**NUA**

Student Portal

Joshua Leivers | 6033

Joshua Leivers

Contents

[1 Analysis 3](#_Toc21805503)

[1.1 Introduction 3](#_Toc21805504)

[1.2 Existing Systems 3](#_Toc21805505)

[1.2.1 Moodle 3](#_Toc21805506)

[1.2.2 SIMS 4](#_Toc21805507)

[1.2.3 Website 5](#_Toc21805508)

[1.3 End Users 7](#_Toc21805509)

[1.4 Objectives 8](#_Toc21805510)

[1.5 Constraints and Limitations 11](#_Toc21805511)

[1.6 Proposed Solution 11](#_Toc21805512)

[1.6.1 Languages 11](#_Toc21805513)

[1.6.2 Frameworks 12](#_Toc21805514)

[1.6.3 Developer Environment 12](#_Toc21805515)

[1.6.4 Database Engine 12](#_Toc21805516)

[1.7 Evidence of Analysis 13](#_Toc21805517)

[1.7.1 Interviews 13](#_Toc21805518)

[1.7.2 Questionnaires 14](#_Toc21805519)

[1.7.3 Online Research 16](#_Toc21805520)

[2 Design 18](#_Toc21805521)

[2.1 Database 18](#_Toc21805522)

[2.1.1 Users 18](#_Toc21805523)

[2.1.2 Requests 19](#_Toc21805524)

[2.1.3 Permissions 21](#_Toc21805525)

[2.1.4 Sessions 22](#_Toc21805526)

[2.1.5 Detentions 23](#_Toc21805527)

[2.1.6 Grades 24](#_Toc21805528)

[2.1.7 User Classes 24](#_Toc21805529)

[2.1.8 Classes 24](#_Toc21805530)

[2.1.9 Feedback 24](#_Toc21805531)

[2.1.10 Feedback Access 25](#_Toc21805532)

[2.1.11 Announcements 25](#_Toc21805533)

[2.1.12 Homework 25](#_Toc21805534)

[2.1.13 Previous Sessions 25](#_Toc21805535)

[2.2 Report Handling 25](#_Toc21805536)

[2.2.1 Report Scheduling 25](#_Toc21805537)

[2.2.2 Command Reporter 26](#_Toc21805538)

[2.2.3 Export Scripts 26](#_Toc21805539)

[4 Testing 30](#_Toc21805540)

# 1 Analysis

## 1.1 Introduction

A student portal is an online gateway and system for providing students of a school, college or university with access to important academic information, resources and statistics (such as grades, attendance, announcements, etc.), software for download and use, links to other academic websites, and useful webpages.

Such a portal would usually use an existing login system/database, as well as linking in with the registration/academic records of the school, but this may conflict with GDPR or the Data Protection Act.

The system behind a student portal would use a database of connected tables, a web application and a script/program of some kind to either link into existing academic databases/information, or to export information from the existing records and import them into the student portal’s database.

## 1.2 Existing Systems

NUAST currently uses Moodle, SIMS, Office 365 and a website. SIMS and Office 365, as well as the school network/computers, are connected to shared resources with the Nova Education Trust, which the academy is a part of, meaning the trust’s policies apply.

Except the website, which only provides general information and forms to fill in, and does not require logins for anything, all of the school’s systems share a login system. This login system is Microsoft’s Active Directory, and while it is technically possible for my project to use this system for users to log in, the school/trust’s policies do not allow this.

### 1.2.1 Moodle

Moodle allows teachers to set up “courses” which they can add students to, where the teachers can upload links and files for students in the course to download. This covers part of what a student portal should do, and so my project should focus more on the other aspects of it.



Figure .2.1.1 Moodle Course Overview

Moodle also allows teachers to add announcements, news and events, however the implementation of this is not effective, as each individual course has an announcements section and users are not notified in any way if there are new announcements.



Figure 1.2.1.2 Moodle Course Example

### 1.2.2 SIMS

SIMS is the registration system used by the academy, which records student attendance, detentions and academic information – including student timetables. For anything relating to these statistics and information, my project will probably need to link into or work with SIMS in some way. Staff mainly use PARS for interacting with SIMS, but SIMS is the system that controls everything and that my project would need to interact with.



