5-25-2018

LUCCA: Lab Usage Controller for Centralized Administration

Daniel Eynis
Portland State University

LUCCA: Lab Usage Controller for Centralized Administration

by

Daniel Eynis

An undergraduate honors thesis submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

University Honors

and

Computer Science

Thesis Adviser

Wu-chang Feng

Portland State University

2018

Abstract

This paper explores the Lab Usage Controller for Centralized Administration (LUCCA) system. This exploration is meant to act as a retrospective to show the development of the LUCCA system. The system was developed to be used by the Electronics Prototyping Lab to help manage users and keep track of machines. The system is meant to streamline management. The aim is to dive into the specifications, design, and implementation of the system. These parts of the system will open the understanding for the choices that were made during the development process. The system was built as a web app on top of the Node.js and Express.js framework. The system includes a user interface, backend database, and API for machines. The system was successfully delivered to the EPL and was approved by the project sponsors meeting most or all of the stated requirements of the system. All code and documentation of the system is available GitHub and is accessible from the following link: https://github.com/LUCCA-Capstone/LUCCA

Table of Contents

1. Introduction

The goals of this projects was to create an access control system for Portland State University (PSU) Electronics Prototyping Lab (EPL). This project involved seven students working together to deliver a product as part of the Computer Science Capstone project. The access control system that was delivered to the EPL sponsors was built as a web app to help lab administrators streamline managements of users and machines within the lab. The web app delivered a front facing user interface, a robust backend, and an API system to help with machine communication. The beneficiaries of the work done by our team would be the administrators and managers of the EPL. The system is there to alleviate complexities in managing all the users and machines in the lab. In addition to this the users of the lab would also benefit as it would make it easier for them to get set up on the lab system and start using machines within the lab. The scope of the project had to be narrow enough to be completed within six months. There were two main focuses for the project: the frontend and the backend. The frontend would be the user interface where admins could log in and interact with and would allow them to manage users and their permissions to machines within the lab. The backend involved implementing all the logic and infrastructure to store user data and provide an API to get information from machines within the lab. The first 3 months of the project were focused heavily on documentation and requirements collection. The second 3 months were focused on implementation, testing, and final delivery of the product. Our team split into two subteams, one focused on the frontend and the other on backend.

The assumptions that our work was based on were the documents specifying the requirements of the system. These requirements were collected through numerous meetings with the project sponsors regarding what they wanted to see be features in the final product. Crafting the requirements the following things were taken into consideration: time constraints, and team knowledge and capability. In the end the access control system was successfully delivered the the EPL sponsors with most if not all requirements met.

2. Background

This project focuses on developing an access control system for the EPL. The EPL is a lab for rapidly prototyping electronics projects. It is meant for both students and the community to be able to walk in with an idea and walk out with a fully functioning prototype (http://psu-epl.github.io/). The EPL reached out to the Computer Science Capstone program to build them an access control system for managing users and machines within the lab. The lab consists of administrators, managers, users, and machines/stations. Users need to sign up for the lab and sign a waiver. Users must be trained on a station by an administrator or manager before they can independently use it. All of this information must be tracked and secured. This is the problem that has been identified by the sponsors from the EPL. The sponsors want to solve this problem by proposing creating a system that would be built specifically to handle the problem they are facing by having to keep track of everything that is going on in the lab. The biggest stakeholders of this project are the sponsors from the EPL and the developments team

for this project. The minor stakeholders are the users that use the EPL and their resources. If the project failed then the sponsors would not get a completed project at all or very little of what they required to be in the final product. Time would have been wasted. We as the group that is developing the product are major stakeholders because we are investing our time to develop a project for the sponsors. Any change in requirements would potentially mean more development time or commitment. The goal of this project is also to release the software as open source available for anyone. Delivering a working product that users can access is crucial as the software represents our development skills and progress. The users of the lab are minor stakeholders as the way the product is built will affect how they will interact with the lab when the product is integrated. Until then the lab will use its old method of manually keeping track of users, machines, and permissions.

