UNIVERSITY OF CAPE TOWN

DEPARTMENT OF COMPUTER SCIENCE

Computer Science Honours
Final Paper
2017

Title: Honours Lab Locker Controller

Author: Norman Pilusa

Project Abbreviation: Lockit

Supervisor(s): Gary Stewart

Category	Min	Max	Chosen
Requirement Analysis and Design	0	20	15
Theoretical Analysis	0	25	0
Experiment Design and Execution	0	20	0
System Development and Implementation	0	15	15
Results, Findings and Conclusion	10	20	10
Aim Formulation and Background Work	10	15	10
Quality of Paper Writing and Presentation	10		
Quality of Deliverables	10		
Overall General Project Evaluation (*this section allowed only with motivation letter from supervisor*)	0	10	
Total marks		**80**	

Lockit: Honours Lab Locker Controller

Norman Pilusa

Department of Computer Science

University of Cape Town

Rondebosch 7701

Cape Town, South Africa

plsnor001@myuct.ac.za

ABSTRACT

Public areas like universities often have lockers for students to store their belongings. However, there is only a limited number of lockers which is often less than the number of students needing them. This imbalance requires an administration process that will ensure that everyone gets a chance to use the lockers. This paper presents a webserver for managing lockers installed in the computer science honours lab at the University of Cape Town. The webserver is developed as a software component of an embedded system solution. The webserver forms a middle ware between user interfaces and a Raspberry Pi that controls the locking mechanisms installed on the lockers.

KEYWORDS

Application Programming Interface (API), Uniform Resource Locator (URL), Webserver, Security

1 INTRODUCTION

The University of Cape Town's computer science department has installed new lockers for its honours students. The honours class in 2017 was 60 and it is expected to increase every year.

1.1 Problem Statement

Currently the number of lockers installed is 30. Thirty lockers are not enough for the honours class. This makes it necessary to have a management system for the lockers so that every student gets an opportunity to use a locker. Managing means allocating time to students to use the lockers, monitoring over use and penalizing students accordingly.

1.2 Solution Outline

A low cost embedded system solution was built to manage and monitor the use of the lockers. The system is made up of three components: a Raspberry Pi that controls locking mechanism on the lockers, a webserver for managing and monitoring uses, and user interfaces that students interact with. The break-down of the solution can be seen in figure 1. The components were developed using the embedded design life cycle.

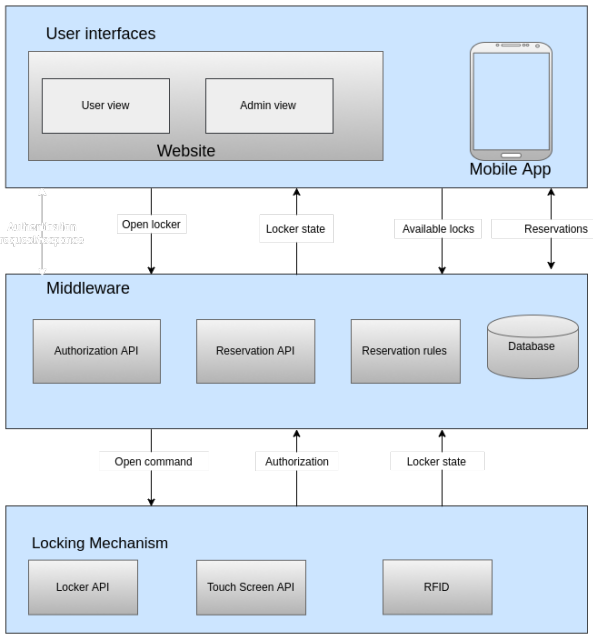

Figure 1: Components of the solution

Figure 1 shows the break-down of the solution into components. The arrows in the diagram show direction of communication or requests. The diagram is a high level representation of the final system. Each component especially the webserver has more components than shown; these are detailed in separate reports. This paper describes the design and implementation of the webserver.

1.3 Report Overview

The rest of this paper is structured as follows. Section 2 gives a background of the problem and the final solution. Section 3 gives a description of the embedded design life cycle followed when developing the system. Section 4 gives a detailed analysis of the requirements of the webserver. Section 5 gives the design of the components that constitute the webserver. Section 6 outlines the

tools used to develop the webserver. Section 7 presents the tests conducted and the results obtained. Section 8 discusses the results of the tests. Section 9 presents related work. Finally section 10 concludes with a discussion of the webserver and future work.

2 BACKGROUND

The computer science honours lab at the University of Cape Town has recently installed 30 lockers. These lockers do not have locking mechanisms and handles to open them. This is because traditional locking mechanism will change the aesthetic appeal of the lockers. There are two main issues with the current lockers:

(1) The absence of locking mechanisms defiles the main purpose of lockers

(2) The number of lockers installed is half the number of students doing honours. This means that students will have to share lockers.

