Secure Communication Platform Project Report

Liliana O'Sullivan

C00227188

Submitted in partial satisfaction of the requirements for the degree of

BSc of Science (Hons) Software Development

Faculty of Science

Institute of Technology, Carlow

27th April 2021

Table of Contents

List of Figures

Chapter 1

Introduction

This report outlines the overall learning experience throughout the project, in a personal and technical form. An outline of the delivered content, specification and design, and an overall review of the project. The project will be referred to as *Ilo*. Ilo was a name chosen for the project. It is a translation of ‘tool’ in Esperanto; Communication, private or public, is a fundamental tool of an individual.

Chapter 2

Description of Submitted Project

The entire project comes together as a platform for sending end-to-end encrypted messages. The backend workload is split among two servers created in Python and Elixir to use each technology in their strengths; Python for rapid development and Elixir for high-concurrent throughput with supervised fault-tolerance.

A desire of the project was the usage and architecture aimed to maximise the potential of each technology used. The technology choice quite quickly became Elixir. However, leveraging Python technologies was also desirable; The challenge came with using them in tandem. Direct Unix-pipe style inter-process communication can be possible, though it limits Elixir from being distributed without a Python partner. Using a database as a buffer is possible, though introducing the overhead of a 'third-party' and Cassandra as a database will not allow for consistent information; Cassandra employs eventual consistency through distribution, meaning data can and will get out of synchronisation. A messaging broker such as RabbitMQ, Celery, Mosquitto or ActiveMQ introduces third-party overhead and complication to the system. The solution was inspired by Enterprise Single Sign On (SSO).

SSO redirects a user's browser to the SSO provider, often places an authenticated random alphanumeric passcode within the user's browser that can be used to verify authentication. Once a client makes a request to Elixir, Elixir will behind the veils makes a while-listed HTTP call to Python, which verifies the authenticity of the request. The elegance comes in the form that once a request is authenticated, Elixir can proceed without being bottle-necked by Python overhead. At any point, Python could 'message' Elixir to revoke a client. Elixir can be distributed and handle concurrent requests with a one-request overhead.

Elixir and Python were architected as independent servers. The Python server uses FastAPI and handles the none-message sending tasks, such as the creation of users. It is also used to document the Application Programming Interface (API) using OpenAPI and JavaScript Object Notation (JSON) schema standards.

Elixir is predominantly a Web-Socket server that handles the sending of messages. Within the server, each authenticated user is an Elixir process that is stored separately for each 'room'. This is created using CowBoy.

Additionally, a reference client was created to showcase how a client may use the API to send messages. This was created on the Tkinter GUI technology within the Python standard library.

The submitted ZIP file contains three folders.

- Python
- Elixir
- Docs

The Python folder contains the FastAPI server and the reference client within the *python/pyclient* folder. The Elixir folder contains the CowBoy Web-Socket server.

The docs folder contains MkDocs based documentation. MkDocs is a documentation tool based created on Python technologies. It allows a developer to write Markdown that will be transpiled to a HTML web page. A developer retains the ability to apply customisation to the structure and styling, with additional functionality that can be implemented through JavaScript. This folder contains the source-markdown used behind each page. It also includes the *mkdocs.yml* file that specifies how the overall documentation should function.

2.1 Technologies

This section discusses the technologies within the project and how they fit into the project development.

2.1.1 Python

Python comprises three essential components of the project; It is used to power one of two servers. It is used as a backend for the reference client implementation and is used to transpile and deploy the developer documentation.

2.1.2 Elixir

Elixir is used as a messaging handling service to send message to connected users concurrently. As the sending of messages is a lightweight and highly concurrent task, the work was sourced to the BEAM virtual machine. While Erlang additionally suitable for this, Elixir was chosen for the modern idioms and syntax provided.

