, wheels, or seats, among other numerous parts. Hence, these identical parts can be written, tested, and debugged once, then shared across the different car classes. This type of association is known as composition, as the car fully owns the parts it is made of. If the car object is destroyed, the car parts are also destroyed. This is opposed to aggregation, in which the child object can exist independently of the parent class, such as the cars and the manufacturer. As with the *is-a* relationship, association greatly reduces code repetition.

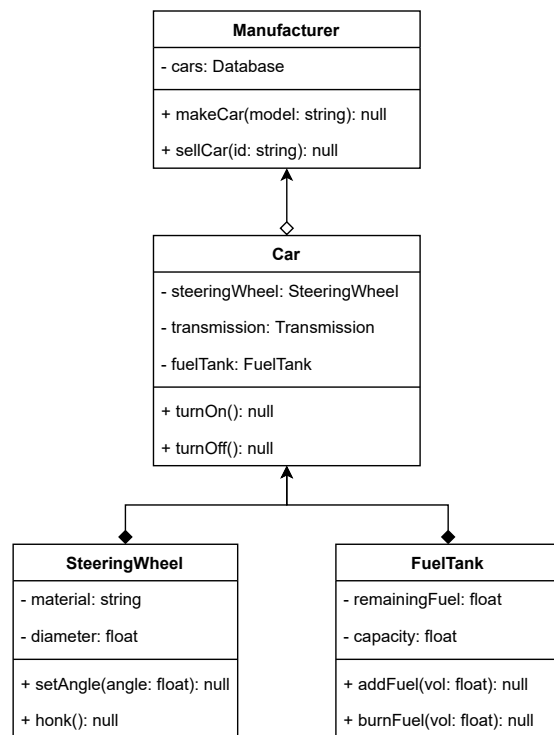


Figure 6: Association UML example.

3.4.4 Polymorphism

When developing a codebase, it occasionally becomes useful to have multiple functions or methods that are named the same but perform differently. In C, the `math.h` library contains functions for finding the absolute value of a number. There are actually two of these functions: the `abs()` function and the `fabs()` function. The reasoning being is that `abs()` takes in an integer whereas `fabs()` takes in a double. Since C is a strong typed language with no polymorphism, this practice of having multiple functions perform essentially the same task becomes inevitable. Even more complicated, consider the case of calculating the area of a shape. Since the areas of shapes are calculated differently from shape to shape, in C there would be one function per type

of shape. E.g. `circleArea()`, `rectArea()`, `triangleArea()`, etc. As program requirements start involving extremely similar functions and datatypes, the number of near-identical functions to keep track of becomes impractical.

Polymorphism solves this issue by allowing multiple functions to share the same name. One such manner of doing so is called “method overloading”: allowing for a class or object to have multiple methods with the same name but different parameter lists in terms of their datatypes. In Python, this is actually not possible since the datatypes of variables does not have to be declared. Instead, Python achieves pseudo-overloading by allowing for a parameter to have one of multiple valid datatypes (such as `num` being either an `int` or an `float`) or by having parameters with default values. Further still, a function in Python can have an optional list of miscellaneous, unnamed parameters and another optional dictionary of named parameters. For example, the Python function

```
1 def formatNumbers(num, *args, precision=2, **kwargs):  
2     ...
```

must be called with a number, but can allow multiple arbitrary numbers to be formatted via the `*args` list, optionally specify the precision to format the numbers to (defaults to 2 decimal places), and also be called with arbitrary named parameters that are stored inside the `**kwargs` dictionary (short for “keyword arguments”).

Using the *is-a* inheritance principle, it is also possible to achieve polymorphism via “method overriding”. Going back to the area of shapes example, we can calculate the areas of different shapes without needing multiple methods with different names. Instead, a base class `Shape` can be written, that has the method `calcArea()`. This method is known as an “abstract method” since it has not been implemented, meaning the `Shape` class is also abstract. Abstract classes cannot be instantiated into objects, making them useless by themselves. Instead, subclasses of the `Shape` class can be written that implement this abstract method, such as `Circle`, `Rectangle`, and `Triangle`. The significance of which means that outside code does not need to concern itself with what exact type of shape a `Shape` object is, only that it has a `calcArea()` method. Hence, functions can be written once for any arbitrary shape object and more shape types can be written latter without having to change any of these functions.

3.5 Application Programming Interface

The world of modern software development is one heavily comprised of using code and services created by third-party organisations or individuals. This is an inevitable consequence of the daunting product requirements that modern software must abide by to remain competitive. However, the pitfalls of using third party services quickly become apparent: *How do we know future changes won't break our code? Will our application still work when this service changes? What security flaws or exploits are exposed allowing others to use our service? What do we do if a third party service does not allow us to access their service via code?* Consider the need for a

music editing mobile app that can use music owned by the user from their Spotify, Google Music, Bandcamp, or Apple Music accounts. How does a development team even begin to integrate these services their mobile app?

Solving these concerns introduces the need for Application Programming Interfaces, or APIs. An API, simply put, is a set of commands a program can use to access and upload data to another program. They can be as simple as the set of functions and constants within a library to the monolithic 2 billion line of code Google API. Well designed APIs allow external parties to develop their own applications integrating your service without needing to know how the API works, nor what the external application is or does. APIs also greatly mitigate security issues, as APIs can hide the implementation of the service. External developers using an API can also be sure that future changes to the API will not break their application, as API updates come with a version number, signifying if an API change has occurred. If an API change has occurred, the external developers can simply not use the newer version until they have updated their application to use the newer API. Most relevant to this project, the Mailchimp API is the only way a program can access a user's Mailchimp account without hacking the Mailchimp servers. Without this API, the project would not be possible.

3.6 JSON

While developers can use whatever common or obscure data format they want for their APIs, JavaScript Object Notation (JSON) has dominated API usage in recent years. Originally designed for the Javascript programming language as a way of creating objects without a class or constructor function, JSON has since evolved into a text format for storing and transporting data in a lightweight manner. While confusing for new comers, it quickly becomes evident as to why JSON is so popular: its ability to scale for various types of data easily in a humanly readable format.

JSON is comprised of a few special characters and a few datatypes. Every JSON message always starts and ends with curly brackets, denoting the message as an object. The properties of the object are then writtin in key-value pairs, where the key is always a string. The value can either be a string (within quotations), a number, an array (denoted using square brackets), a boolean (either `true` or `false`), `null`, or a nested object (denoted with curly brackets). The following JSON conventions are either required or strongly recommended:

1. Key names must always be strings.
 - (a) Key names should be alphabetical where possible.
 - (b) Key names should use hypens or (prefferably) underscores instead of spaces.
 - (c) `snake_case` is preferred over `camelCase` for key names.
2. Whenever the elements in a collection have unique identifiers, they should be stored in a nested object.

3. Whenever the elements in a collection do not have unique identifiers, they should be stored in an array.
4. Whenever the elements in a collection need to be ordered, they should be stored in an array.
5. Nested objects should contain data modelling a single entity.
6. An array should contain elements of the same datatype.
7. If an array stores objects, every object should have the same keys.
8. Always use quotations ("") instead of quotes (") when declaring a string.
9. If a value is not known, assign it as `null`.