The computer science honours lab does not have a full time administrator. This is an additional problem because lockers have to be fairly distributed amongst students. Existing locker administration systems allocate lockers for a period of a semester to a year. The problem with this arrangement is that not all students use their allocated locker daily. Allocating daily makes administration an even lengthier task.

The Lockit project proposes a solution which allows users to use a mobile application, website, RFID or touch screen interface to reserve and use the lockers. A Raspberry Pi is used to control the locking mechanism installed on the lockers. The Raspberry Pi runs application programming interfaces for controlling the locking mechanism. Adding other computational tasks would make the system respond poorly to requests. This paper investigates the feasibility of using a webserver to integrate the cross platform applications with the embedded system. The aim of the proposed solution is to use low cost hardware to manage the lockers in the honours lab without affecting the aesthetic appeal of the lockers. The webserver will handle most of the computational overheads and the Raspberry Pi will handle opening of the correct locker as requested by the webserver.

3 SOFTWARE DEVELOPMENT METHODOLOGY

The design of an embedded system requires hardware and software to be developed in parallel. It is for this reason that this project was developed using the embedded design life cycle [2]. Figure 2 shows a diagram of the embedded design life cycle. The embedded design life cycle has seven phases. The rest of this section gives details of how each of the seven phases was followed.

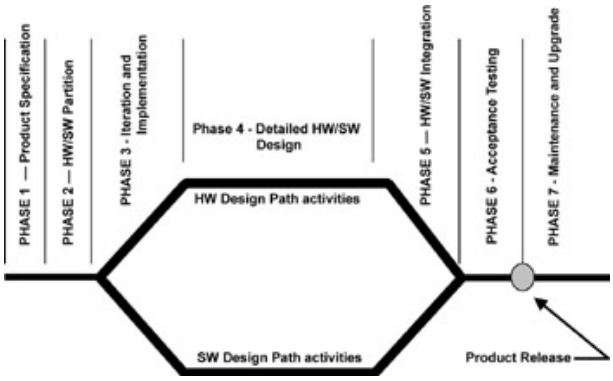

Figure 2: Embedded design life cycle diagram

3.1 Product Specifications

The first phase of the embedded life cycle starts with product specification. In this phase, the product owners of the lockers gave specifications for the webserver. The specifications were collected through weekly meetings. The details of the collected specifications are in section 4.

3.2 HW/SW partition

The second phase of the embedded life cycle is software and hardware partitioning. In this project the hardware partition consists of the Raspberry Pi and locking mechanisms. The user interfaces and webserver make up the software partition.

3.3 Iteration and implementation

The third phase of the embedded design life cycle represents the early design work before the hardware and software partitions diverged. Even though software components were defined, there was plenty of leeway to move these boundaries as more of the design constraints were understood and modeled. In this phase the design was fluid so, it was decided on possible inputs to the webserver and what the outputs should be for use cases.

3.4 Detailed HW/SW design

In this fourth phase, the hardware and software partitions were developed in parallel. The webserver was designed using use cases. A use case would be specified together with its inputs and outputs. A detailed design of the webserver is in section 5.

3.5 HW/SW integration

The hardware and software integration phase did not need special tools because of the way the system was modularized. Defining expected inputs and outputs in the iteration and implementation phase made this phase easier for the different components. In this phase the three components (see figure 1) are connected to communicate with each other.

3.6 Acceptance testing

The sixth phase is testing of the final product. Testing the webserver was mostly based on ensuring security; this is because the webserver is a single point of failure. If it fails, then the safety of the lockers would be compromised. More on the details of tests and results are in section 7.

3.7 Maintenance and upgrade

In this phase of the embedded life cycle, the final product will have to be maintained to ensure continuous operation. We have not reached this stage, however the webserver has been designed to keep logs for anyone who may need to upgrade the final product. The code for the webserver has been extensively modularized and commented to make it easy to upgrade. The webserver has also been designed to give precise details of any errors that may arise. The development tools used (as will be detailed in section 6) are freely available and have a large developer community to support future developers of the system.

4 REQUIREMENTS ANALYSIS

This section presents the requirements gathered in the first phase of the development cycle as understood by both the development team and the product owners. The requirements are as follows:

4.1 Cancel reservation

The users must be able to cancel a reservation if they need to.

4.2 Email notification

The webserver should notify users when their locker is opened. This should be added as a security feature to minimize theft.

4.3 Locker configuration

The administrator should be able to add,update or remove lockers from the system. This would be required during maintenance or when scaling.

4.4 Penalties

The webserver should penalize users that keep the lockers longer than they reserved them for.

4.5 Reservation modifications

The users must be able to modify their reservation times. This can only happen if the user has not exceeded the maximum allocated time to use a locker and if there is no booking for that particular locker.

4.6 Reserve locker

The users of the lockers must be able to reserve a locker for a period of time. The webserver has to ensure that users use a locker that they reserved. Lockers must only be available for the duration of a reservation.