2.1.3 HTML/CSS/JavaScript

In the development of a reference client, the intent was to leverage the existing implementations provided by a web browser. The intention was to leverage the browsers existing cryptographic primitives with the native WebSocket to develop a reference client. As development progressed, the cryptographic primitives became problematic, and as such, this codebase was deprecated in place of the Python reference client. The code remains for legacy purposes.

2.1.4 Web-Sockets

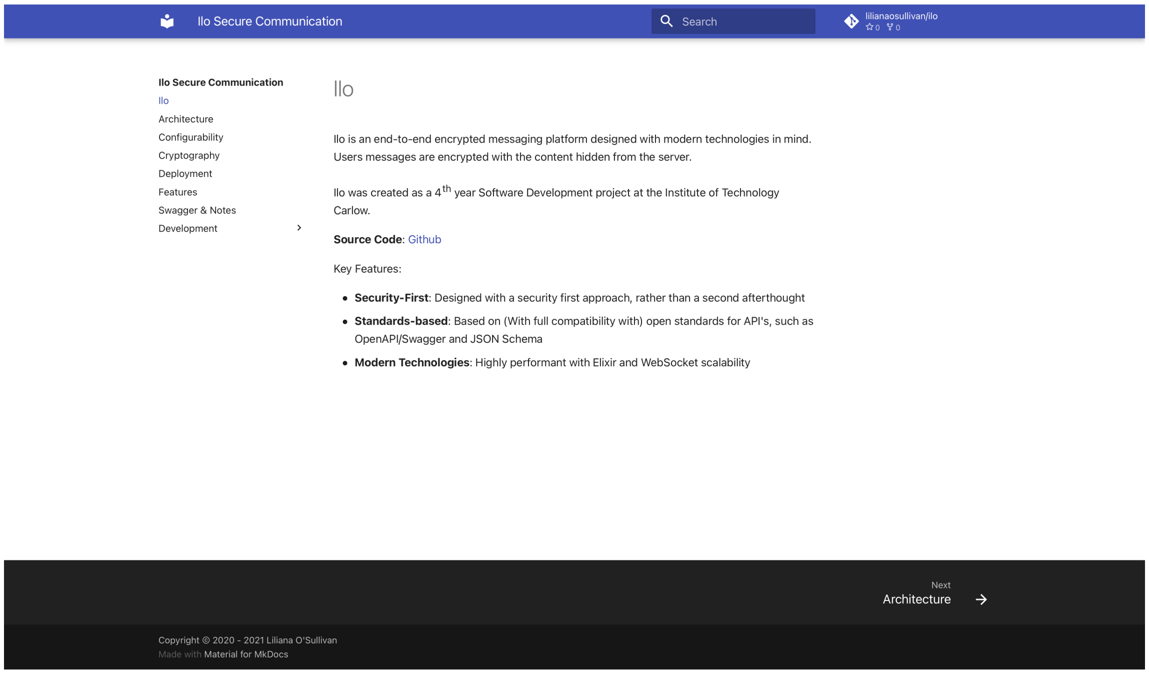
This is the technology used as a connection between the client and the server. It allows the client or the server to send information at any point. It can be likened to a consistently open pipe; A client or a server can send information through this pipe without requesting an update, unlike in the HTTP client-server architecture.

2.2 Developer Documentation

The developer documentation was created using the Python-powered *Mkdocs* project. Mkdocs is a static site generator that enables developers to write Markdown based pages that are transpiled to HTML. A developer can customise the site being generated through a YAML-based file that serves as the documentation base. Themes can be created to customise the appearance and behaviour of a site, with additional optional parameters configured through the YAML configuration. To exemplify this, the source markdown behind the documentation homepage can be found in figure 2.1 with the transpiled result in figure 2.2.

```
# Ilo

Ilo is an end-to-end encrypted messaging platform designed with modern
↪  technologies in mind. Users messages are encrypted with the content
↪  hidden from the server.<br><br>
Ilo was created as a 4<sup>th</sup> year Software Development project at
↪  the Institute of Technology Carlow.

**Source Code**: <a href="github.com">Github</a>

Key Features:

* **Security-First**: Designed with a security first approach, rather than
↪  a second afterthought
* **Standards-based**: Based on (With full compatibility with) open
↪  standards for API's, such as OpenAPI/Swagger and JSON Schema
* **Modern Technologies**: Highly performant with Elixir and WebSocket
↪  scalability
```