Below is a simple example of a club's details stored in JSON format:

```
1 {  
2   "club_name": "The Book Club",  
3   "start_year": 2007,  
4   "university": "Curtin University",  
5   "bank_balance": 120.92,  
6   "oday_enrolled": false,  
7   "age_demographic": null,  
8   "club_themes": [  
9     "books",  
10    "literature",  
11    "casual"  
12  ],  
13  "members": {  
14    "ordinary": 120,  
15    "associate": 12  
16  },  
17  "committee": {  
18    "president_id": 1992350,  
19    "vice-president_id": 18988823,  
20    "treasurer_id": 2000971,  
21    "secretary_id": 1945670  
22  }  
23 }
```

4 Attendee Management

4.1 Internal storage of attendees

Each attendee will need to be stored as an object, requiring the ability to read and write both required and optional attendee attributes. At a minimum, the attendee objects will need to store the email address, given name, and family name. Furthermore, future upgrades may require other attributes to be stored. Since these optional attributes are not known, the attendee object must have methods for adding, getting, updating, and removing any arbitrary attributes. Fortunately, the attendee objects only need to store data, meaning no methods beyond setters and getters are needed.

Once the interface for an attendee class has been agreed upon, an attendee manager can be written. This attendee manager will store all attendee objects for convenience across all other modules that use the attendee records. Hence, the attendee manager will need methods for adding attendees, iterating over attendees, and removing attendees. Updating attendee records can be done via the attendee objects themselves, as they will already validate the data. Since the attendee objects already validate themselves, the attendee getters in the attendee manager should return a reference to the attendee, not a copy. This is to ensure that updating an attendee record can be done with removing and re-inserting the attendee. To keep track of each attendee within the manager, each attendee will need some unique identifier, which can be the email address.

4.2 Attendee File IO

Given that the attendees must be loaded and stored from a CSV file, an attendee file manager will be needed. Since the attendees are stored as tabular data, the Python Pandas library can be used to store the attendee data. It is important to note that this data is raw in nature, meaning it will not be validated beyond any file IO error checking. The purpose of the attendee file IO module is just to load the raw data in tabular form, needing further data validation from the attendee and attendee manager classes. Ergo, an iterator should be implemented to retrieve the raw attendee data one-by-one.

Once the attendee records have been internally finalised, they will then need to be saved to a file. Considering that the data is already in tabulated form, it should be a simple matter of saving the data in a CSV file.

5 Certificate Editing and Creation

Once the attendees are known, the certificates can be made. Unfortunately, the certificate template on Canva is useless, as it cannot be downloaded as a editable document. Instead, the certificate template should be recreated in Word with at least one merge field or variable (denoted via double braces) for the attendee name. Word documents can then be loaded into Python with their merge fields or variables identified via the `docx-mailmerge` or `docxtpl` Python libraries. Once the attendee's details has been inserted, the certificate can be saved as a Word document then converted to a PDF via the `docx2pdf` library. With these libraries, all certificates can easily be created from a template.

6 MailChimp Authorisation

To prevent malicious parties from accessing Mailchimp accounts, applications must provide authorisation with each HTTP request. The Mailchimp API provides developers two methods of auto-authorisation: an API key and OAuth2, both with their strengths and weaknesses.

6.1 Using API and Server Keys

Owners of a Mailchimp account can use their API and server keys to allow applications to access their account. The developer simply needs to log into their Mailchimp account, generate an API key, find the server key in the URL, then input both keys into their application. The power of API keys lies in their simplicity, only needing a few minutes to generate and start using the API key. Once the key has been retrieved, the key just needs to be included into the config of HTTP requests to Mailchimp, only taking a few lines of code. The example below is taken straight from the Mailchimp API:

```
1 from mailchimp_marketing import Client
2
3 mailchimp = Client()
4 mailchimp.set_config({
5     "api_key": "YOUR_API_KEY",
6     "server": "YOUR_SERVER_PREFIX"
7 })
8
9 response = mailchimp.ping.get()
10 print(response)
```

If the request succeeds, the response body will contain

```
1 {
2     "health_status": "Everything's Chimpy!"
3 }
```

However, an API key is equivalent to a root admin's login details; giving full permissions to the entire account. With an API key, a developer can change user permissions, modify contacts, and send emails. For security reasons, all API keys can be disabled at any time, preventing compromised keys from being used negligently or maliciously. Regardless, it is extremely rare an app will need full access to a Mailchimp account.

Furthermore, generating then inputting their API is a very user unfriendly task. First off, the average user will have no idea what an API even is, none-the-less an API key. Secondly, the user must ensure to never share or write down their API key outside of the app, which is extremely

unlikely, causing a high security risk. Similarly, the app must be secure enough to not expose the API key, which is extremely difficult for a webapp. If an API key is exposed by a service provider, the provider must notify their users ASAP and the users must disable the API key before being hacked. Clearly, API keys present a high risk for applications, especially when accessing accounts not owned by the service provider.



Warning: Under no circumstances should an API key ever be disclosed to anyone! This includes any written form, even in source code! Authorised individuals should log into the associated account to get the API key. The API should not be written down, instead it should be copied to the devices clipboard, where it can be pasted into an application's input. To input an API key into an application, it should be inputted at run time as a command line argument, an environmental variable on a secure device, or through an input text box. It should never be saved by the application or sent anywhere outside the service requiring the API key.

6.2 Using OAuth2

A far more modern method of gaining access to a user account from another service is to use the OAuth2 protocol. Short for Open Authorization 2.0, OAuth2 has quickly become the de-facto way for electronic services from different organisations to communicate with each other automatically. The protocol is comprised of several steps: first, the app must be registered on the service, which will provide a `client_id` and `client_secret`. Next, the app the user is running requests access from the user to another service. If the user agrees, the app makes an authorisation request to the third-party service using the `client_secret`, redirecting the user to login to the account they own. Once logged in, the service will ask the user to confirm that they consent to giving the displayed permissions to the app. If the user agrees, the user is then redirected back to the