4.7 Time allocation

The webserver should be able to dynamically calculate the maximum time a user can use a locker. This should be based on the total number of registered students, total number of lockers, previous requests and uses for lockers.

4.8 University authentication

The server must authenticate users using the same authentication method used by the university. This includes card authentication and user name and password authentication. The user name and passwords must not be stored on the server.

4.9 Unlock locker

The webserver has to communicate with the Raspberry Pi API to open a locker. Furthermore, the webserver must ensure that users only open a locker that they reserved. Thus preventing users from opening other user's lockers.

4.10 User information

The administrator should be able to add, update or remove students allowed to use the system. Only students doing honours in computer science should be able to use the system.

4.11 View logs

The administrator should be able to see, who opened which locker and when. This is for security reasons to track user activity.

4.12 View reservations

The administrator alone, must be able to see all reservations made. Any user that is not an administrator must not be able to do this.

5 DESIGN

The webserver is made up of APIs and a database. This section gives detailed designs of both the database and the APIs.

5.1 Application Programming Interfaces

The purpose of an API is to specify how software components interact. The design of the APIs that make up the webserver is shown in figure 3. In figure 3, the first line in Bold is the name of the API. Below the name of the API is the URL for accessing that

particular API. Below the URL is all the methods allowed by the API; the methods in red can only be accessed by the administrator. The methods in blue can be accessed by anyone that is authenticated. Below the methods is the response data returned by the API when a GET request is received. The data returned has a type specified in bold. For POST and PUT requests all APIs respond with an empty Python dictionary.

The arrows represent the relationships between the APIs. There are two kinds of relationships in the design diagram, these are “have one” and “have many”. The “have one” relationship is interpreted as a link to a single API object. The “have many” relationship is interpreted as a link to one or more API objects. For example a Locker can be linked to many LLogs. So there could be many LLogs links/URLs that have information about the same Locker. Another example, a Student is linked to only one Reservation. In this last case, there is only one Reservation link/URL for a Student.

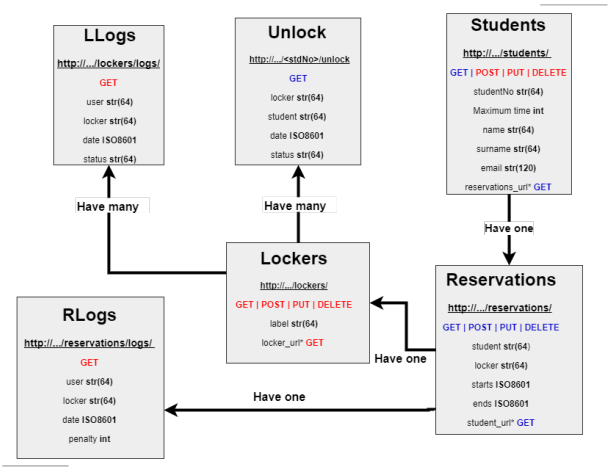

Figure 3: Design of APIs

5.1.1 Lockers. The Lockers API is responsible for synchronizing information of lockers. This API is only accessible to the administrator. It has four methods, GET, PUT, POST, and DELETE. It is used to configure lockers. Label is used to uniquely identify a locker. The administrator can set this to anything using a POST/PUT request.

5.1.2 LLogs. The LLogs API is responsible for keeping record of requests to open a locker. It is only accessible to the administrator via the GET method. The value for user is a UCT student number. Locker is the label of a locker, date is the date and time when the action was performed. Status is an indication of whether the action was successful or denied. This API responds with all records of unlocks.

5.1.3 RLogs. The RLogs API is responsible for keeping record of reservations. The records stored on this API are never deleted. This API is also used calculate the maximum time to allocate lockers and the penalties that users incurred. It is only accessible to the administrator via the GET method. Penalty indicates the hours a user is penalized for next reservation.

5.1.4 Reservations. The Reservations API is responsible for active reservations. This includes creating, deleting and modifying the reservation. This API has four methods which are accessible to any role, GET, POST, PUT, and DELETE. The value for starts is a date and time which indicate the start of a reservation. The value of ends is also date and time except that here it indicates the end time of a reservation.

5.1.5 Students. The Students API is responsible for queries related to student information. It allows for four methods, GET which is accessible by any role and POST, PUT and DELETE which can only be accessed by the administrator. The value for maximum is the calculated maximum hours that a user can book a locker for that day.

5.1.6 Unlock. The Unlock API is responsible for communicating with the locker raspberry pi that controls the lockers. This API only has a GET method which is accessible to any role. The response of this API is the log that was created in the LLogs API. However this is raw data and not a link.

5.2 Database

The database stores information needed by the server's APIs. The schema of the database is shown in figure 4.