The constraints for the solution is that there are six months to deliver the products. There is also the knowledge programming constraint as not all team members have experience will building software in an industry setting which we were trying to achieve. We would also needed to pick a technology stack that everyone can adapt and learn fairly quickly due to the time constraints. The existing solution that the lab currently uses is to sign up users through a paper sheet. Once users are signed up all tracking of users, machines, and permissions is done through a spreadsheet. This approach early on could have been maintainable, but with the growing number of users and the expansion of the lab they must adapt a better solution. This is the problem that our team was trying to solve. Currently there does not exist a system that would fully meet the

requirements of the lab. Researching online one system does come up called “membership-system” by South London Makerspace. The system is a “membership management system, it's chiefly a database of member data for legal purposes, setting up subscription payments, managing access control permissions, logging events, and interfacing with Discourse permissions” (https://github.com/southlondonmakerspace/membership-system). This system is probably the one that gets close to what the sponsors would like but does not check all the boxes. The system also includes unnecessary features such as subscriptions payments, and an interface to Discourse. The EPL wants a way for users to sign up for the lab and sign the waiver electronically which this system does not provide. In addition there does not seem to be any API to interface with machines within the lab which is another crucial part to the proposed EPL system. The EPL wants a system that is tailor made to their lab and other similar labs within the area.

The proposed project tried to solve the problem by creating an access control web app. Which we have named the Lab Usage Controller for Centralized Administration (LUCCA). LUCCA is built on top the Node.js and Express.js framework. This will allow the EPL to set up a server through which they can access the interface online. LUCCA also requires a robust backend database for which we have decided to use PostgreSQL. This backend database will store things such as user information, credentials, and permissions. The frontend will use Nunjucks.js as its rendering engine combined with Bootstrap. The API will be served through a secondary Node app which will handle requests from machines within the lab. Communication between clients and

server are secured through SSL. The codebase is hosted on GitHub and the development process is to use Git for version control of the project. This is the major high level overview of the project. To achieve the stated aim of the project, we have developed a plan that involved the collection of requirements, crafting the design of the project, implementation planning, testing, and then final successful delivery of the product to the EPL.

3. Specification

The specification section will describe what the software system was required to do. The specification section will cover various functional and non-functional requirements of the LUCCA system. There are specifications for the frontend, backend, and API system. The primary focus of the system is to keep track of who is using the lab and which machines within the lab are being used. The system provides a registration interface which will be used to gain access to the EPL. The system will record all machines a user has access to and the times in which they have used it [3]. A manager or administrator has the necessary functionality to train a user on a machine and then update the trained users profile on the backend database.The administrator will have the functionality of adding or removing machinery from the EPL, removing users from the EPL database, revoking a user's access to the EPL, and generating and reviewing reports. The database will store persistent data on a backend database. The database will store information on all users, managers, admins, machinery/stations and daily

activity, including: clock in and out times of all user types and the users and machine usages. The system has an API interface to communicate with the machines.

An administrator has the highest clearance level. An administrator needs to badge into the EPL once they arrive if they wish to use the system. An interface is provided to achieve this. There is also an interface for badging out as an administrator. An administrator can start using any machine within the lab by swiping their badge. The machine will then use the API to contact the server to grant them access. To stop using a machine the administrator can swipe their badge again. The administrator can train anyone on any machine in the lab. This is achieved by having a user swipe their badge first and getting a no access error, once the error pops up an administrator can swipe their badge within a variable number of seconds and the system will grant that user access to that specific machine by using the API. A users permissions to various machines in the lab can also be modified through the user interface. An administrator will also be provided a set of login credentials to access additional features through the user interface. This will be an email address and a password. Administrators will be able to grant other users administrator privileges as well as managerial status. The admin interface has a framework for reporting about user and machine activity and information. An administrator is able to search for specific users. Remove users from the system permanently, revoke or grant users access to various machines within the lab. As well as add or remove new machinery to the lab through the LUCCA system. Administrators will also be able to reset or change their passwords from the user interface. This is the high level specification for an administrator in the system.

A manager has the second highest clearance level. Managers will not have access to the features that are available to an administrator through the user interface. The same as the administrator a manager will be required to badge into the lab though the LUCCA system as well as badge out of the system once they are done using the lab. Managers are limited to using machines within the lab that they have been trained on and have gained access to through another manager or administrator. Managers can grant machine privileges to other users only if they themselves have privileges to that particular machine. A manager is similar to an administrator but they do not have access to the administrator user interface.