Abstract Protocol Flow

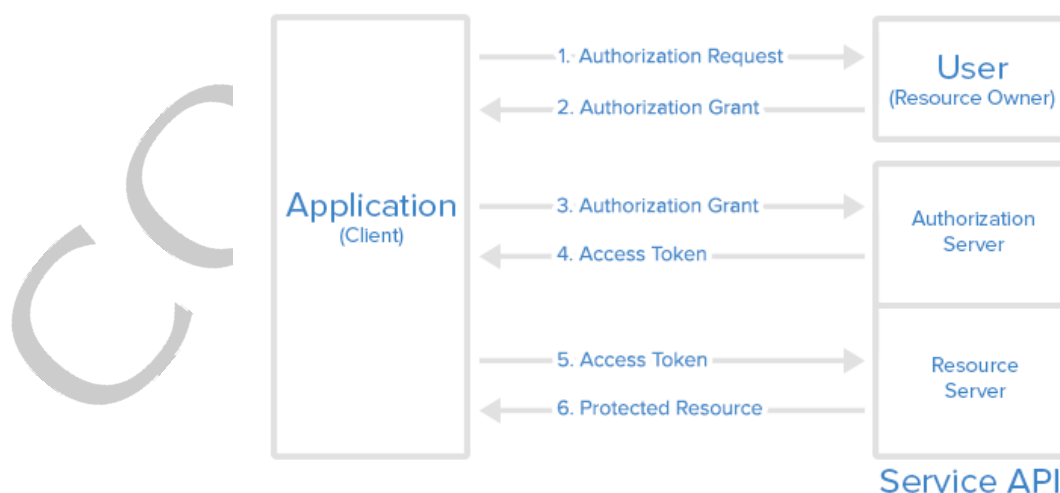
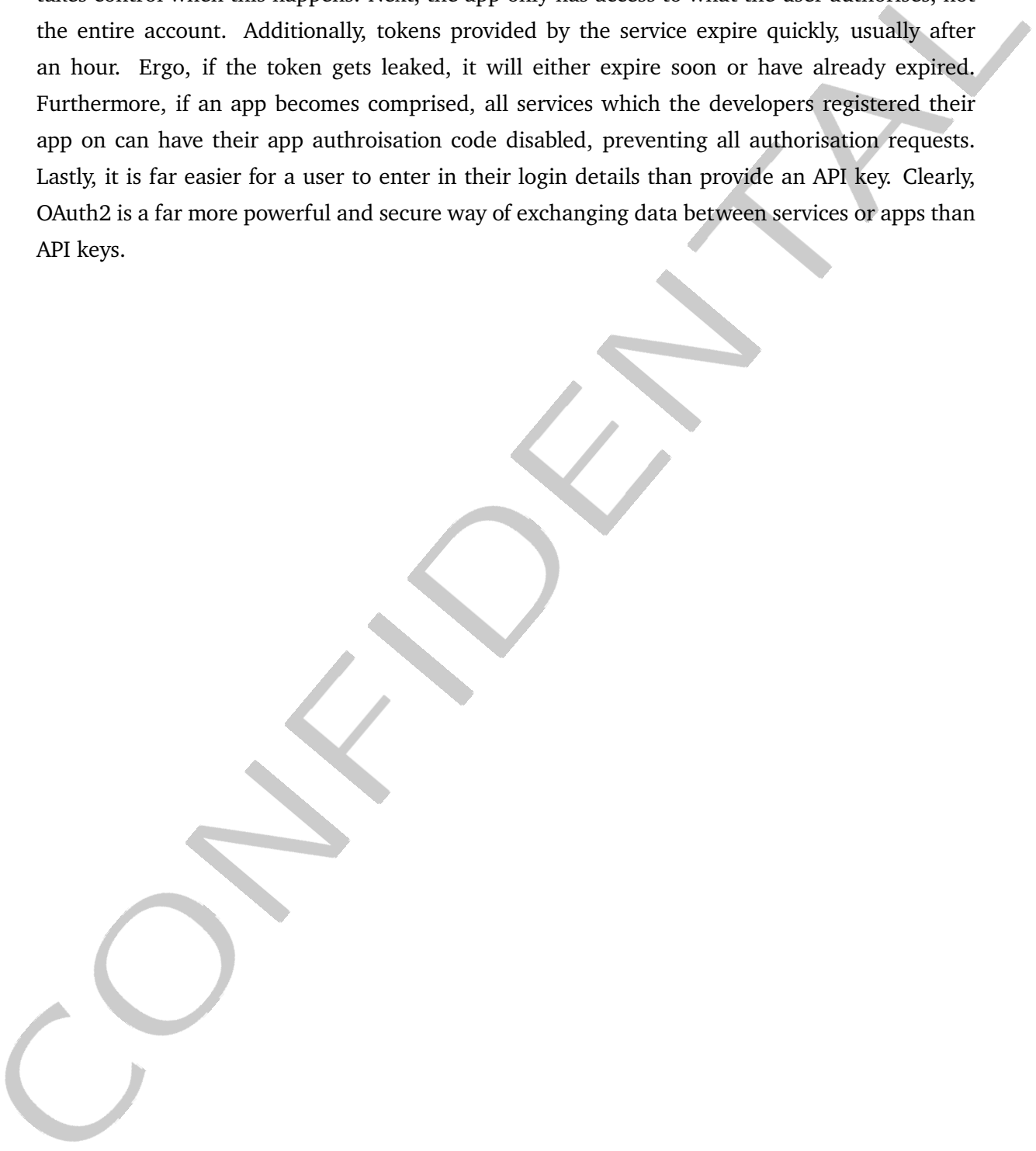


Figure 7: OAuth2 simplified flowchart.

app, where it uses the access token (a long, computer generated hexadecimal string) given by the service to access the user's resources. Figure 7 visualises this process.

While more complicated to implement than using an API key, OAuth2 has multiple key advantages. Firstly, the app requesting access never observes the login details for the service, as the service takes control when this happens. Next, the app only has access to what the user authorises, not the entire account. Additionally, tokens provided by the service expire quickly, usually after an hour. Ergo, if the token gets leaked, it will either expire soon or have already expired. Furthermore, if an app becomes comprised, all services which the developers registered their app on can have their app authorisation code disabled, preventing all authorisation requests. Lastly, it is far easier for a user to enter in their login details than provide an API key. Clearly, OAuth2 is a far more powerful and secure way of exchanging data between services or apps than API keys.



7 Certificate Uploading

7.1 Doing single POST requests

After gaining Mailchimp authorisation, the certificates can be uploaded to Mailchimp. First of, a folder should be created to store the files within. According to the Mailchimp API, this is possible by making the POST request below:

```
1 import mailchimp_marketing as MailchimpMarketing
2 from mailchimp_marketing.api_client import ApiClientError
3
4 try:
5     client = MailchimpMarketing.Client()
6     client.set_config({
7         "api_key": "YOUR_API_KEY",
8         "server": "YOUR_SERVER_PREFIX"
9     })
10
11     response = client.fileManager.create_folder({"name": "name"})
12     print(response)
13 except ApiClientError as error:
14     print("Error: {}".format(error.text))
```

The response of which will contain the folder ID, an integer that uniquely identifies that folder. Given the folder ID, the certificates can then be uploaded via a POST request

```
1 import mailchimp_marketing as MailchimpMarketing
2 from mailchimp_marketing.api_client import ApiClientError
3
4 try:
5     client = MailchimpMarketing.Client()
6     client.set_config({
7         "api_key": "YOUR_API_KEY",
8         "server": "YOUR_SERVER_PREFIX"
9     })
10
11     response = client.fileManager.upload({
12         "name": "name",
13         "file_data": "file_data",
14         "folder_id": folder_id})
15     print(response)
```

```
16 except ApiClientError as error:
17     print("Error: {}".format(error.text))
```

The responses of uploading the files will contain a `full_size_url` field, which is needed for the next section. Hence, a functional Mailchimp manager module will need a function to take in the certificates, the certificate filenames, and the folder name. Given these parameters, the folder must be created first, have its folder ID recorded, then upload the certificates into the folder. As the responses from uploading the certificates are received, the `full_size_url` must be recorded and returned to the caller.