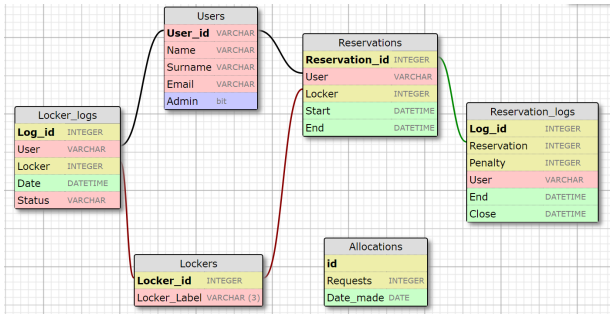

Figure 4: Database design

5.2.1 Allocations. The allocations table stores the total number of reservation requests made on any day. The data stored in this table is used to calculate the maximum time users get to use a locker, for the same day the following week.

5.2.2 Lockers. The lockers table stores locker information.The information stored is simply the locker label. This table helps with configuring lockers on the server so that they are correctly mapped to the physical lockers.

5.2.3 Locker logs. The locker logs table stores information about accesses to lockers. It stores the locker, user that tried to open the locker, the date, and status which indicates whether the locker opened or not. The purpose of this table is to help trace lost goods in case someone opens a locker maliciously.

5.2.4 Reservations. The reservations table stores information about a reservation. The information includes: user (from the users table), locker (from the locker table), start time and end time of the reservation. When a reservation ends, it gets removed from this table. The purpose of this table is to store active reservations.

5.2.5 Reservation logs. The reservation logs table stores information about all reservations made. the information stored includes: reservation (orphaned from reservations table), penalty acquired, user, end time of reservation, time the reservation was closed. The penalty could be made dynamic but the cost of constant calculation is higher than a once of write.

5.2.6 Users. This table stores user information. The information includes student number, name, surname, email and role (administrator or not).

6 IMPLEMENTATION

This section presents the tools used to implement the design in the previous section. The webserver is running an apache webserver software installed on an Ubuntu virtual machine.

6.1 Application Programming Interfaces

The APIs were implemented with Flask-RESTful [5]. Flask-RESTful is an extension of flask (a Python micro-framework) that adds support for building Representational State Transfer (REST) APIs. Flask is based on the Python programming language. Python has plenty of libraries and user community to support future developments on the webserver [9]. Python is also widely used for Internet of Things[4], which makes it even more relevant for our project.

6.2 Database

The database has been implemented using MySQL. MySQL is a popular open source database that has been optimized for web based applications [8]. The popularity of MySQL makes it have a large user community to assist with any issues that may be encountered during the lifetime of the webserver. MySQL has been designed to achieve high levels of scalability, availability and performance using low cost commodity hardware [8]. The database is managed with SQLAlchemy. SQLAlchemy is a Python SQL toolkit and Object Relational Mapper that abstracts away SQL queries [1].

6.3 Penalty calculation

The penalties that users incur are in hours. For every 45 minutes that a user is late they lose 1 hour of the next booking. The final count of penalties incurred is returned when an unlock request is received. When a late user opens their locker, the reservation is gets deleted. This only happens when a user is late.

6.4 Security

The security of the server has a two parts, username and password authentication and token authentication. The username and password are used to request an authentication token. The details of the two parts are as follows:

6.4.1 Card authentication. The webserver also uses a UCT access card number to authenticate users. When an access card number is provided by the Raspberry Pi, the server sends the card number to an access card API. If card number is authenticated, the webserver generates a token that is then used to make subsequent requests.

6.4.2 Token authentication. The university's LDAP (Lightweight Directory Access Protocol) is used to authenticated a user; a token is then generated using the TimedJSONWebSignatureSerializer function in Python. The token generated only lasts for an hour. After an hour, users have to request another token. An administrator token has different privileges from a user's token. The administrator token only lasts for 30 minutes.

6.4.3 Username and password authentication. The username and password are used to obtain a token. This uses flask 's HTTP Basic Auth module [5]. When a user enters a username and password, the webserver queries the provided detail on the university's Lightweight Directory Access Protocol (LDAP). The user's passwords are not stored on the server. LDAP simply searches for the user and returns information about the user[7]. The information returned is then used to determine whether a user is authorized or not. The information returned includes username, surname, faculty, academic career, courses, affiliation with the university and student status.

6.5 Time allocation

The time allocation is calculated to ensure that every user gets an opportunity to use a locker. The time allocated to students varies depending on use and demands for lockers. The time allocation is calculated using the equation below.

$$Allocation = \frac{(24 \times lockers \times students)}{requests} - used - penalties \quad (1)$$

where:
lockers = The number of lockers installed
students = The number of students registered to use the lockers
requests = The total requests received on the same day last week
used = The hours used for the day by the student
penalties = The number of penalties from previous reservation
The 24 is the hours in a day.

7 EVALUATION AND RESULTS

This section presents the tests conducted and the results obtained.