Figure 2.1: Source of homepage of documentation


Ilo Secure Communication

Search

lilianaosullivan/ilo

Ilo Secure Communication

Ilo

Architecture

Configurability

Cryptography

Deployment

Features

Swagger & Notes

Development

Ilo

Ilo is an end-to-end encrypted messaging platform designed with modern technologies in mind. Users messages are encrypted with the content hidden from the server.

Ilo was created as a 4th year Software Development project at the Institute of Technology Carlow.

Source Code: Github

Key Features:

- **Security-First**: Designed with a security first approach, rather than a second afterthought
- **Standards-based**: Based on (With full compatibility with) open standards for API's, such as OpenAPI/Swagger and JSON Schema
- **Modern Technologies**: Highly performant with Elixir and WebSocket scalability

Next
Architecture

Copyright © 2020 - 2021 Liliana O'Sullivan
Made with Material for MkDocs

Figure 2.2: Documentation homepage

Chapter 3

Description of Conformance to Specification and Design

The project maintained a high degree of consistency to the specification created. A deviation from the original specification was the usage of ‘rooms’ within the system. The original specification drafted a system where a user could directly message a user, or create groups for multiple users. The delivered system instead uses multiple rooms by default; This room can attain one or multiple users. Each user is maintained as an Elixir process.

Chapter 4

Description of Learning

This chapter outlines the learning and the overall experience within the project life-cycle. Split among two sections: Technical and Personal. Technical will outline direct programmatic skills and knowledge attained; Personal illustrates soft skills attained.

4.1 Technical

4.1.1 Web-Sockets

Web-Sockets overall seem like an appropriate fit for the Ilo platform, as the Ilo servers require a full-duplex connection while benefiting from the lightweight nature. A server must have the ability to receives messages that are being sent while also retaining the ability to forward any messages it receives from other users.

Web-Sockets, in my experience, seemed to be lacking comprehensive support for client development in a none-browser or JavaScript environment. Web-sockets seemed to be a less-travelled area collectively. Building the Python reference client and finding mature web-socket libraries with matching documentation posed a greater challenge than initially anticipated.

4.1.2 Python

Python is an essential technology to the Ilo platform, making up 63.4% of the codebase. As part of the two interconnected-server architecture, Python is the base of one of the servers. Additionally, through the Tkinter bindings as part of the standard library, the reference client in part was created with it. The documentation, while not created in Python, is a project that uses Python as a backend.

One of Python's strengths lies in the productivity it provides the developer. A single line of Python can loosely be equated to multiple lines in a lower-level language such as C++ or Java.

FastAPI

FastAPI came from research into Python web frameworks. The aim was to find a modern high-performance framework. During this research, FastAPI was reassuringly mentioned in recent publishings. This is due to FastAPI being relatively new in comparison, with the initial public commit dating the fifth of December 2018. For comparison, Flask's initial source publishing came on the sixth of April 2010 and Django, another popular web framework that the Python Software Foundation uses for Python.org, had its debut on the thirteenth of July 2005.

FastAPI provided an intuitive experience, closely related to Flask, while provided increased response performance for the intended work, with modern Python features such as utilising type hinting. FastAPI provided the ability to document the codebase. This came as a surprise.

One of the project outcome desires was the usage of technologies that could handle high throughput, ideally with the usage of modern technologies such as Rust, Python, Go, NodeJS or Elixir; Without resorting to the C++ Drogon framework. FastAPI fit this. It serves to show that modern Python can be intuitive and performant.