7.2 Doing multiple async requests with batches

However, the exact manner in which the requests are made can cause the program to lock up for prolonged periods of time. This is due to the `client.fileManager.create_folder()` and `client.fileManager.upload()` functions being “blocking functions”. Such functions halt the program’s progress for a long period of time while waiting for a response from the Mailchimp server. Furthermore, to prevent server overloading, the Mailchimp API will only allow 10 concurrent requests per API key. Considering that it takes 120 seconds for the for a request to time out, the worst case scenario is that it will take an hour and 40 minutes to process 50 attendees in a sequential manner. With concurrent requests, this worst case reduces to 10 minutes, still far too long for a user to wait.

The solution is to use Mailchimp’s batch feature. Instead of making one POST request per certificate, all the POST requests can be bundled into a batch. Making the batch request given the individual requests is trivial, as the body of the request is simply a JSON object containing one field, `payload`, whose value is a Python list (or Javascript array) of the individual requests. The code below is Mailchimp’s example of adding 1000 contacts at once.

```
1  from mailchimp_marketing import Client
2  import json
3
4  mailchimp = Client()
5  mailchimp.set_config({
6      "api_key": "YOUR_API_KEY",
7      "server": "YOUR_SERVER_PREFIX"
8  })
9
10 list_id = 'YOUR_LIST_ID'
11
12 users = [
13     {
```

```
14         'id': '1',
15         'email': 'user1@example.com'
16     },
17     {
18         'id': '2',
19         'email': 'user2@example.com'
20     },
21 ]
22
23 operations = []
24 for user in users:
25     operation = {
26         "method": "POST",
27         "path": f"/lists/{list_id}/members",
28         "operation_id": user['id'],
29         "body": json.dumps({
30             "email_address": user['email'],
31             "status": "subscribed"
32         })
33     }
34     operations.append(operation)
35
36 payload = {
37     "operations": operations
38 }
39
40 response = mailchimp.batches.start(payload)
41 print(response)
```

Furthermore, the `response = mailchimp.batches.start(payload)` is not a blocking function. Instead, it returns a response that Mailchimp generates and sends as soon as possible, even though the individual requests have not been processed. The body of the initial request includes a batch ID, which can be used to do another request to see the status of the batch via

```
1 response = mailchimp.batches.status(batch_id)
```

Once all requests have been processed, a gzipped archive of the responses can be downloaded through the `response_body_url` available in the batch response.

Since downloading, extracting, and processing the responses would be slow, it is strongly recommended the method that makes the batch request simply return the final batch response. If

the responses are desired, then another method should be written to use the `response_body_url` to download the responses. From there, the file can be unzipped, which reveals one JSON file per request. It should be noted that Mailchimp will only store the responses for seven days. Furthermore, the `response_body_url` will only last for 10 minutes for security purposes. If the `response_body_url` expires, another can be retrieved via a batch status request using the batch ID.

7.3 Using callbacks to retrieve the status

Since it will take a noticable amount of time to process the requests, the user should be informed of how the request is going. However, when calling the method that does the batch request, control of the program is handed over to this method until the batch is complete. Hence, a callback will be needed to inform the user of the batch's progress. This callback should be called whenever the batch progress changes. Combined with multithreading, this will allow for a program that can simultaneously handle user inputs, output the current status of a batch request, and perform the batch request.

8 Contact Updating

Once the certificates have been uploaded with the file URIs recorded, the attendee list can finally be updated. This can be accomplished via a method in the attendee manager taking in a mapping between the attendees and their certificate URIs, then updating the attendee objects. Once updated, the attendee objects can be saved as a CSV file, overwriting the original attendee record. Lastly, the attendee CSV file can then be uploaded to Mailchimp via a batch request. However, care must be taken as some attendees may not be contacts on our Mailchimp account or existing contacts using a new email address.² The Mailchimp API provides a PUT request for adding or (if already exists) updating a contact, shown below. Once all contacts have been updated, the program can finish.

```
1 import mailchimp_marketing as MailchimpMarketing
2 from mailchimp_marketing.api_client import ApiClientError
3
4 try:
5     client = MailchimpMarketing.Client()
6     client.set_config({
7         "api_key": "YOUR_API_KEY",
8         "server": "YOUR_SERVER_PREFIX"
9     })
10
11     response = client.lists.set_list_member("list_id", "subscriber_hash", {
12         "email_address": "Rick70@gmail.com",
13         "status_if_new": "subscribed",
14         "file": file_uri})
15     print(response)
16 except ApiClientError as error:
17     print("Error: {}".format(error.text))
```

²Mailchimp uses the email address as the unique identifier, so existing members can be added multiple times through different email addresses. This has been an ongoing issue with our members for years.

9 User Interface

9.1 Command Line Interface

For the sake of a prototype, a command line interface (CLI) can be implemented. In Python, the simplest and fastest way of implementing a CLI is through `print()` function calls. While simple, the `print()` function is extremely limited. Most notably, the `print()` function does not allow for the text cursor to be moved backwards, meaning the entire screen must be cleared or enough lines must be written to remove old text. Additionally, the lack of colour coding heavily reduces readability, as successes, warnings, and errors are not immediately obvious. Additionally, displaying boxes and other structured displays is extremely tedious and impractical with `print()`. While it is possible to achieve such feats in a terminal, different terminals use different codes for colour and special formatting, all of which are complex and highly difficult to read and understand. This leads to numerous compatibility issues between terminals.

Going back decades, the `curses` library attempts to partially mitigate the limitations of the humble `print()` function. Implemented in C, Python's `curses` library is simply a set of wrappers calling the C `ncurses.h` functions.³ This alone has two advantages: (1) this makes it extremely easy to transition between C and Python when using `curses`, and (2) the functions are highly optimised. This is on top of the ability to easily colour text, move the cursor around, and draw boxes. While simple in concept, figure 8 below is an example of advanced usage of the `curses` library. Using `curses`, it is possible to easily create GUI-like user interfaces without the complexity of GUI frameworks.

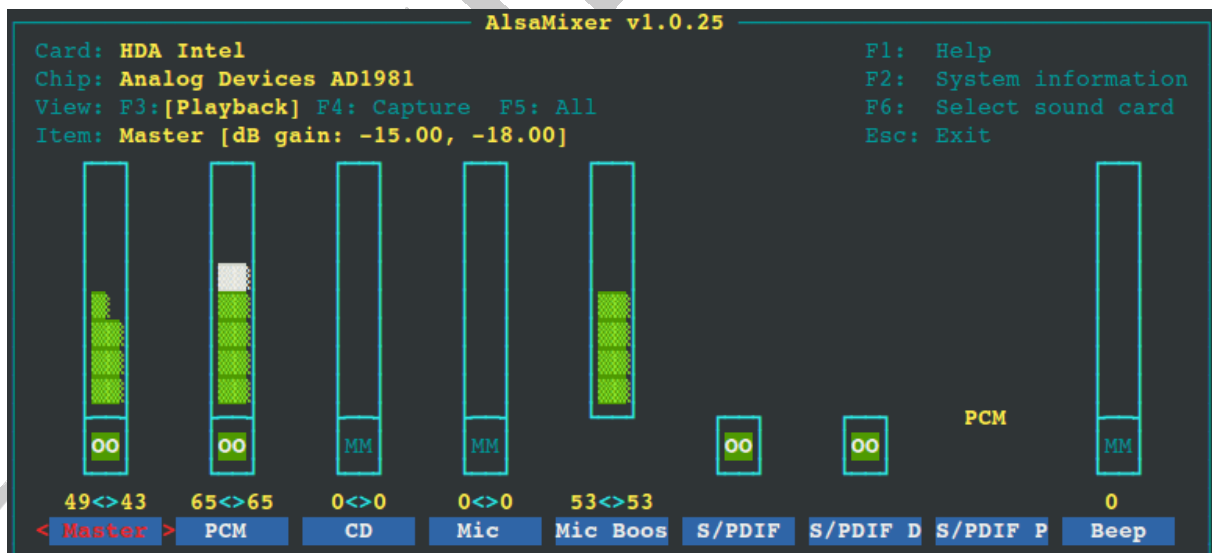


Figure 8: Advanced example of `curses`.