7.1 Acceptance tests

These test were performed to test whether the webserver has met the specifications in section 4. The manner of conducting these tests involved sending requests to the webserver and checking the response if it is as expected. Figure 5 shows an example of a response received from the webserver. A summary of the rest of the results for such tests is presented in table 1.

```
HTTP/1.1 202 ACCEPTED
Access-Control-Allow-Headers: x-requested-with, Content-Type, origin, authorization, accept, client-security-token
Access-Control-Allow-Methods:POST, GET, OPTIONS, DELETE, PUT
Access-Control-Allow-Origin: *
Access-Control-Expose-Headers: X-Auth-Token
Access-Control-Max-Age: 1000
Connection: Keep-Alive
Content-Length: 101
Content-Type: application/json
Date: Thu, 21 Sep 2017 17:24:28 GMT
Keep-Alive: timeout=5, max=100
Location: http://lockit.cs.uct.ac.za/api/v1/logs/?id=422
Server: Apache/2.4.18 (Ubuntu)

{
  "date": "2017-09-21 19:24",
  "locker": "1",
  "status": "success",
  "student": "plsnor001"
}
```

Figure 5: Unlock locker

Figure 5, shows the response of the webserver when a GET request is sent to unlock a locker. The response was generated by sending the following request:
GET https://lockit.cs.uct.ac.za/api/v1/plsnor001/unlock. This requires the student with student number plsnor001 to have an active reservation. If this is not the case, the response shown in figure 6 will be returned.

```
HTTP/1.1 403 FORBIDDEN
Access-Control-Allow-Headers: x-requested-with, Content-Type, origin, authorization, accept, client-security-token
Access-Control-Allow-Methods:POST, GET, OPTIONS, DELETE, PUT
Access-Control-Allow-Origin: *
Access-Control-Expose-Headers: X-Auth-Token
Access-Control-Max-Age: 1000
Connection: Keep-Alive
Content-Length: 101
Content-Type: application/json
Date: Thu, 28 Sep 2017 19:33:01 GMT
Keep-Alive: timeout=5, max=100
Server: Apache/2.4.18 (Ubuntu)

{
  "error": "Locker Error",
  "message": "No active reservation for: plsnor001",
  "status": 403
}
```

Figure 6: Failed Unlock request

Table 1: Functional Requirements table

Functionality	Pass/Fail
Reserve locker	Pass
Unlock locker	Pass
View reservations	Pass
University authentication	Pass
View logs	Pass
Cancel reservation	Pass
User information	Pass
Locker configuration	Pass
Reservation modifications	Pass
Time allocation	Pass
Email notification	Pass
Penalties	Pass
Card Authentication	Pass

Table 1 shows impressive results, the webserver has met the specifications. These results were not enough to determine the feasibility of the webserver. The rest of this section presents two tests that validate the feasibility of the webserver.

7.2 Load tests

The load tests, were conducted to test the amount of time it takes the webserver to respond to a given number of requests. These tests were tested for 100 requests sent by 25 different concurrent users. Each user doing 100/25 = 4 sequential requests. The last two tests were tested for 5 requests by 2 different concurrent administrators. One administrator makes 3 sequential requests and the other makes two. The tests and results are as follows:

7.2.1 Available lockers. This test checks the response time of the server when a user requests a list of free lockers. Figure 7 shows how the response time of the server changes as the number of requests increases.

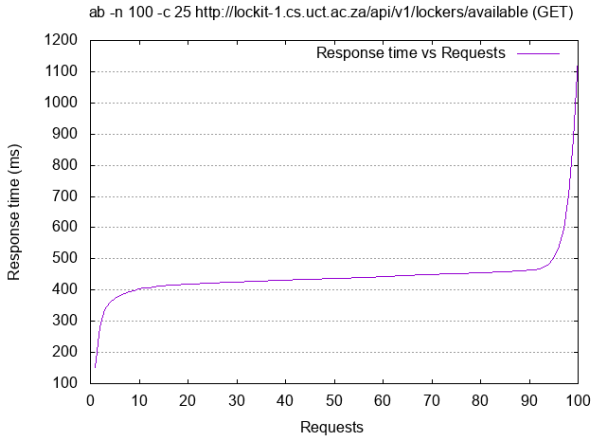

Figure 7: Available lockers

The server takes about 150ms to respond to a single request for free lockers. It takes between 400ms and 500ms for requests between 10 and 95. Overall the server takes about 1.1 seconds to serve 100 requests for free lockers.

7.2.2 Reserve locker. This test checks the response time of the server when a user makes a reservation. Figure 8 shows how the response time of the server changes as the number of requests increases.