A user has the lowest clearance level out of all the three groups. A user will primarily interact with the machines within the lab. A user will be required to badge into the lab before they can start using any of the machines that they may have privileges to within the system. A user will also need to make sure to badge out of the lab once they are done. The administrator interface has functionality to badge out a user if they forgot to badge out. If the user badges into the lab for the first time they will be prompted to fill out a registration form through the user interface. Once a user submits the form they will be recorded in the LUCCA database. The user will use a web interface to achieve this and they will input things such as their badge id, signature, contact information, emergency contact information, and email. The web interface will receive a confirmation from the backend database if no input fields were left blank and that input fields are valid. Appropriate messages will be displayed to indicate any errors. The web interface

will display a message indicating that the user has been accepted into the LUCCA system

Each machine in the lab will will be identified by its name, description, unique id, and station id in the backend database. A badge swiping device will be used to obtain any users badge # when the user wants to use or stop using the machine. A machine will make a request through the API if a user is trying to use a machine. The machine will then receive a response from the backend server indicating if that user has privileges for that particular machine. A machine will send another API request once a user is done using a machine. A machine will also make a request to grant a user privileges if an administrator swipes their badge within a variable number of seconds after a user has been denied access. The lab clock in station will serve two purposes. Primarily is will allow any user type to clock in and out of the EPL. Additionally, it will allow new users to register for access into the EPL [3]. The clock in interface will send a user's badge # to the database after a user swipes their card and will receive a notification whether user has clocked in or clocked out, or is new. The interface will redirect appropriately and will display messages as to the outcome of the badge swipe.

The backend database will store persistent data on all users, machinery, stations, and usage activity in the EPL. It will communicate with the clock in station, all machine badge stations, the registration interface, and the admin interface. Information provided to the LUCCA system is secure. No trained user will be denied access to a machine they have been trained on. No untrained user will be granted access to a machine they haven't been trained on. The backend database and frontend interfaces

are competent enough to prevent the simplest forms of the top OWASP attacks. Communication from the backend database is fast and reliable to ensure users aren't waiting an inordinate amount of time for a response. Clear, unambiguous, documentation for setting up and initializing the software is easily available [3]. There is also clear, unambiguous, documentation for recovery from crashes and/or malfunctions are easily available and understandable. This is the outline of the various requirements of the LUCCA system that are available when correctly set up.