Tkinter

Tkinter is a library bundled with the Python Standard Library that provides bindings to the Tcl's language Tk Gui Toolkit. It is the technology behind the reference client. These bindings were relatively accessible. Parts of the bindings are showing their age. Parts of Tkinter have remained untouched for multiple years, some spanning 12 years. For example, doing a paste command with a Unicode into the entry when text already exists causes Tkinter to misbehave, sometimes needing an application restart.

Tkinter overall provided experience in GUI development. My experience mostly came in C# WinForms from work experience, and Java Swing from Dr Jason Barron in the second year of the Software Development Course at the Institute of Technology, Carlow.

4.1.3 API Development

API development is more challenging than originally anticipated. While it can be painless to create something that may seem intuitive to you, this may not be towards other developers. An example of this is HTTP methods/actions such as POST, GET, HEAD etc. Each method has a generally associated usage. Some may seem clear from the name, such as GET or DELETE, while others are intuitive from experience, such as using POST to create or update a resource. Others may not seem as evident, for example, OPTIONS or PUT. Learning and conforming to these common practices and standards aids in creating a more intuitive final product.

Additionally, standards existed that I was not aware of, such as JSON Schema. Additionally, the OpenAPI Specification (Formally known as Swagger) I was loosely aware of by name but not in their application or creation.

API Tools

When I started this project, I was aware exclusively of Postman as a tool to manually send requests, without having used it. The intent at the beginning was to investigate the possible use of these tools. If they seem undesirable for any reason, I can default to using *cURL* or a web library such as Python Requests. I was not satisfied with the push to create an account on Postman or the proprietary codebase for sending HTTP requests. I discovered the Insomnia client, effectively Postman as an open-source client.

Using Insomnia provided a learning experience and aided in API development. It provides intricate controls of the request being made, such as the HTTP method being used (Including using a custom type), supports multiple authentication types, control over the header being sent, control over the request body and much more. As a nice-to-have feature, it provided the time each request used. While not essential, it aids in being mindful of performance and the overall state of the application.

4.1.4 Elixir

Within this project, Elixir served as a companion to FastAPI. Elixir is used to offload the highly concurrent workload of handling Web-Socket connections away from FastAPI by running a CowBoy Web-Socket server. While FastAPI has native support for Web-Sockets, Elixir is capable of handle the workload much more efficiently. This CowBoy server stores each connection as a process; once a user sends a message through the Web-Socket, Elixir will forward it to all the other processes in the appropriate room.

I found Elixir to be a unique development experience from many languages I have come across. While compiling to the same BEAM VM codebase as Erlang, similarities can be found between the two. For example, atoms are used in similar manners while having syntactical declarative differences. Elixir atoms begin with a colon and support mixed cases, while Erlang atoms often begin with a lowercase letter. This can be seen in figures 4.1a and 4.1b.

```
iex(1)> is_atom(:my_atom)
true
```

(a) Elixir Shell – atom showcase

```
1> is_atom(my_atom).
true
```

(b) Erlang Shell – atom showcase

Figure 4.1: Erlang versus Elixir – atoms

Elixir provided many modern features, such as the usage of Ruby syntax, named parameters or the ability to pipe output from one function to the next. The Pipe operator was unique and yet intuitive. Within the project, an IP address is automatically split among tuples by Cowboy. An IP address with a port such as `25.191.1.2:7212` would be stored as `{{25,191,1,2},7212 }`. While Cowboy can make sense of this, processing this can be problematic. Using the pipe operator, `|>`, it was possible to format this into a string of solely the IP Address without the port. This code extract is shown in figure 4.2.

```
elem(request.peer,0) |> Tuple.to_list |> Enum.join(".")
```

Figure 4.2: Minted - line numbers' inside frame

Elixir usage within the project was demanding. It provided a unique experience that a more traditional such as C# would not have been capable of providing.