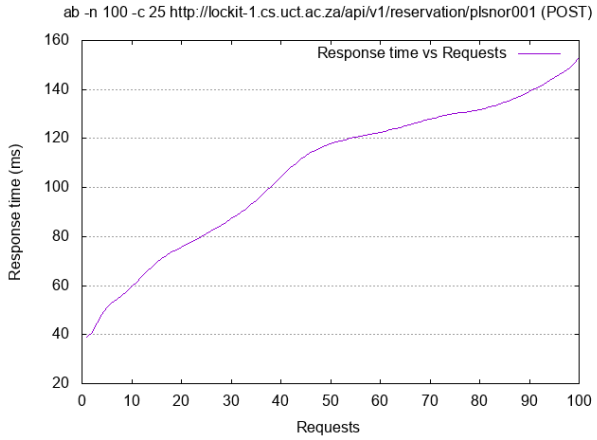

Figure 8: Reserve locker

The response time for making a reservation increases linearly with increasing number of requests. It takes the server 40ms to respond to a single request to make a reservation. It takes 150ms to serve 100 requests.

7.2.3 Token authentication. This test checks the response time of the server when a user requests a token. Figure 9 shows how the response time of the server changes as the number of requests for tokens increases.

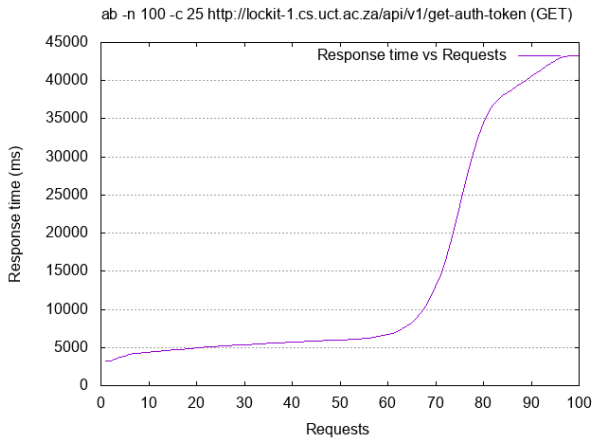

Figure 9: Token authentication

The server takes 5 seconds to respond to 10 to 50 token requests. For requests between 60 and 80, the response time increases exponentially. Overall, the server takes about 43 seconds to respond to 100 requests for authentication tokens. It takes the server about 3 seconds to respond with a token to a single request.

7.2.4 Unlock locker. This test checks the response time of the server when opening a locker. Figure 10 shows how the response time of the server changes as the number of requests increases.

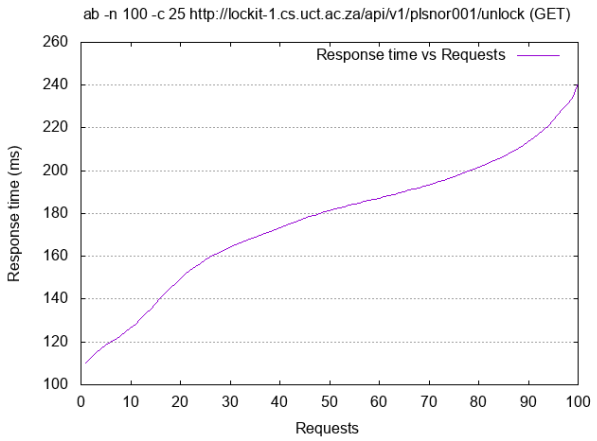

Figure 10: Unlock locker

The response time for opening a locker increases linearly with increasing number of requests. It takes the server 100ms to respond to a single request to open a locker. It takes 240ms to serve 100 requests.

7.2.5 View all students. This test checks the response time of the server when the administrators view all students. Figure 11 shows how the response time of the server changes as the number of requests increases.

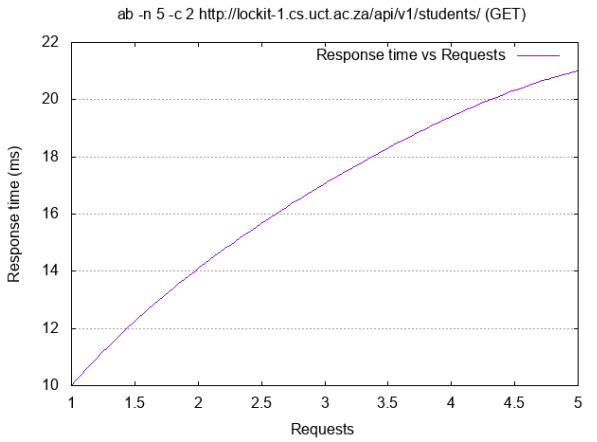

Figure 11: View all students

The response time for viewing all students increases linearly with increasing number of requests. It takes the server 10ms to respond to a single request to view students. It takes 21ms to serve 5 requests.

7.2.6 View logs. This test checks the response time of the server when the administrators view all logs. Figure 12 shows how the response time of the server changes as the number of requests increases.