4. Design

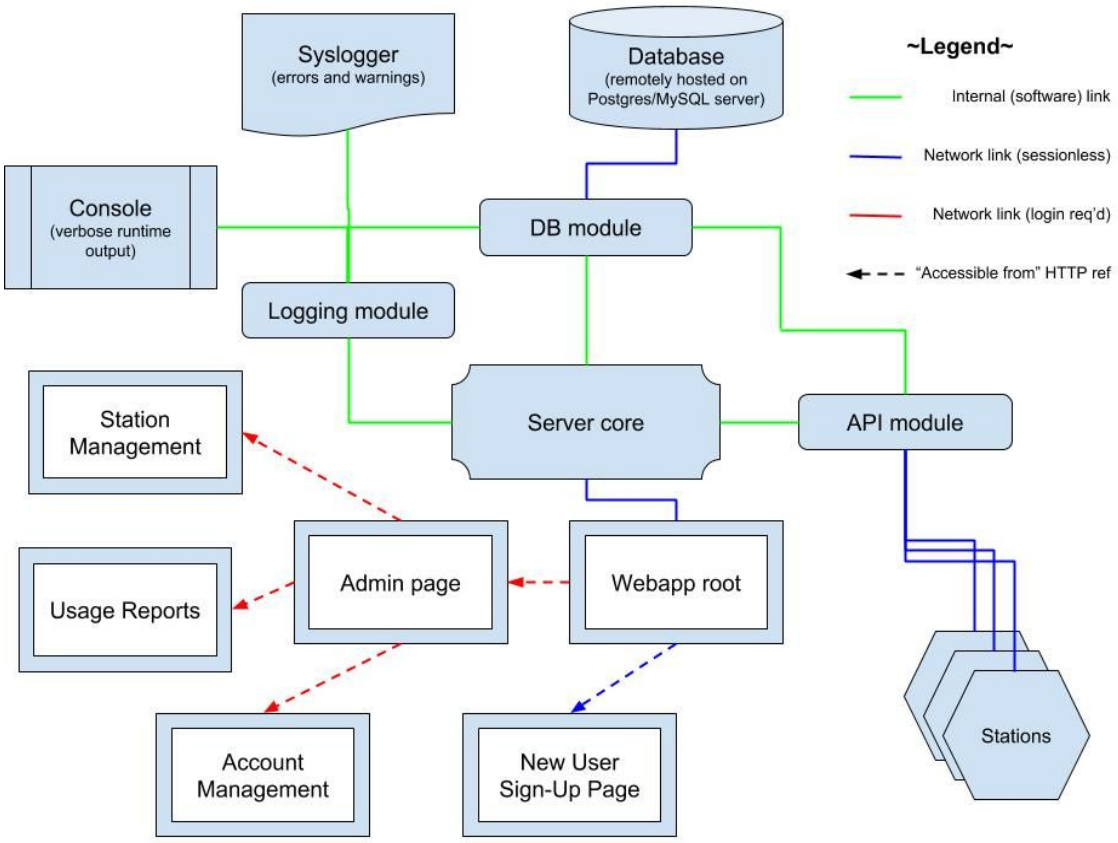


Figure 1: System Architecture

The design of the system involves being able to support a frontend user interface system, an API system for stations, and a backend database to persist data. The design of the system can be seen in Figure 1. The most important part of the system is its beginning from the server core. The server core will be responsible for initializing and connecting to the frontend, API, and database interface. The webapp root and corresponding boxes represent the frontend user interface system. Authentication and rendering of pages to the user will be handled by routes defined in the server core.

Routes to the places such as the new user sign up will not need to be authenticated or have a session required. Other routes that require administrative permission will require authentication and a session associated with logins into the system. The red dotted arrows in Figure 1 represent that a login by an administrator is required to gain access to those pages. A login will require an email and a password. Once an administrator is authenticated the server core is responsible for starting a session which will authenticate the administrator for any subsequent requests to the server. This means that the administrator does not have to keep logging into the system every time they want to access a protected page. Session information will be stored in main memory of the server. User information such as encrypted passwords will be stored in the database. When information will need to be retrieved the server core will communicated with the database module through an interface. This interface will provide various functions to retrieve data from the database. The use of the database will be abstracted out to an interface which other modules such as the server core and API can use to retrieve data.

The database module will provide predefined and implemented function that given an input will provide some predefined type of output. The functions that have been determined to be necessary and sufficient for the system are defined in Table 1. This table provides the name of the functions, their input arguments, and the type of returned values. In addition to this there is an explanation of the purpose of each function in the database interface. The reason there is a database module interface is so that there is an abstraction from the server core and API module. The only way that

Table 1: Database interface functions [4]

Function Name	Input Arguments	Returned Values	Explanation of Purpose
getUsers	<Date:createdBefore=undefined>,<Date:createdAfter=undefined>	<List:Object:userData>	For use by GUI to get a list of registered users for admins. Also used specifically for page where admins can review recent registrations.
createUser	<Object:userData>	<Object:statusCode>	For use in the registration page route handler, when receiving POST data from a user registration form submission.
modifyUser	<String:badgeId>,<Object:userData>	<Object:statusCode>	For use in admin GUI pages whenever making permanent alterations to the info associated with a user account.
validateUser	<String:badgeID || String:email>	<Object:userData || undefined>	For use by the user-access API route handler, to evaluate a user badge swipe to access a lab station.
deleteUser	<String:badgeID>	<Object:statusCode>	For use by admin management GUI to delete a user account from the system
getLoggedIn	<String:badgeID || String:email>	<Object:userData || undefined>	For use by the user-access API route handler, to evaluate a user badge swipe to access a lab station.
getPrivileges	<String:badgeID>	<List:Object:privilege>	For use by API to get privileges for user to access a certain station
grantPrivilege	<String:badgeID>,<String:stationID>	<Object:statusCode>	For use by API or GUI to grant a certain users privileges to a station.
removePrivilege	<String:badgeID>,<String:stationID>	<Object:statusCode>	For use by API or GUI to revoke a certain users privileges to a station.
getStations	<Boolean:registered || undefined>	<List:Object:stationData> || undefined	For use by GUI to generate a list of registered or non-registered stations identified by the system, to be shown to lab managers and admins.
createStation	<Object:stationData>	<Object:statusCode>	For user by API to register a new station in the system
modifyStation	<String:stationID>,<Object:stationData>	<Object:statusCode>	For use by GUI to alter information about a station in the system
deleteStation	<String:stationID>	<Object:statusCode>	For use by GUI to delete a station from the system.
logEvent	<String:eventClass>,<String:event>,<Date:eventDate=undefined>	<Object:statusCode>	For use throughout program code to write permanent log entries
getEvents	<String:eventClass>,<Date:before=undefined>,<Date:after=undefined>	<List:Object:eventData>	Returns a list of log objects, as defined by the database schema. Any of the parameters may be omitted. Omitting all parameters will return all events.
deleteEvent	<Integer:eventID>	<Object:statusCode>	Deletes one event with the corresponding ID. Event objects returned by getEvents() contain an ID field, which should be used here.

they can interact with the database is through the interface which splits up responsibility and makes any changes to the code and refactoring much easier to handle within the system. The only thing that modules using the database interface need to know is the input to the functions and what type of output they can expect.