Figure 1.2.2.1 SIMS Interface

SIMS is primarily an information management system, based on a large database containing student information. My project ideally would link with this database, rather than interact with SIMS – as this would easily allow my system to access up-to-date info such as a student’s attendance without taxing resources. However, SIMS’ database operation is subject to change between updates and is deliberately not easily accessible. Instead, SIMS is able to automatically generate reports, so I may be able to either schedule reports to be exported, which can then be accessed by my system, or set up a script that does this instead.



Figure 1.2.2.2 SIMS Report Options

Linking into SIMS would require high standards of data protection and cyber security, as confidential information is stored in SIMS, and my system should be secure and only allow users access to information they are allowed to access.

The school/trust is currently planning on changing to a different information management system next year, which likely will mean a different database structure and report system. This would require certain parts of my project to be changed if the school implements it in the future, specifically anything that requires access to the database or SIMS reports, but the existing features would almost certainly be available within the new system, and would only need small query changes and/or new report scheduling (as well as report handling).

### 1.2.3 Website

NUAST’s website is mainly used for sharing general information with the public and with parents. This means that no information is about any specific student, so most of the information it provides would be different to what my project would provide access to.

In terms of tools, the website only provides forms for anyone to fill in. This includes a contact form and a form for people to send any concerns they have to staff. This form does part of what I wish for my project to do, however it is potentially unsecure, as it uses email to send form responses, and the form is not widely known to students.



There are also forms to fill in to give general feedback on different topics such as homework, however as these use SurveyMonkey, and no verification of identity, anyone can submit as many of the questionnaires as they want, skewing results and/or rendering submissions useless.



## 1.3 End Users

There are varying end users for this project. The main end user would be the students, who the majority of systems within the project would be aimed towards. However, there would be other end users, these being teachers and other school and trust staff. These users would be able to access different tools and parts of the application depending on their roles.

For example, a computer science teacher who is also part of the safeguarding team would be able to interact with computer science students within tools, and would also be able to access tools available to the safeguarding team (e.g. see safeguarding reports), but would not be able to view their own attendance, as they are not a student.

Currently, students do not have a way to access their academic information without asking teachers to manually find it, or from physical reports, which are not released often. They also do not have a way to make suggestions or report issues securely, nor do they have a way to easily see what homework has been set, information about it, or easily receive or set reminders about it.

Staff do not have an easy method to send a message to different groups of people within the school or to receive reports and suggestions.

Teachers do not have a method to set homework for their students online for them to view, or to set reminders of homework for them.