4.1.5 Testing

While testing the application multiple learning experiences came about. Pytest is used as the main testing framework. Pytest itself while having a mildly heigher learning curve than initially anticipated, showcased its elegance. The use of fixtures and automatic test discoverey provided a much cleaner and more logical codebase that provided a focus on the test content without the need to using boiler plate code.

Coverage

Coverage can be a metric used to verify the tests are covering a the intended amount of code and possibilities, either through line or branch coverage. Initially in the testing of the application, running a line coverage check resulted in over sixty percent of the code being covered yet most of the code has not been tested. This was due to the code being run in startup or often code that is executed but has no runtime effect, such as documentation strings. Certain files also had minimial lines of code, some had zero such as python's ___*init*___*.py* files. As these files were executed, it resulted in one hundred percent coverage, greatly raising the average coverage of the application as a whole.

Mutation Testing

I desired to try mutation testing as, in my view, it is a interesting and unique form of testing that can almost test the quality of tests, or the sensitivity to change. It loosly can be described as what coverage is to code; Coverage ensures the code and it's paths are being executed, mutation testing tests the tests, behaviour has changed, a test should catch it.

Once I choose a mutation testing library to use, mutmut was the tool of choice, the first execution quickly brough disappointing results; Much of the code being tested was either irrelvant to a mutation, such as the changing of a documentation string, or the files being mutated entirely were not supposed to be tested; The test files should not be mutated, are tests supposed to have tests?

Going through the documentation of mutmut, it mentioned the ability to create advanced whitelisting that could allow running of a python created function before any mutation, enabling insights such as what line number is being tested, the filename, the source code line, the ability to skip this line and more. This became problematic as it seemed the function did not execute. After some investigation through GitHub issues, Stack Overflow and exploring and forking the source code of mutmut, I created a test scenario checked the exact values this pre-mutation function was recieving. As mutmut is utilising concurrency, certain functions such a breakpoint were not possible, resulting in a broke pipe exception by the Python interpreter. The output came in using a write without a with, this can be seen in figure 4.3, with a singluar write out seen in figure 4.4. The point of highlight came in the filename; It was named *./CassandraModels.py*. Mutmut filename provided was a relative path, not a filename which cause a confliction in the mutmut source code to ignore tests. This was resolved for the project by splitting at the '/' and taking the last element providing the true filename.

```python
f = open("mut.txt", "a")
def pre_mutation(context):
    ignore_list = [
        "gui_layouts.py",
        "gui.py",
        "helpers.py",
        "mutmut_config.py",
        "conftest.py",
        "CassandraModels.py",
        "test_apikeys.py",
    ]
    filename = context.filename.split("/")[-1]
    if filename.startswith("test_"):
        context.skip = True
    if filename in ignore_list:
        context.skip = True

    if context.skip == False:
        f.write(f"{str(context.__dict__)}\n")
```

Figure 4.3: mutmut_config.py

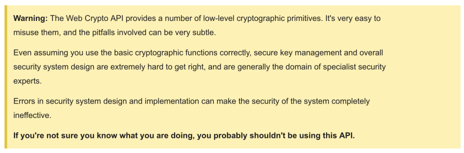
```
{
    "index": 11,
    "remove_newline_at_end": False,
    "_source": "import time..",
    "mutation_id": MutationID(line="logged_in = columns.Text(required=False,
    ↪  index=True)",
    index=2, line_number=13, filename=./CassandraModels.py),
    "performed_mutation_ids": [],
    "current_line_index": 13,
    "filename": "./CassandraModels.py",
    "stack": [],
    "dict_synonyms": [
        "",
        "dict"
    ],
    "_source_by_line_number": None,
    "_pragma_no_mutate_lines": None,
    "_path_by_line": None,
    "config": <mutmut.Config object at 0x1061b3fa0>,
    "skip": False
}
```