The API module handles communication with the stations. The stations make a request to the API and the API will handle various requests and make the necessary modification to information within the database. The API supports HTTPS/1.1 with JSON payloads over TLS 1.1 or greater. TLS ensured that requests are authenticated and encrypted securing information communicated between stations and the API. The requests to API includes a non-standard message header, ‘Station-ID’, whose value shall be the ID string of the transmitting station [2]. The API may respond with any of the four following codes: 200 OK, 400 Bad Request, 401 Unauthorized, and 403 Forbidden. User access requests can submit a POST request to the following route: /api/user-access. The header to the request should include the following field: Station-State: <station state>. The value <station state> must be either “Enabled” or “Disabled” which indicates the state that the controller wishes the station to be in after the request was processed. correctly-formed user access request is received by the API and the user is permitted to use the station, the API will respond with ‘200 OK’ in the reply body, its own ‘Station-State’ header. In the response, the ‘Station-State’ header indicates the state that the station should be in after the badge request is processed [2]. If <station state> is “Enabled”, the following header fields will also be included in the response: User-ID-String: <user ID> where <user ID> is associated with the user

identity token which the controller believes to be responsible for enabling the station. If a valid user access request is received by the API, but the identified user is not permitted to use the station, the API will respond with ‘403 Forbidden’, and no “User-ID-String:” header (Station API specification document). Hardware reset notifications are of the following form: POST /api/local-reset. The hardware reset notification is an informational action provided as a method for a station to indicate that a hardware reset button or similar unauthenticated local mechanism was used to disable the station and end the most recent active user’s station enablement. State recovery requests are of the following form: GET /api/last-state. This API feature is provided as a method for stations to ask the system controller what state they are expected to be in. The intended use case is scenarios where the station has been power-cycled, or has been unable to establish a heartbeat for extended periods of time. Heartbeat echoes are of the following form: POST /api/station-heartbeat. This feature is provided to allow the API to respond to a heartbeat packet from the station. Whenever the controller receives a heartbeat from an unrecognized station, the Station-ID, network identity, certificate information, and the current timestamp are all recorded in a database. At a later time, any system administrator can review the list of stored, unrecognized station IDs as a list of “unassociated stations”. Through the administrator interface, the system administrator can permanently register the station with the system controller. The way user access requests are handled by the API is defined the Figure 2 which is a flowchart of questions asked in order to identify if the system needs to authenticate a user or

potentially grant a user access to a machine as a result of an administrator badge swipe in the system.