## 1.4 Objectives

|  |  |  |
| --- | --- | --- |
| No. | Objective | Performance Criteria |
| 1 | Create a secure login system to access the student portal | This must use salting and hashing, with each user having their own random salt, to login. Every time a password is entered, it must be salted and hashed using a secure hashing algorithm, which will be Bcrypt. Passwords must not be stored unsalted and unhashed anywhere. Each user’s salt must be randomly generated upon the creation of the account and be stored within the login information table.  No tools or information should be accessible unless the user is logged in. Logins should persist unless the user logs out, is logged out, their browser user agent changes to significantly, changes occur to the account that require the user to be reverified (e.g. password changes), or the account is disabled, deleted or suspended.  Any previous or active login should be recorded, including the time of login, time of logout, IP address and user agent. |
| 2 | Data protection legislation and policies should be adhered to and data protection should be done to the highest reasonable standards | No user should be able to access data that they are not authorised to access. Users should only be able to access their own information, and if they are a staff member, only allowed to access information for students and other staff where it is appropriate, legal, and makes sense. |
| 3 | Students should be able to view their own attendance information, timetables and grades from SIMS | This information must be obtained from SIMS, and kept reasonably up to date, with information updated at least every 24 hours, except under certain conditions, such as technical issues, and if the students are on a break from school. |
| 4. | Users should be able to receive notifications through the web app | These notifications should appear within the web app, and be sent in push notifications where possible (such as through Google Chrome). |
| 5. | Users should be able to send notifications to other users through the web app where appropriate | Users must only be able to send notifications to other users appropriate to their relationship. For example, a teacher to the classes and students they teach, a student council head of year to students in their year, a head of department to teachers and students within their department, and the head teacher to every student and member of staff. Sent notifications, their recipients, and when they were sent, should be logged. |
| 6. | Every person within the school should have their own account within the web app | Every student, teacher and staff member should have their own account, with the username being the same as their username within other school systems. Each account should have set permissions depending on the user’s role within the school. These accounts should either be created automatically based on information from school systems, or be created upon request, requiring new users to verify their new account via school email. |
| 7. | Users should be able to set their own password | Users must be asked to set a password upon account creation/first login. They must be able to change this password at any time from the web app. No other user should be able to set another user’s password. Technicians/admins should only be able to trigger a password reset, not set a new password. When a new password is set, the information on how it was set should be logged, including login session ID, time, and method. |
| 8. | Users should be able to reset their password if they are unable to get into their account | A password reset should be able to be triggered by a ‘forgot password’ function, and by a technician/admin. Once a reset is triggered, an email should be sent to the user’s school email, including a link. If the user clicks on the link, they should be taken to a page to create a new password. If they do not click the link, their password should remain unchanged. A user should be able to click a different link in the email to mark a reset request as suspicious, and to add a message, which technicians/admins should be able to view and take action on, by either doing something about it if it is a technical issue, or by notifying other appropriate staff members if it is either a safeguarding or punishment issue. Password reset requests should be logged, including the method they were triggered, who by and what session ID (when by technicians), IP address and user agent (when from ‘forgot password’), and whether they were confirmed. |
| 9. | Passwords must be secure | When a user attempts to set/change their password, the new password should be denied if it:   * Does not contain a lowercase letter, uppercase letter, number and special symbol * Matches a dictionary of common/breached passwords (possibly using the haveibeenpwned API) |
| 10. | Users must be asked to reverify their password to access confidential information, change important/security settings, or to access secure tools | Whenever users attempt to access anything that is confidential/sensitive, or to change any important/security settings for the system or for their account, they must be prompted to re-enter their password, and two-factor authentication code if enabled, to be able to access it. If they enter an incorrect password three or more times, they should be forcefully logged out and the incident logged for the user and technicians/admins to see. Otherwise, if they cancel accessing that tool/information, they should be returned to the previous page but not logged out. |
| 11. | Users must be able to enable two-factor authentication on their accounts | This two-factor authentication should use TOTP 2FA, specifically allowing the use of Google Authenticator. Users must be able to enable or disable this within their account settings. |
| 12. | Technicians/admins and leadership staff (e.g. head teacher, head of sixth form) should be able to disable users accounts | In case of an account breach, investigation, or other circumstances, technicians and high-level staff members should be able to disable users accounts, leaving the accounts still there, but not allowing users to log into them. To disable the account, the user doing so must give a reason, which should be shown to the anyone attempting to log into the disabled account. Technicians and high-level staff members should then be able to re-enable disabled accounts.  Whenever account is disabled or re-enabled, the account it is done to, the user that did it, the time, session ID and reason. |
| 13. | Upon account creation, permissions should be set for the new account automatically where possible | The system should be able to identify if the user is a student, teacher or staff member, who they are on SIMS, and from that identify what classes they teach/have, retrieve academic information for the user where applicable, and assign permissions based on their role within the school. A high-level staff member or technician should be able to change, add or remove permissions in case the user should/shouldn’t have permissions but that information isn’t available from SIMS. |
| 14. | Students should be able to report issues and make suggestions using the web app | Students should be able to report different types of issues and make suggestions, being able to select a category, department, etc. and then write a message explaining their issue/suggestion. For example, a student could report a safeguarding issue, in which case safeguarding staff would be able to read it, or suggest new equipment/services to the computer science department, or report maintenance issues.  Students should have the option for these reports and suggestions to be anonymous, in which case the origin of the report/suggestion should not be recorded, with staff receiving it only seeing the message and that the student is anonymous. There should be no way to circumvent this. |
| 15. | Teachers should be able to set homework for students to see using the web app | Teachers should be able to set homework for specific students or classes they have, including the due date and any information about the homework (i.e. what the homework is). They should be able to set reminders about the homework for themselves and for the students it is set for. They should automatically be reminded at the start of the lesson it is due for.  Teachers should also be able to mark set homework as received, late, cancelled, or not handed in. |
| 16. | Students should be able to view homework set for them using the web app | Students should be able to see what homework has been set, when it is due for, what teacher set it, when it was set, who was it set for (which class or group of students) and any extra information about the homework.  Students should also be able to set reminders for themselves about the homework set, add notes about the homework set, and mark whether they have completed it, have started it, or haven’t started it (should be set to haven’t started by default) |
| 17. | The web app should be secure against attacks | The web app should be secure against SQL injection, XSS and XSRF attacks |

## 1.5 Constraints and Limitations

* Access to student information must comply with data protection legislation (mainly GDPR and Data Protection Act)
* There is no readily available SIMS API and accessing the database directly is not possible due to updates changing its structure
* NUAST/Nova Education Trust’s policies may not allow access to some tools, software and/or information

## 1.6 Proposed Solution

A web app written in Python using Flask that uses an SQL database to store login, security, settings and other information uploaded to the app. This web app would use a secure login system, using the bcrypt hashing algorithm, to be able to access any tools or information. In addition to this, scripts and/or scheduled tasks would be set up to export student information from SIMS every 24 hours to the app’s database for students to access. The app would allow students to report issues and make suggestions in different categories (e.g. physics, safeguarding, maintenance).