³The “n” is short for new.

9.2 Desktop GUI

While useful for prototyping and developers, CLI have not been suitable for the average computer user for decades. Instead, graphical user interfaces (GUI) are the norm. Whereas CLIs are sequential, always following some order of IO actions in a linear or tree structure, GUIs are event driven. In GUIs, the structure of the interface remains constant until an event occurs, such as the user selecting a button, saying “Hay, Siri”, or a message being received over the internet. GUIs achieve their event driven nature via heavy usage of callbacks and OOP: an object is given the callback, then sits and waits until an event or “trigger” occurs. Once the trigger activates, the object uses the callback to determine what to do. Separating the when (the object) from the what (the callback) allows for reusable and small code.

In Python, creating a GUI for a desktop application requires one of several GUI frameworks. These differ from Python wrappers around older frameworks, such as Qt5 and wxWdigets, to pure python, like Tkinter and PySimpleGUI, and hybrid frameworks, like Kivy. Most frameworks still in use have maintained relevance by finding a niche. The older frameworks implemented in C++ are well-established with online support and documentation going back decades, as well as being highly efficient, running extremely fast with minimal overhead. Newer GUI frameworks made for Python benefit from simpler APIs, encouraging faster development. Choosing a framework is a matter of the product requirements, both for current builds in development and future versions.

By far the most popular, so much so that the option for installing the framework is included when installing Python, is Tkinter. Well known for its simplicity and user friendly GUI builder, Tkinter tends to be a Python programmer’s first GUI framework. While excellent for simple Python projects, Tkinter is restricted to Python, meaning the code and assets cannot be reused for other programming languages. This represents a fatal flaw when attempting to redo a codebase in a different language, as the GUI must be completely redone. Additionally, Tkinter being a Pythonic framework suffers from performance issues relative to older frameworks. Hence, Tkinter and similar Pythonic GUI frameworks are inappropriate for this project.

Kevi is a more interesting GUI framework based on its specialty. Kevi is written Cython, making it a Pythonic framework, making it incompatible with other languages. Where Kevi excels is mobile app development, as its widgets are designed with touchscreens in mind and the native assets of desktop applications are ignored. This makes rapid development of mobile apps feasible once the small learning curve has been overcome. Since the certificate maker program is a desktop app with no mobile version planned, Kevi is also impractical.

These leaves the older GUI frameworks, of which only Qt and wxWdigets have remained competitive. Fortunately, both Qt and wxWdigets have Python wrappers, called PyQt and wxPython. Both are fast and efficient frameworks with multithreading and multiprocessing built-in, with wxWdigets being slightly faster due to its simpler design. Additionally, wxWdigets has a lower learning curve with only just above 20 widgets available. The con, however, means that wxWdigets is not as powerful as Qt. Despite Qt’s far higher learning curve, Qt Designer allows developers to quickly design a GUI without writing a single line of code. Even better, Qt Designer exports GUIs

to .ui files, compatible with Qt across all languages without modification. From this evidence and the given program requirements, a competitive product should use Qt.

9.3 Webapp

With the rise of high-speed internet, most developers have moved away from installable desktop applications and embraced web apps in their place. The benefits of which are immediately obvious: the app can be used without installation of any software, the user's OS is abstracted through the browser, and the code and assets used to create the app can be hidden on the server. From a user's perspective, it is simply far more convenient to use an app immediately instead of waiting for a download to finish, installing the software (taking potentially valuable HDD or SSD space), then have to worry about compatibility issues with their device. A constant internet connection also guarantees updates and announcements can be made immediately, instead of annoying the end-user with "Update available" notifications. Furthermore, advancements in public clouds, such as Microsoft Azure, Amazon Web Services, and Google Cloud, have made it far more seamless for developers to build webapps without needing to build nor maintain private servers. Lastly, the internet connection allows for mass data collection of user behaviours and activities, vital to further app improvements and marketing. From these advantages, it is clear as to why modern developers have made the switch to webapps.

This flexibility, ease of use, and scalability comes at a cost: the constant maintenance required, cybersecurity concerns, and server costs. This is in addition to the setup required, needing a front-end UI, backend databases, an administrator control view, an auditing system, interfaces for third-party services, among numerous other systems. Typically, these systems are built using various programming languages, requiring expert knowledge in numerous protocols. While a for-profit business may be able to afford such costs with full time staff, a student club does not have the finances nor the people to build and maintain such a system.

10 Version Control

10.1 Context

Pre-2002, the Linux OS was extremely difficult to maintain due to its open-source nature. Changes were passed around as patches and archived files, all of which assumed specific versions of the Linux OS were already installed. As Linux grew into numerous distributions, all of which had their own developers with their own patches, it became completely difficult to keep track of what patches needed to be installed. Even worse, the complexity of the patches grow exponentially as distributions split off into multiple distributions. Sometimes, a user should install a patch as it affected the Linux Kernel or the exact distribution installed, whereas other times installing a patch could break the user's system. This is assuming the user can even find the patches to begin with, a task that can prove frustrating and time consuming in of itself.

To better manage the various Linux distributions and their respective patches, BitMover, a for-profit business, allowed the Linux team to use their version control software (VCS) BitKeeper for free. The condition being that Linux developers were not allowed to develop software that competes with BitMover's products. The relationship was uneasy, as many of the pro-open software Linux developers were not happy with using a commercial service by a for-profit business. In 2005, the relationship between the Linux team and BitMover broke down when a developer added commercial features for free. With no VCS to use, the Linux team needed a replacement.

10.2 What is Git?

The replacement came in the form of the open-source software Git. Designed to be lightweight, fast, and simple to use, Git has become the dominant VCS software worldwide. In short, Git allows a team of developers to collaboratively work on a project without the hassles of conflicting code changes, file reserving, nor untracked changes. This is achieved through a tree based version control structure, where each new feature, bug fix, or documentation update is tracked in its own branch of the tree. Within each branch are "commits": snapshots of the tracked files with a short description of what changed and why. Once the update is complete, the branch can be "merged" back into the master (or main) branch. Figure 9 visualises such a workflow.