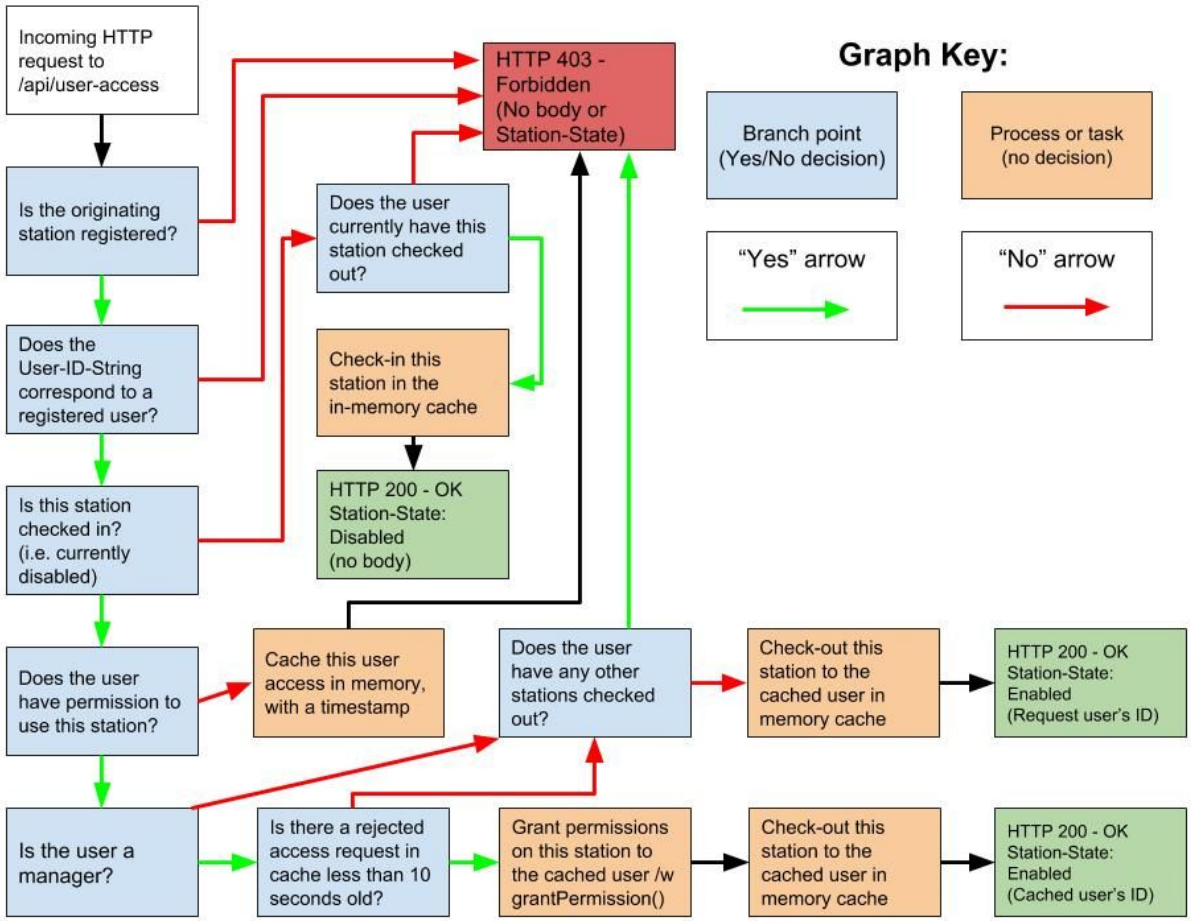


Figure 2: API user access resolution

The LUCCA system also includes a logger which is used to log events that occur in the system. All system logs are stored in the database and logs can be written to the database using the provided interface functions. The system will log various events such as login and logout attempts, privilege modifications and changes to user information. Loggin happens from the frontend side and the API side, as well as from the database interface side. Logs are split into five different categories to help group

logs of the same type when displaying them to administrators who are interested in specific types of logs. Logs can be any of five event types defined in the following table. Associated with each log is an entity field which is used to identify who initiated a various event that led to that type of event to be logged in the system.

Table 2: Log event types [4]

Event type	Context for 'entity' field
`user traffic`	user badge ID
`privilege`	modified user
`administration`	admin responsible
`station`	station ID
`internal error`	error location (function name or whatever is relevant)

5. Implementation

This section will cover the various implementations aspects of the project. It will do so by examining some of the code in the system. This section will not try to cover all the code or implementation techniques but will focus on a few interesting ones. The project was built on top of Node.js and uses the framework to create and run a working web app of the LUCCA system. At the top level of the repository there are various folder and files. The program app begins its life in the `bin/www` file. In this file the server core which will serve frontend web pages and the API service is set up to run on HTTPS.

```
var server = https.createServer({
    'pfx': fs.readFileSync(config.get('Server', 'pkcs_cert_file')),
  }, app);
```
[1]

The server is created with certificate file which is used to create secure connections between the client and the server.

```
var api_server = https.createServer({
    'pfx': fs.readFileSync(config.get('API', 'pkcs_cert_file')),
    requestCert: config.get('Developer Options', 'require_client_certificate') == 'false' ? false : true,
    rejectUnauthorized: config.get('Developer Options', 'validate_client_certificate') == 'false' ? false : true
}, api_app);
```
[1]

Similarly the API server is also created with a certificate file but a configuration also defines whether or not the client is required to make requests with valid certificates and if the certificates are not valid whether to reject requests from that client. The various configuration options for the app are available in the lucca.conf file at the top level of the repository. At the top level there is also the package.json file where app dependencies are defined and are required to run the app. These dependencies can usually be installed through Node Package Manager (NPM). There are also folders in which code has been categorized. The database folder holds all the code that has to do with the database and implements the interface. The lib folder contains code to track machines and users who are using the API. The local_modules folder includes modules created natively to be reused and now only includes a module to parse the configuration file. The passport folder has the code that defines the various strategies for registering and logging into the system as well as resetting passwords. The public folder includes resources and code that can be accessed by web pages that are served to the client. The routes folder includes the server routes and api routes by which clients can reach different parts of the app. The test folder includes code to test various modules of the

app. Finally the views folder includes nunjucks files which are server side rendered and then served to the user.