### 1.6.1 Languages

As the majority of my project will be focused on the web application, the web app will have to be programmed in one of the five language options for the NEA. Out of these, Python and Java are very popular for web development.

Comparing the two, and from research online, I have found that Python is much more modern, and is often favoured due to its high-speed delivery and low overheads. There is a very active developer community based around Python web development, whereas Java web development tends to be based around enterprises, so it would likely be much easier to find information and documentation for Python-based web development. This should cut time down for programming greatly.

On top of the web app, there also may need to be scripts run for exporting SIMS information to my database, and Python would allow this to be done very lightweight.

NUAST’s computers also tend to have issues running some parts of Java, but have no issues running Python.

Together with the web app and scripting, there would also need to be web pages to display, which would need to be written in HTML, however for styling, I intend to use the CSS from NUAST’s website to keep styling consistent, perhaps with some minor changes to allow dark and light modes.

In conclusion, Python would likely be the best choice of programming language for the web application and scripting, along with HTML for web pages, and some minor use of CSS for alternative styling.

### 1.6.2 Frameworks

There are two very popular frameworks for Python web development: Flask and Django. Django is has been around much longer and is much more popular than Flask, however Flask has massively less overhead and tends to be much faster. In exchange for the overhead, Django also provides a lot more functionality, however much of it would not be useful for my project.

In conclusion, the best web framework for Python to use for my project would be Flask.

### 1.6.3 Developer Environment

As the entirety of my project will be based around Python, with side elements of HTML and CSS, using a full IDE would only complicate development, and so I intend to use a text editor, then run files separately.

For a text editor, I intend to use Atom, as it also provides some IDE functionality that may be useful for development while still being lightweight and easy to use.

To test my project, I will use Google Chrome, Mozilla Firefox, Internet Explorer, and Microsoft Edge. Together, these browsers take 72.76% of the overall usage share of web browsers, 90% of the desktop web browser usage share, and 60.87% of the mobile market share.

Another browser, Safari, is very popular within the Apple ecosystem, and has an overall usage share of 15.15%, desktop usage share of 3.32%, and mobile usage share of 20.12%, and would be useful to include in my testing. However, I do not have access to any device which can use Safari, and so I am unable to test this actively. A lot of what works within most other browsers, but especially Chrome (as it uses Blink, a fork of WebKit which is Safari’s engine) should work within Safari.

### 1.6.4 Database

#### 1.6.4.1 Assumptions

The database must be able to run using hardware and software that the school/trust has access to. It must be run on-site and be readily available for technical staff to access physically and to connect to so where anything goes wrong, problems can be fixed quickly to prevent extended downtime or user issues. The school would not be able to purchase enterprise-level hardware and software for the operation of the database due to limited funding and, given the scope of the database, would not need to. Therefore, the database must be able to run without being resource-intensive, and must run using either the software and hardware already available to the school, or that which is cheap to buy and operate. The school has access to Windows 10, which would be preferable for use due to the existing infrastructure within the school and its trust relying on it. The database must be able to operate continuously and reliably so that staff and students can access information from it at any time.

#### 1.6.4.2 Constraints

The database must run on Windows 10, using software and hardware either already or cheaply available to the school that is compatible with existing systems. It must be able to run continuously without interruptions. It must comply with data protection regulations due to the handling of students’ data.

#### 1.6.4.3 Risks and Mitigation

As the database will handle private student information, it must be able to keep this information secure and only allow access to information where the person accessing it is legally allowed to access it. If the database is insecure in any way, it may allow unauthorised users to access private information, which would be illegal both for the user and the school for allowing it to be insecure. If this does happen, the school could face fines and other legal ramifications, as could I due to my creation and involvement with the system. As such, access to the database should be strictly controlled, with any systems and staff with direct access to the database only having the permissions they are legally allowed and that they need (e.g. technical staff only have access to schema and other things they need access to unless they *need* access to the actual information in the database). The system the database is hosted on should be strictly access-controlled.