Git itself can take years to master as there are now hundreds of commands to memorise, each with their own hyper-specific purpose. Fortunately, becoming confident and competent in Git only requires memorising a handful of commands. In order of usage, most of these are

1. `git clone <url> -b <branch>`, clone branch <branch> from the repository <url>
2. `git branch <name>`, create a branch with name <name> from the current branch
3. `git checkout <name>`, switch to the branch <name>
4. `git add <files>`, command Git to start tracking the files <files>

- (a) Once files have been added, they are referred to as "staged"



Figure 9: Simple example of Git workflow.

(b) Wildcards are accepted for <files>, e.g. `git add src/*.py` means add all Python source files in the `src/` directory

5. `git commit`, creates a snapshot of the staged files

(a) Will open a text file in Vim to write a short message about the commit changes

(b) Can skip Vim usage via `git commit -m "changes made"`

6. `git push`, uploads all new commits in current branch to remote repository

Other need-to-know commands include

1. `git pull`, updates current working branch with new commits from remote repository

2. `git branch`, displays all branches in the local repository

3. `git merge <branch>`, combine the history from branch <branch> into current working branch

4. `git merge`, combines remote repository into current local branch

5. `git log`, Display version history for the current branch

6. `git diff <branch1> <branch2>`, compare differences between two branches

7. `git status`, displays the current working branch, how many commits the branch is from the master branch, what files have been staged, and what modified files have not been staged

8. `git revert HEAD`, undos the last commit in the current working branch

(a) Note that HEAD refers to the current working branch

9. `git reset <commit>`, undoes all commits from `<commit>` onwards
10. `git rm <files> --cached`, removes added files from the current branch in the tree
 - (a) Does not delete the files
 - (b) To delete the file(s) and remove it from Git's tree, use `git rm <files> -f`
 - (c) If a file was deleted without using `git rm <files>`, Git will complain about missing files
 - (d) To remove a deleted file from Git's tree, use `git commit -a`
11. `git blame <file>`, who changed what in `<file>` and when

It is **strongly** that readers research Git cheat sheets online, pick one, then use it as a reference.

10.3 Using Github

With the rapid increase in internet speeds, working off-site, and the willingness for companies to let their developers to work from home, hosting Git repositories locally has largely become a thing of the past. Instead, dedicated online Git repositories take the burden of server management off organisations so they can focus on development. The three most popular such services are Github, Bitbucket, and Gitlab. Of these three, Github, owned and maintained by Microsoft, is the most popular, especially for small organisations and personal projects. Regardless, all three use Git as their VCS. The main differences between the three is their business models. Fortunately for CIET, the premium services offered by these three are merely nice to have. Hence, Github shall be used for this project as an arbitrary choice.

With an online repository setup, the best practice manner of contributing to a project is a somewhat convoluted process. To prevent the project Git repository from becoming populated with unnecessary commits and branches, contributors never actually commit directly into the project. Instead, contributors must “fork” the project, creating a copy of the project on their account. With this fork, the contributor can make their commits without affecting the main project repository. Once the desired changes have been made, a “pull request” can be made. These pull requests are requests to merge the modified codebase to the set location, typically the main repository the forked one originates from. When the pull request is made, this notifies the moderators or admins of the main repository. The moderator will then review the pull request by inspecting what has changed. If approved, the pull request becomes a merge so that everyone now benefits from the contributor's changes. Usually, a pull request will result in changes being requested, as it is extremely unlikely that the changes in the pull request will satisfy the moderator's or the repository's pull request standards. Sometimes, the pull request is deleted, as the work required to salvage what good exists is more than redoing the pull request. While convoluted, this process ensures all code is reviewed and held to a necessary standard before being made official.

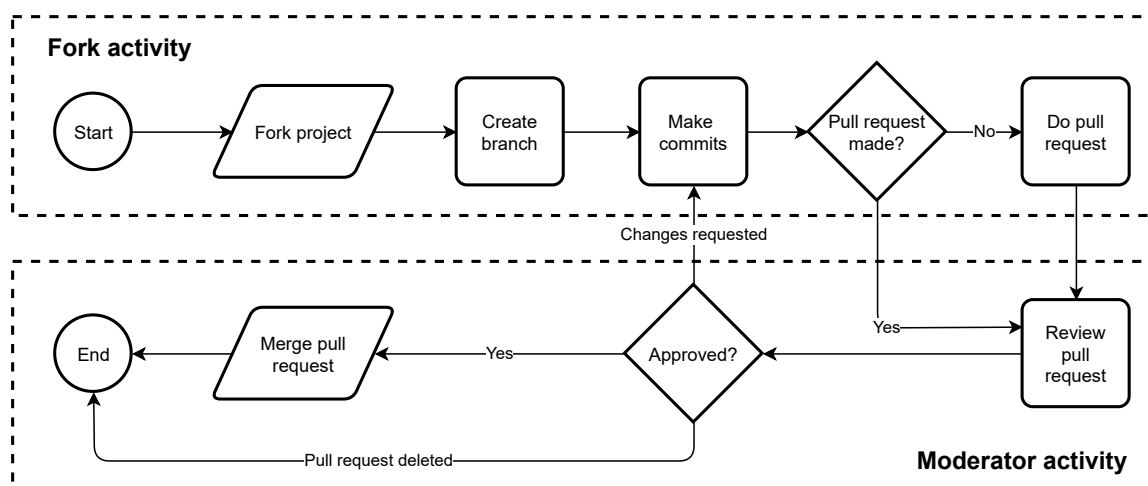


Figure 10: Contributions using Git forks.

10.4 Recording Library Requirements

As a matter of practicality, modern software is built using code already written. This can range from using libraries built-in to the language of choice, all the way to using complex frameworks built on so many other layers of libraries, it becomes impossible to fully grasp just how deep and expansive the code reliances are. Regardless of the quantity of external code used, all libraries and frameworks used must be declared.

Python provides a simple manner to explicitly declare what libraries and frameworks are being used. This is achieved through a `requirements.txt` file: a simple text file where each line represents a library or framework, potentially with the accepted versions. Since multiple versions may be compatible with the codebase, there are multiple ways for declaring which versions are acceptable. A double equals (`==`) means only that specific version is accepted, whereas a greater-than-or-equal-to symbol (`>=`) means any version at least as new as the version specified. If a range of versions is accepted, then a comma (,) can be used to specify a range. A not equal to symbol (`!=`) means any version except for the one specified (useful if that version has a known bug). Most confusingly, the tilde-equals (`~=`) means a compatible version: so any version matching the same numbers excluding the last number, where the version number must be at least as high as specified. This version is useful for specifying a range without two version numbers. Furthermore, the wildcard `*` is useful for specifying any version mathcing the start of the specified version. The text below is an example that uses all the version numbering conventions.

```

1  ##### Requirements without version specifications #####
2  numpy
3  pandas
4
5  ##### Requirements with versions specified #####

```

```
6  tqdm == 4.3.0                # Only version 4.3.0 is accepted
7  requests >= 2.2.0            # Only versions from 2.2.0 onwards
8  opencv-python >= 3.3.0, < 4.0.0 # Only versions from 3.3.0 to less than
   ↪ 4.0.0
9  keyring != 3.5               # Any version, excluding 3.5
10 pytest ~= 6.2.1             # Any version starting with 6.2 and >=
   ↪ 6.2.1
11 django == 3.*               # Any version of the form 3.x.y
```

10.5 Virtual Environments

While installing libraries and frameworks is a necessity for modern software, it quickly leads to bloat and conflicts. Whenever a library or applications is installed, the requirements are also installed. Ergo, even with only a few libraries installed, developers will quickly find their language containing dozens of libraries of various versions. This issue becomes especially problematic when different projects require different versions of the same libraries.