In app.js file the app is initialized. The app utilized the Express.js framework to easily define routes and handle requests in the system. In this file a lot of the dependencies are retrieved and utilized.

```
var app = express();
```
[1]

The nunjucks view engine is set up in order to be able to utilize it to render the nunjucks files in the views folder. Nunjucks helps make it easier for the system to dynamically render pages as information changes in the database and then displayed to the client. The server first retrieves data from the database if required and the tells nunjucks to render a specific page and passes it the data to render on that page before it is sent to the client.

```
nunjucks.configure('views', {
  autoescape: true,
  express: app
});

app.set('view engine', 'nunjucks');
```
[1]

In addition to this the routes are registered with the server for both the frontend and the API. As well as the passport strategies that tell Passport.js how to handles various requests to be authenticated with the system.

```
app.use('/', require('./routes/index')(passport));
app.use('/api', api);
```
[1]

```
passport.use('login', login_strategy);
passport.use('register', register_strategy);
passport.use('reset', reset_strategy);
```
[1]

Similar setups can be seen in api_app.js file for the API side of the server but with much more minimal dependencies.

In the routes folder resides the code to handle requests to various routes. One of the important things that was required was to make sure that routes can be secured. This is where Passport.js came into the implementation. With Passport.js the user can be authenticated with the 'login' strategy we have defined which used an email and a password. Once the user is authenticated they start a session so they do not have to login every time they want to access a secure route. The session does expire after a particular amount of time. To protect routes there is a checkAuth function that can be used with any route. If the request is authenticated the function will allow execution to continue, otherwise it will redirect to the app root page.

```
var checkAuth = function (req, res, next) {
  if (req.isAuthenticated())
    return next();
  res.redirect('/');
}
```
[1]

The function can be used in a route definition as follows meaning that route is protected.

```
router.post('/userManagement/confirmUser/:badge', checkAuth, function (req, res) {
        ...
});
```
[1]

Another important part of the app was form validation. To help with form validation we have utilized express-validator. This helps the app easily verify valid emails, phone numbers, and other custom constraint fields.

```
router.post('/adminRegister', checkAuth, [
  check('email')
    .exists()
    .isEmail().withMessage('Please enter valid email')
    .trim()
    .normalizeEmail(),

  check('password')
    .exists()
    .isLength({ min: 8 })
    .withMessage('Passwords must be at least 8 characters long'),

  check('reenterpassword', 'Re-enter password field must have the same value as the password field')
    .exists()
    .custom((value, { req }) => value === req.body.password),
], checkRegistration, passport.authenticate('register', {
  successRedirect: '/logout',
  failureRedirect: '/adminRegister'
}));
```

[1]

The app can make sure when forms are submitted that fields exist and they meet constraints set in the app. If any a form fails to pass any of the constraints then a message will be flashed to the user indicating the error. In the code above once the form has been validated it will be sent to Passport.js for handling using the ‘register’ strategy to register an administrator into the system and then redirect them to the page depending on the success or failure of the operation. To render a page the app utilizes the call the render function.

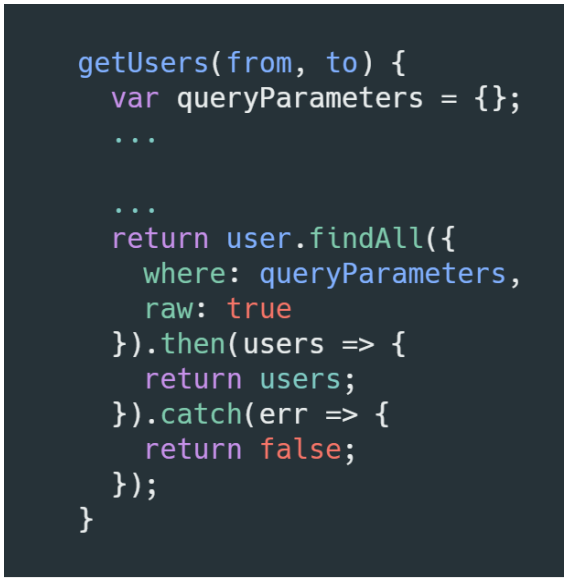
```
router.get('/adminRegister', checkAuth, function (req, res) {
  res.render('adminRegister.njk', { authenticated: true });
});
```

[1]

This is the basics of the frontend service. There are also similar implementations in the API part where no web pages need to be rendered. The API routes mainly deal with

getting data from a store about machine and user usage and the database to determine privileges to use various machines.