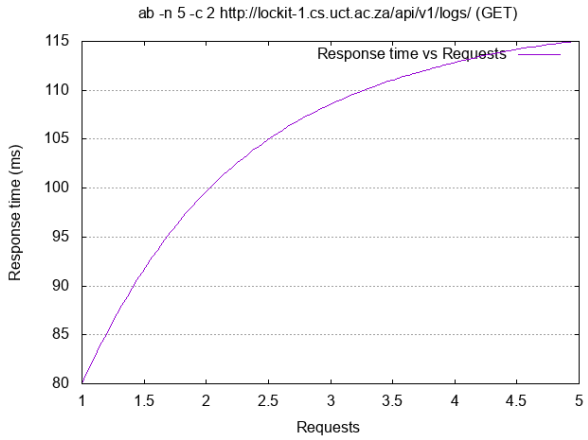

Figure 12: View logs

The response time for opening a locker increases exponentially with increasing number of requests. It takes the server 80ms to respond to a single request to view logs. It takes 115ms to serve 100 requests.

7.3 Security tests

The security tests were conducted to demonstrate the webserver's responses to potential security threats.

7.3.1 Administrator only access. This test is conducted by using a user that is not an administrator to request a URL that is restricted to the administrator.

```
HTTP/1.1 401 UNAUTHORIZED
Access-Control-Allow-Headers: x-requested-with, Content-Type, origin, authorization, accept, client-security-token
Access-Control-Allow-Methods: POST, GET, OPTIONS, DELETE, PUT
Access-Control-Allow-Origin: *
Access-Control-Expose-Headers: X-Auth-Token
Access-Control-Max-Age: 1000
Connection: Keep-Alive
Content-Length: 88
Content-Type: application/json
Date: Thu, 21 Sep 2017 17:51:16 GMT
Keep-Alive: timeout=5, max=100
Server: Apache/2.4.18 (Ubuntu)
WWW-Authenticate: Basic realm="Authentication Required"
{
  "error": "unauthorized",
  "message": "Please send admin token",
  "status": 401
}
```

Figure 13: Non-administrator token failed response

Figure 13 shows the response of the server when a user that is not an administrator makes a request to a URL that is restricted to the administrator. The web server response with an unauthorized status as expected.

7.3.2 No token provided. In this test a user sends a request to the webserver without first requesting a token. The response is shown in figure 14.

```
HTTP/1.1 401 UNAUTHORIZED
Access-Control-Allow-Headers:x-requested-with, Content-Type, origin, authorization, accept, client-security-token
Access-Control-Allow-Methods: POST, GET, OPTIONS, DELETE, PUT
Access-Control-Allow-Origin: *
Access-Control-Expose-Headers: X-Auth-Token
Access-Control-Max-Age: 1000
Connection: Keep-Alive
Content-Length: 102
Content-Type: application/json
Date: Thu, 21 Sep 2017 17:53:19 GMT
Keep-Alive: timeout=5, max=100
Server: Apache/2.4.18 (Ubuntu)
WWW-Authenticate: Basic realm="Authentication Required"

{
  "error": "unauthorized",
  "message": "Please send your authentication token",
  "status": 401
}
```

Figure 14: Unauthorized unlock fail

The above response was generated by sending a GET request to https://lockit.cs.uct.ac.za/api/v1/plsnor001/unlock. The webserver responds with an unauthorized status but with a unique message as expected.

7.3.3 Token authentication. This test tested the webserver to confirm a token response when requested using valid UCT login credentials. The response is shown in figure 15.

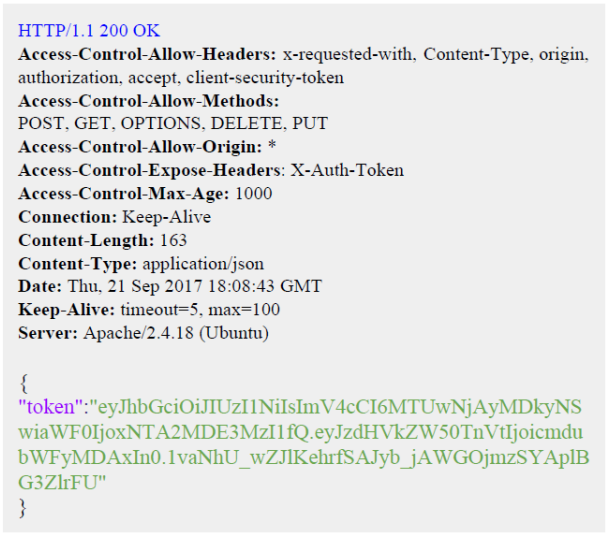

Figure 15: Authentication token

The above response was generated by sending a GET request to https://lockit.cs.uct.ac.za/api/v1/get-auth-token. The response body has a long string of random characters. If this string gets altered in anyway, the will send the response shown in figure 16.

7.3.4 Username and password authentication. Outputs for sending incorrect UCT student number and password to get a token.