Figure 4.4: mutmut write out

4.1.6 Cryptography

The Cryptography behind the project was the most challenging aspect of the project. This became most apparent once the reference client was being developed in the browser. The first challenge encounter came in accessing the cryptographic primitives required. Within the browser, this is known as the Web Crypto API. The browser required the use of a HTTPS connection to the server to enable access. This was resolved through the use of a self-signed certificate for the *localhost* domain. This seems counter-intuitive, as a certificate aims to verify the authenticity of a domain. In this scenario, neither a domain existed, nor was the authenticity being verified; the certificate was self-signed. With access to the WebCrypto API, the management of generated keys became problematic such as exporting the private and public keys of an RSA algorithm. The browser does not support exporting of these keys, with included protections from serialising these objects. While this may not impact many users, this prevents key persistence. My intuition is the lack of demand around end-to-end encryption leaves this as an area of improvement.

Additionally exploring documentation around Cryptography will often lead to unwelcoming messages as shown in figure 4.5a and 4.5b. While cryptography is an acquired skill, a developer must start at a point, and telling developers not to use these API's can be a contributing factor around the 'fear' or 'stigma' of cryptography among developers. If developers were consistently warned that concurrency within their applications is difficult and "pitfalls involved can be very subtle" and that "If you're not sure you know what you are doing, you probably shouldn't be using this..", developers would consistently be increasingly cautious around concurrency, even if it can directly benefit many applications.



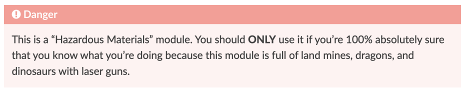
Warning: The Web Crypto API provides a number of low-level cryptographic primitives. It's very easy to misuse them, and the pitfalls involved can be very subtle.

Even assuming you use the basic cryptographic functions correctly, secure key management and overall security system design are extremely hard to get right, and are generally the domain of specialist security experts.

Errors in security system design and implementation can make the security of the system completely ineffective.

If you're not sure you know what you are doing, you probably shouldn't be using this API.

(a) Mozilla Web Docs – WebCrypto Warning



Danger

This is a "Hazardous Materials" module. You should **ONLY** use it if you're 100% absolutely sure that you know what you're doing because this module is full of land mines, dragons, and dinosaurs with laser guns.

(b) Python Cryptography Package – Warning

Figure 4.5: Cryptography Warnings

4.1.7 Databases

In this project, a NoSQL database was chosen for these two reasons, a lack of a schema and high throughput. MongoDB is one of the most popular NoSQL databases, with schema-less JSON-like documents; it allows developers to store almost any type of information into a document (A MongoDB equivalent of a row). This can be essential when developing with an unknown schema. MongoDB's dynamic design can be a powerful combination with dynamic languages such as JavaScript, Python or Ruby. JSON is designed to be a method of serialising and deserialising objects, which allows dynamic languages to map documents to MongoDB with relative ease, especially as objects themselves can vary greatly. As dynamic languages are dominant in the Ilo platform, MongoDB was a natural fit for development.

Cassandra

I was initially charmed by the raw throughput and horizontal scalability that Cassandra is capable of. Additionally, it would be a potent combination of Elixir with Cassandra; it opens the possibility for the technology stack to be distributed and to scale horizontally as required. Initially, Cassandra was difficult to understand how to create a schema (This is called Modelling within Cassandra) and how to query a table once it is created. Cassandra can be loosely likened to SQL without the relational features, such as JOIN or GROUP BY. This increases the difficulty of creating an efficient table schema. The project for most of the life-cycle was using MongoDB with a transition to Cassandra once the table schema became less volatile.

Additionally, Cassandra does not allow a table column to be queried without first knowing the primary key of the desired row. This can become problematic as not all queries are desiring a specific row, for example, a query to check if a username exists or all ages that are greater than thirty. This was resolved by applying an index to the required columns.

4.2 Personal

4.2.1 Meeting Skills

As the project began, with meetings with the supervisors, learning how to use the meetings effectively is a developed skill. Knowing how to ask questions, what questions to ask and preparing for meetings became a skill. While it is difficult to quantify, the productivity of the meetings progressively felt to have increased as I learnt the structure and timing management associated with the meeting. As meetings progressed, a plan for each meeting began to form with prepared questions that accrued between the meetings. This aided in making efficient use of a meeting.