For the database there are a few pieces of important code. This web app uses Sequelize which is a Promise based Object Relational Model for Node.js. The database that the app uses is PostgreSQL and Sequelize allows the app to seamlessly communicate with the database. The models for our database are defined in the models folder inside the database folder. Each file defines the columns and datatypes for each column in the database. The relationships between the models which can also be thought of as tables are defined. The dbm.js file inside the controllers folder includes the implementation of the interface by which other parts of the system can use to retrieve various data such as a list of all users. Each function returns a Promise object meaning that the results may be retrieved at a later time and program execution continues.



[1]

During the project development there were not any critical unforeseen problems. The only issues that may have come up was the complexity with Promise objects. The

thinking here changes because it means that a piece of data will arrive at some later time and there needs to be a callback function defined to handle that data once it arrives. There have also been cases with a lack of sufficient documentation with things such as Promise objects and using Sequelize queries. Currently the app also uses an in memory session store. This store stores information about active authenticated sessions. If the server is rebooted then all session information is lost. An attempt was made to implement a more robust session store in the database, but due to complexity and time constraints this was not achieved. Besides these unforeseen problems the implementation of the app was satisfactory for the purpose.

6. Results and Evaluation

Most or all of the features specified in the requirements section have been met by the current implementation of the LUCCA system. There were automated and manual system tests that have been conducted. The developer tests included in the repository test parts of the code to ensure they work as desired and are verified. Most of the testing on the system that has occured have been manual. Our of developers tested the frontend and API functionality. Given the time constraints the critical paths in the system were identified and a combination of paths were constructed. Each path was tested in the system to ensure there was no undesired or unexpected behavior. Any unexpected or undesired behavior was patched. The system was then presented to the stakeholders, specifically the EPL sponsors. The capabilities of the system were

demonstrated to the sponsors and the sponsors have approved the system to be ready to be implemented into the EPL.

7. Future Work

The future of the system depends on how the system will be maintained in the future. Dependencies in the system should be updated if updates are available and are compatible with other dependencies in the project. This is especially important if vulnerabilities are discovered in the project. The way frontend dependencies are handled can be improved upon. Currently content delivery networks are used to retrieve dependencies when pages are served to the client. It may be more wise to download the dependencies onto the server and store them there and serve those dependencies directly to the client. In addition to this the features on the administrator user interface page can be improved. More reporting and querying options could be added to the system. The system could keep track and calculate more statistics on user usage of the lab and machines and display graphs to represent usage. This may mean that the capabilities of the database interface would need to get expanded. More functions would need to be added to do these statistical calculation which would include writing various queries. Currently the system uses PostgreSQL, a relational database system. A NoSQL database option could be explored as a usage for the system.

8. Conclusions

Overall the goals of the system have been successfully reached. The sponsor was satisfied with the features that were implemented based on the requirements. This paper has provided the necessary background to understand what kind of system our team was trying to develop. The aim and goals for this project were clearly defined to build LUCCA. This paper has discusses the specification and design of the system. These are crucial parts as they will lay the framework and foundation to have been able to successfully implement the system. Then the implementation of the system was discussed specifically focusing on the frontend user interface, API, and database. This gives insight and discusses the actual code that went into building the system. The system has been completed and delivered to the sponsors but there is always more to work and improve upon this system in the future.

10. References

1. Daniel Eynis, Bishoy Hanna, Bryan Mikkelson, Justin Moore, Huy Nguyen, Michael Olivas, Andrew Wood, LUCCA, (2018), GitHub Repository https://github.com/LUCCA-Capstone/LUCCA
2. Daniel Eynis, Bishoy Hanna, Bryan Mikkelson, Justin Moore, Huy Nguyen, Michael Olivas, Andrew Wood, LUCCA Station API Specification Proposal, (2018)
3. Daniel Eynis, Bishoy Hanna, Bryan Mikkelson, Justin Moore, Huy Nguyen, Michael Olivas, Andrew Wood, LUCCA EPL Requirements, (2018)
4. Daniel Eynis, Bishoy Hanna, Bryan Mikkelson, Justin Moore, Huy Nguyen, Michael Olivas, Andrew Wood, LUCCA Database Request Analysis, (2018)