The solution is simple to use but odd to consider. Instead of installing the requirements directly into the installed language, a virtual environment can be made. A virtual environment can be thought of as an isolated instance of the entire programming language for one project. With this isolated instance, libraries and frameworks can be installed without affecting other projects on the same device.

In Python, using virtual environments is handled via the `virtualenv` package. Once installed with `pip` or `conda`, the virtual environment can be created in the current working directory via

```
python -m venv <name>
```

where `<name>` is the name of the virtual environment, typically `venv`. A file system will be created with an activate file. To start the virtual environment, simply type

```
.\venv\Scripts\activate.bat
```

on Windows, or

```
source ./bin/activate
```

on Unix or MacOS systems. From this point onwards, the virtual environment's Python interpreter will be used, as confirmable by running `which python` in the terminal. Since the virtual environment is made for a specific project, the `requirements.txt` file can be used to install all the required libraries via the command


```
pip install -r requirements.txt
```

After running the command above, the virtual environment is now ready to run the project files.

CONFIDENTIAL

11 Coding Standards

11.1 Readability and Maintainability

Any software engineering project will fail before it even starts if coding styles are not established early on. While somewhat subjective, decades of trial and error has revealed multiple key code style principles that greatly assist in the readability and maintainability of code. These are, in no particular order,

1. Identifiers should be short and descriptive.
2. Class names should be in PascalCase.
3. Constant names should be in UPPERCASE with underscores.
4. Function and variable names should be in camelCase or snake_case.
5. Abbreviations and acronyms should only be used when a common convention (e.g. using num for number), or if obvious based on context.
6. The naming conventions should be consistent across the entire codebase and documentation.
7. Short functions are preferred over long functions, even if they lead to more functions.
8. Global variables must be avoided at all costs.
9. Whenever it is not obvious as to what a line or block of code is doing or why it is written the way it is, a comment must be provided explaining the why and what.
10. Comments should only be used for the above situation or providing formal documentation.
11. Indentation should be used to denote nesting where ever possible.
12. A complex line of code involving several operations should be split up into shorter simpler lines.
13. Blank lines should be used to separate blocks of code (e.g. classes and functions).
14. A single space between operators and operands should be used, excluding around parentheses or before a comma.
15. A line of code should be no longer than 80 characters.
16. If a long statement is split across multiple lines of code, the binary operator should start the line; not end it.
17. `import` statements should be at the top of a file, before any other code.
 - (a) From top-to-bottom, the built-in libraries come first, then the third-party ones, then the project libraries.

18. Public methods of a class should come before protected ones, then followed by the private ones.

Furthermore, Python specific style guide principles can also be established:

1. The datatypes of parameters and return types should be written.
2. When using the assignment operator (=) to assign a default value, no spaces should be used.
3. Use four spaces per layer of indentation.
4. The first parameter of an object method should be `self`
5. The first parameter of a class method should be `cls`
6. `... is not ...` is preferred over `not ... is ...`

11.2 Documentation comments with docstrings

Commenting code is a surprisingly controversial subject. On one hand, a lack of comments can lead to vague code that is extremely difficult to understand and maintain. On the other hand, too many comments can lead to code that is annoying to read and maintain, as more text must be read to understand the code and comments may not be updated when the code is changed, leading to misleading documentation. Hence, a balance must be achieved between readability and simplicity.

In Python, docstrings attempt to solve this issue via simple triple quote comments at the start of a class, function, method, or module. A docstring, at a minimum, is comprised of a single line summarises what is being documented. Additionally, all docstrings may also contain a more detailed description, with a blank line inbetween the summary and description. Functions and methods can also have one line per parameter and return value, detailing their names, datatypes, and purpose. Class docstrings should also have the attributes (i.e. the publicly accessible object variables) and public methods detailed. The example below shows a simple example of an incomplete `ComplexNumber` class. Once complete, the `__doc__` property or the `help()` function can be used to display the docstring without finding and reading the source code.

```
1  """
2  Example docstring using a ComplexNumber class
3
4  This module demonstrates the usage of docstrings within Python, following
5  Google's docstring Python Style Guide.
6
7  Todo:
8      * Implement complex number multiplication, subtraction, and division
```

```
9      * Implement complex number angle calculation
10     * Write Sphinx documentation
11     """
12
13     import math
14
15     class ComplexNumber:
16         """
17         This is a class for mathematical operations on complex numbers.
18
19         This class is constant, always creating a new ComplexNumber object
20         when doing operations. Ergo, safe for passing by reference.
21
22         Attributes:
23             real (int): The real part of complex number.
24             imag (int): The imaginary part of complex number.
25         """
26
27         def __init__(self, real=0.0, imag=0.0):
28             """
29             The constructor for ComplexNumber class.
30
31             Args:
32                 real (int): The real part of complex number (default 0.0).
33                 imag (int): The imaginary part of complex number (default 0.0).
34
35             Raises:
36                 TypeError if real or imag are not floats or ints
37             """
38             if type(real) not in [float, int]:
39                 raise TypeError("real part must be a float or int")
40
41             if type(imag) not in [float, int]:
42                 raise TypeError("imag part must be a float or int")
43
44             self._real = real
45             self._imag = imag
46
47             @property
```

```
48     def real(self): return self._real
49
50     @property
51     def imag(self): return self._imag
52
53     def add(self, num):
54         """
55         The function to add two Complex Numbers.
56
57         Args:
58             num (ComplexNumber): The complex number to be added.
59
60         Returns:
61             ComplexNumber: A complex number which contains the sum.
62
63         Raises:
64             TypeError if num is not a ComplexNumber
65         """
66         if not isinstance(num, ComplexNumber):
67             raise TypeError("num must be a ComplexNumber")
68
69         real = self._real + num.real
70         imag = self._imag + num.imag
71
72         return ComplexNumber(real, imag)
73
74     @property
75     def magnitude(self):
76         """Calculate the magnitude, or modulus, of the complex number."""
77         return math.sqrt(self._real ** 2 + self._imag ** 2)
```

12 Testing Code

12.1 Testing tactics

Testing code has historically gone through several methodologies. At first, code was simply tested by running the code and seeing if it works. Such an approach is known as “exploratory testing”, famous for its ease of use, and notorious for missing bugs. Exploratory testing makes the fundamental mistake in testing: testing is *not* about confirming code works, it is about trying to *break* code. Furthermore, exploratory testing is an extremely repetitive and time-consuming task, leading to testers taking shortcuts and being sloppy. As code must be retested for every update, exploratory testing becomes increasingly impractical.