4.2.2 Documentation Creation

As this project is aimed towards developers, documentation is essential. A question also arises, *how does one document a project/service/package effectively?* My first intuition came in the form of a documentation generator such as PyDoc, Doxygen, JavaDoc etc. I decided to research how does tools and packages I use document their codebases.

I soon came to discover documentation generates that use markup to generate documentation. This was an intuitive solution to the problem at hand; the documentation can be visually appealing, searchable and even scale across devices while being intuitive to create through markup. I reduced my choices to either Sphinx or MkDocs. Sphinx uses reStructuredText, which became popular among the Python community, and it is used to create the documentation found at Python.org. MkDocs aimed to provide similar functionality, with markdown instead of reStructuredText. As I am more familiar with markdown, and with the greater adoption of markdown, it became my documentation tool of choice.

The experience of documenting with MkDocs was quite natural and relatively frictionless. It empowers developers to write documentation, while a designer can create intuitive and appealing themes, similar to the philosophy behind the LaTeX system.

4.2.3 LaTeX

This is a document preparation system that enables a writer to use markup-convention plain text that is compiled to a chosen output format such as PDF. It is prevalent in the scientific and mathematical communities due to the typesetting of formulas and mathematical equations. This is a tool I had a desire to investigate and apply. I had concerns about switching; what if one or multiple concepts became problematic midway through the document? With a desire to give it a try, I followed the tutorial provided by 'LaTeX-Tutorial.com' and learnt enough to feel comfortable to apply it.

This document is the first complete LaTeX document I produced, and beginning this project again, I would have begun with LaTeX from the beginning. The output produced by the documents is of a higher typesetting quality compared to the documents I was capable of producing using a WYSIWYG (What You See Is What You Get) system such as Microsoft Word or Apple Pages.

Challenges did arise from switching. A problematic issue came when creating figures and tables. Once created, they were placed into seemingly arbitrary locations, often outside the intended chapter or section in which they are created in. This raised concern for some time until I recalled image positioning from 'LaTeX-Tutorial.com'. It specified how to position an image by specifying a positioning hint at the beginning of an environment. Specifically, the following options

- h (here) – Relatively in this location
- t (top) – Top of the page
- b (bottom) – Bottom of the page
- p (page) – On a new page
- ! (override) – This attempts to force the specified location
- H – This is an additional option provided by the float package that is stricter than the combination of *h!*

I applied the 'H' position modified, which yielded the desired position for figures and tables.

A secondary example of a challenge came in placing code blocks. Once learning that the 'minted' package is commonly used to syntax highlight code, adding line numbers yielded an unexpected result when combined with the built-in frame functionality; The line numbers are by default placed outside the frame. This behaviour can be seen in Figure 4.6

```
1  from locust import HttpUser, task
2  class QuickstartUser(HttpUser):
3      @task
4      def test(self):
5          self.client.post("/", data={
6              "username": "admin",
7              "password": "password"
8          })
```

Figure 4.6: Minted - default line number and frame behaviour

To encapsulate the line numbers within the frame, a negative ten-point 'number separator' had to be applied, and the code manually indented by a single tab space. The result of this can be seen in Figure 4.7.

```
1   from locust import HttpUser, task
2   class QuickstartUser(HttpUser):
3       @task
4       def test(self):
5           self.client.post("/", data={
6               "username": "admin",
7               "password": "password"
8           })
```

Figure 4.7: Minted - line numbers' inside frame

The usage of LaTeX to create the documents came as an unexpectedly enjoyable experience. Once returning to a WYSIWYG system where necessary came the desire to return to LaTeX. The overall quality of the final document increased, and once development comfort increased with the system, productivity increases soon followed. A system that allows producing more in less time, with higher quality output, is always appreciated. Who does not like to code their documents?