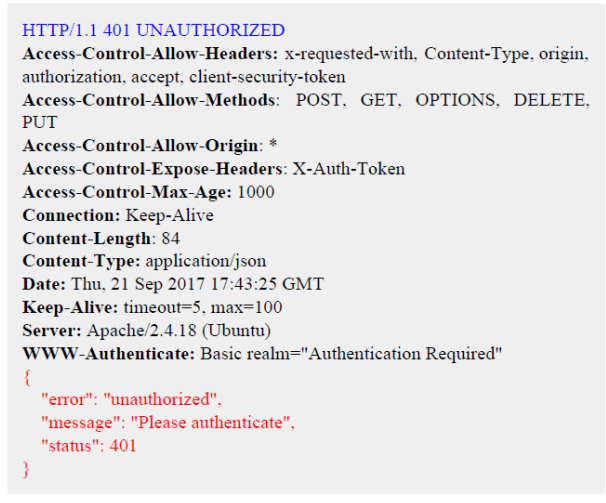

Figure 16: Unlock locker

The response in figure 16 is generated when a token is requested with incorrect UCT credentials or with a card number that is not for a current Computer Science honours student.

8 DISCUSSION

The design of the APIs maximizes cohesion and minimize coupling. This is important for the webserver because it minimizes vulnerabilities. In [6] these two structural metrics are helpful for predicting vulnerabilities. The use of LDAP is safe for protecting passwords for users in case the webserver gets hacked. However this feature makes the server slower as can be seen in section 8.2. The time to authenticate users was the longest with 43 seconds for 25 concurrent requests. This time could be much less if the user passwords were also stored on the server. However this is not required by the product owners. The server also takes time to respond when a user requests to open a locker. This happens only when the embedded system is offline, however when it is online the server responds in 100ms as can be seen in section 8.2.4.

9 RELATED WORK

There has been work done in this area of locker controllers [6] presents a programmable electronic lock that can have multiple users. The problem with this locking mechanism is that it is keypad-activated and it is programmable by the user. This will violates the aesthetic appeal of the lockers and still require a full time administrator to monitor use. Users can retain lockers for a long time without removing their pass codes.

[3] Presents a very similar locker setup. The difference is that [3] has an input device installed for every locker. Furthermore, [3] has a control unit and it needs an administrator to manage the system. This setup affects the aesthetic appeal of the lab and this solution is expensive. This would work if every student owned a locker for a semester.

In [10] an electronic locker is activated using a telecommunications network. Users open lockers by making a call to a number assigned to a locker. Subscribers are stored in a database that is used to authenticate the users when they make calls to lockers. This system does not use the same authentication as the university. Furthermore, the locker numbers will need to be changed every time a booking ends otherwise anyone can make a call to the locker.

10 CONCLUSIONS AND FUTURE WORK

The basic requirements of the specifications have been met by the server. The server can reserve a locker, open a locker, show logs and prevent unauthorized accesses. The results of the webserver's tests show that it is feasible to use a webserver to integrate cross-platform applications with an embedded system. In future when the server expands to accommodate more than 100 users, the performance can be increased by using apache's mod cache module to cache requests. This will improve the performance by 5 to 10x over all methods combined. Overall the performance of the webserver can be improved by utilizing caching mechanisms provided by apache.

ACKNOWLEDGMENTS

Our thanks to Gary Stewart, Craig Balfour and Samuel Chetty who have been excellent guides throughout this project. To Marion and Raees, my project partners for showing interest and dedication to this interesting learning experience.

REFERENCES

[1] Mike Bayer. 2010. SQLAlchemy Documentation. (2010).
[2] Arnold S Berger. 2002. *Embedded systems design: an introduction to processes, tools, and techniques.* Focal Press.
[3] Kevin E Booth, Harry N Popolow, Richard R Ford, Edward E Johnson, Jon S Loftin, Lance C Osborne, and David W Johnson. 2005. Electronically-controlled locker system. (April 12 2005). US Patent 6,879,243.
[4] Wesley J Chun. 2012. *Core Python Applications Programming.* Prentice Hall Press.
[5] Miguel Grinberg. 2014. *Flask web development: developing web applications with python.* " O'Reilly Media, Inc.".
[6] Yucel K Keskin and Asil T Gokcebay. 1999. Programmable digital electronic lock. (April 13 1999). US Patent 5,894,277.
[7] Jim Sermersheim. 2006. Lightweight directory access protocol (LDAP): The protocol. (2006).
[8] Brent Ware et al. 2002. *Open source development with LAMP: using Linux, Apache, MySQL and PHP.* Addison-Wesley Longman Publishing Co., Inc.
[9] Aaron Watters, James C Ahlstrom, and Guido Van Rossum. 1996. *Internet programming with Python.* Henry Holt and Co., Inc.
[10] Isaac DM White and James Dickens. 2005. Activation of electronic lock using telecommunications network. (April 26 2005). US Patent 6,885,738.