To mitigate the issues with exploratory testing, several testing tactics have been developed. First, testing should be planned into test cases: an independent test case that only checks for one potential failure. With test cases setup, the entire codebase can be thoroughly investigated, diagnosing faults to exact lines of code. Additionally, it is important to note that roughly 80% of bugs are located in roughly 20% of the code. With this in mind, writing test cases is a matter of prioritising where bugs typically appear, particularly boundary value analysis (BVA) and equivalence partitioning. Lastly, the error handling of a program must be tested, not just the successful outputs. With these principles in mind, testing can thoroughly investigate the stability of a program with minimal test cases.

12.2 Historical automated testing using asserts

Even with the minimisation of test cases, manually testing the code every time it changes is extremely time consuming. Hence, programmers created the `assert` statement, allowing for code to be checked automatically. The `assert` statement is a simple function that takes in a Boolean value and checks if it is true or false. If true, nothing happens. If false, the `assert` function attempts to crash the program with an error messaging stating the assert failed, with a line number and the Boolean expression. In older languages like C, the `assert()` function will always crash the program, completely stopping all other subsequent `assert()` statements from being checked. Additionally, inserting the `assert()` into the source code means they will always be executed every time the program runs, slowing the program down. Python has a slightly more advanced usage form of the `assert` statement, allowing for a default error message and, more importantly, throws a catchable `AssertionError`, meaning the error can be processed without crashing the entire program, as shown below:

```
1 # Function to be tested
2 def kelvinToFahrenheit(kelvin):
3     # Test that runs everytime function is called
4     # REALLY BAD, DO NOT USE ASSERTS LIKE THIS!
5     assert kelvin < 0.0, "Cannot be colder than absolute zero!"
```

```
6
7     return (kelvin - 32.0) / 1.8 + 273.15
8
9 # test case 1
10 def test_neg_kelvin():
11     try:
12         fah = kelvinToFahrenheit(-0.00001)
13         assert False, "Negative Kelvin did not throw error"
14     except Exception as err:
15         assert isinstance(err, AssertionError), "Error is not an
16             ↳ AssertionError"
17
18 # test case 2
19 def test_zero_kelvin():
20     fah = kelvinToFahrenheit(0.0)
21     assert abs(fah - 255.37) < 0.01, "0 Kelvin did not convert successfully"
22
23 # test case 3
24 def test_positive_kelvin():
25     fah = kelvinToFahrenheit(150.0)
26     assert abs(fah - 338.71) < 0.01, "150 Kelvin did not convert successfully"
27
28 # run test cases
29 print("Running test cases", end='')
30 for test_func in [test_neg_kelvin, test_positive_kelvin, test_zero_kelvin]:
31     try:
32         test_func()
33         print('.', end='') # notify user test case passed
34     except Exception as err:
35         print('F', end='') # notify user test case failed
```

While simple, `assert` statements are very tedious to work with. This is primarily due to the lack of automated driver code to run the test cases. Instead, the tester must write their own driver code, usually in the form of a `main()` function that runs each test case. For each test case, it is possible that the function could raise an error, meaning the driver code must be able to catch errors and process them without crashing the test script, as the subsequent test cases must run. Furthermore, when expecting the same output for different input values, the test cases either become adundant with repetitive code or complex with nested error handling. Lastly, each test case is executed one-by-one, leading to slow test scripts once the test cases become numerous.

Fixing this involves complex multiprocessing code, making test scripts difficult to write, maintain, and read.

12.3 Modern automated testing using `pytest`

Multiple frameworks exist for Python to make test scripts smarter, thorough, and easy to use. Of these frameworks, `pytest` has proven the simplest and easiest to use without sacrificing test thoroughness. By simply overriding Python's `assert` statement, `pytest` allows developers to write simple code in the form of functions, at least one per test case. Below is a simple script written using `pytest` framework:

```
1  # code being tested
2  import math
3
4  def test_sqrt_0():
5      assert math.sqrt(0) == 0.0
6
7  def test_sqrt_neg():
8      try:
9          num = math.sqrt(-1.0)
10         assert False, "math.sqrt() did not fail when giving negative number"
11     except Exception as err:
12         assert isinstance(err, ValueError), "Exception not an AssertionError"
13
14  def test_sqrt_1():
15      assert math.sqrt(1) == 1.0, "sqrt(1) must be exactly 1"
16
17  def test_sqrt_square_num():
18      assert math.sqrt(9) == 3.0, "sqrt(9) must be exactly 3"
19
20  def test_sqrt_smaller():
21      assert 0.0 <= 0.5 <= math.sqrt(0.5), "sqrt(0.5) must be greater than 0.5"
22
23  def test_sqrt_larger():
24      assert 1.0 <= math.sqrt(5) <= 5, "sqrt(5) must be less than 5"
25
26  def test_sqrt_dec():
27      assert abs(math.sqrt(10) - 3.1623) < 0.0001, "sqrt(10) should be ~3.1623"
28
29  # This test will fail
```



```

30 def test_sqrt_fail():
31     assert math.sqrt(4) == 4, "sqrt(4) must equal 4"

```

Note that every function starts with `test_` in their name. This is required for `pytest` to recognise the function as a test case that needs to be run. To run the test script, type `pytest <filename>` in the shell. For the example above, the output will be

```

$ pytest pytest_simple.py

===== test session starts =====
platform win32 -- Python 3.8.8, pytest-6.2.4, py-1.10.0, pluggy-0.13.1
rootdir: $TEST_SCRIPT_PATH
collected 8 items

pytest_simple.py .....F                                     [100%]

===== FAILURES =====
----- test_sqrt_fail -----

    def test_sqrt_fail():
>         assert math.sqrt(4) == 4, "sqrt(4) must equal 4"
E         AssertionError: sqrt(4) must equal 4
E         assert 2.0 == 4
E         + where 2.0 = <built-in function sqrt>(4)
E         +   where <built-in function sqrt> = math.sqrt

pytest_simple.py:31: AssertionError

===== short test summary info =====
FAILED pytest_simple.py::test_sqrt_fail - AssertionError: sqrt(4) must equal 4
===== 1 failed, 7 passed in 0.04s =====

```

Additionally, `pytest` also has several advanced features to allow for smarter design of test scripts. If the exact same input data is used for multiple test cases, a fixture can be used.

```

1 import pytest
2
3 @pytest.fixture
4 def input_value():
5     input = 39
6     return input
7

```

```
8 def test_divisible_by_3(input_value):
9     assert input_value % 3 == 0
10
11 def test_divisible_by_6(input_value):
12     assert input_value % 6 == 0
```

Similarly, if multiple sets of data need to be tested against the same test, a parameterised test can be used:

```
1 import pytest
2
3 @pytest.mark.parametrize("num, output", [
4     (1, 11),
5     (2, 22),
6     (3, 35),
7     (4, 44)]
8 )
9 def test_multiplication_11(num, output):
10     assert 11 * num == output
```

Lastly, pytest can run test cases in parallel by installing the `pytest-xdist` plugin, then running pytest with an `-n` flag, specifying the number of processes to be created. For example,

```
pytest test_script.py -n 6
```

will run `test_script.py`, splitting the test cases across six processes. pytest also supports third-party plugins that add numerous useful features, the ability to skip failed test cases, and categorising test cases so that only one category of test cases gets executed. Hence, this project should use pytest where ever possible.