Chapter 5

Review of Project

This project began by choosing the technologies and designing an architecture that would be efficient and practical. The architecture of a two server setup and their communication mechanism inspired by Enterprise Single Sign On (SSO) was suitable and fit the requirement. The technologies chosen are a suitable fit.

Retrospectively, Elixir was more demanding than initially anticipated. This came primarily in a lack of learning material and resources to turn to once troubleshooting an undesired occurrence. I believe this to be the case due to the lack of popularity at the time of writing surrounding Elixir; Elixir ranks forty-eight on the Tiobe Index, with Erlang absent from the top fifty. Additionally, as a language, it is substantially distinct in the feature set it provides and the core functionality of the language. The functional programming model is unconventional in an Object-Oriented dominant environment.

FastAPI, and the use of Python in general, was an adequate fit. Overall I found myself asking 'How should it behave' instead of 'How to create this'. This is the highlight of Python, the development was dominated in architecting the behaviour instead of a confrontation of technical challenges. This was the original intent behind integrating Python technologies into the project; It allows developers to optimise for the most precious resource in the software development process; The human time used in the creation of software.

A place of technological oversight came within the development of the client. While complications were expected, the obstacles that came up were not expected. The client initially began development as a HTML web page with the expectation that a browser would allow rapid development through the native Web-Socket support. This retrospectively was the initial obstacle as cryptographic key persistence can become problematic. The second came when attempting to access and employ native cryptographic functions. To access any cryptographic functions in a browser, a HTTPS connection must be established. This was swiftly resolved with a self-signed Transport Layer Security (TLS) certificate. With the functions available, it soon became apparent that the scope and variety of functions were too limited for the scope of the project during their usage. For example, the exporting of RSA generated keys was not supported. It became clear the browser was not the ideal technology for this, and a native application was a superior fit. This became the driving force to developing the Tkinter-based desktop client that now resides within the *"python/py_client"* directory.

The transition of MongoDB to Cassandra remained outstanding for most of the project. Once experimenting with technologies at the beginning of the project, Cassandra proved to be demanding in designing the database schema. It can be likened to Structured Query Language (SQL) without the ability to join tables or the ability to auto-increment primary keys with columns being out of sync due to the distributed architecture. The project development began with MongoDB as it has excellent integration with native Python through the PyMongo driver and allows the schema to be developed as the project evolves. A core strength of Cassandra is the write, and a compelling but not as impressive read, performance. The transition to Cassandra was not as difficult as anticipated initially once the schema was well defined. An issue that arose is the inability to query a column (Cassandra queries columns instead of tables) by default. This was fixed by indexing the column. A place of potential expansion is the transition to Scylla. Scylla is the entirety of Cassandra implemented in C++ instead of Java. It has complete compliance with Cassandra APIs, allowing existing Cassandra tools and drivers to be used. Scylla aims to provide additional throughput through Cassandra by removing the interpreted nature of Java and the need for a garbage collector.

Chapter 6

Acknowledgements

I would like to recognize the invaluable assistance of Paul Barry as project supervisor for his continued support within the project process. Paul guided the project throughout its life-cycle, consistently taking the time to respond to any queries or concerns. His knowledge and expertise were invaluable in the project; While I had not been picking Paul's technical insight, he provided consistent support throughout our time together. It was truly a pleasure.

I want to extend a heartfelt thank you to Derry Brennan for his continual support by providing consolation and patience; Notably in listening to technical ramblings. All software is created by human developers. The people supporting these developers are often unsung heroes; Derry is this kind of hero.

The entire Python community deserves acknowledgement for their continued support of the language and its packages and their assistance for new developers and seasoned professionals. This project used tools such as Jedi, FastAPI, PyYaml, MkDocs and more, that were created and refined with the community's efforts.

Finally, I want to acknowledge the help of lecturer Richard Butler for continuously taking the time to clarify technical security concepts.