#### 1.6.4.4 Engine

The database will need to store private information. This data also needs to be quickly displayed through the web application. As such, the engine should allow for secure but fast and reliable access. I also need to be able to find documentation and support for the engine easily.

From research online, I have found that there are many database engines that I could use, including Oracle 12c, MongoDB, MariaDB, and MySQL. However, I have chosen to rule out several, including Oracle 12c and DB2 due to their high pricing. I have also ruled out Microsoft SQL Server, as it is known to be resource-intensive.

My remaining are as follows:

* MongoDB
* MariaDB
* PostgreSQL
* MySQL

MongoDB is very popular, however it does not use SQL, takes a long time to setup, and can be insecure if not configured correctly. MariaDB is relatively new, and so does not have as great of an established community as others that I can find answers from. PostgreSQL itself looks promising for my requirements, however its documentation is unreliable. This leaves me with MySQL, one of the most popular database engines, which is free and reliable, with a large online community to find answers from where needed alongside the documentation, and while it does not have some functionality that other engines do, what is missing is not required in this project.

In conclusion, I will be using MySQL as the database engine for my project.

## 1.7 Evidence of Analysis

### 1.7.1 Interviews

#### 1.7.1.1 IT Technician

**Q.** What does the school use for a login system?

**A.** Microsoft’s Active Directory

**Q.** Would it be possible to link my student portal with the school’s login system?

**A.** No. Nothing can connect to the school’s systems.

**Q.** When is the school changing away from SIMS?

**A.** Next year.

**Q.** Are there any restrictions to staff email usernames? [NOTE: Staff emails all follow the format ‘<first initial><surname>@nuast.org’.]

**A.** No, they all follow the format exactly.

#### 1.7.1.2 Head of Computer Science

**Q.** Does the school have access to CommandReporter for SIMS?

**A.** Yes

**Q.** Can SIMS schedule reports for teachers?

**A.** Yes. Using PARS, teachers can even have reports automatically emailed to them on a schedule.

### 1.7.2 Questionnaires

#### 1.7.2.1 Students’ Opinions on Usefulness











### 1.7.3 Online Research

#### 1.7.3.1 SIMS

From doing research online, I found that SIMS’ database structure tends to change between updates, and so hooking directly into the database in my app would likely only work for a short time, before a new update changes the database structure and breaks any queries within the changed tables.[[1]](#footnote-1)

In general, the suggested way to access information from SIMS is to export reports (this can be scheduled using Task Scheduler), have them converted to CSV files by CommandReporter, then import them into the system that the user wants the data in.[[2]](#footnote-2)

#### 1.7.3.2 Website

The website does not provide many tools, mainly providing general public information about NUAST, however there are forms that can be filled out. One of these is a form for reporting issues, optionally anonymously. However, this form sends the results via email[[3]](#footnote-3), which can be insecure during transit – especially if unencrypted – and upon receipt.[[4]](#footnote-4)

# 2 Design

My project will be split into three components that interact with each other. These components will be the database, report handling and web application.

The report scripts will gather information from reports exported by SIMS and upload it to the database.

The web application will contain a secure login system and allow users to access tools and information appropriate to their role within NUAST.

## 2.1 Database

The database will store multiple tables. These tables will store account, personal, and academic information about users of the web application, as well as other data necessary for the operation of the web app. The database will use MySQL.

The database can be created using the following query:

**CREATE DATABASE IF NOT EXISTS portal**

### 2.1.1 Users

The users table will store any account and identification information. The username will be used as the primary key, as every username must be unique. The password will be given in a salted and hashed form by Bcrypt within the web app, and does not need to be given during initial account creation, or when a password reset is forced. The ‘disabled’ value will determine whether an account can be logged into through the web app.

#### 2.1.1.1 Structure

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Field Name | username | password | forename | surname | disabled | attendance | otp |
| Type | VARCHAR | CHAR | TEXT | TEXT | TEXT\*\*\* | DECIMAL | CHAR |
| Size | 64\* | 60\*\* | N/A | N/A | N/A | 5, 2\*\*\*\* | 120\*\*\*\*\* |

\*set based on the limit of 64 characters in the username of an email

\*\*Bcrypt always produces a 60-character result

\*\*\*This will store information about an account being disabled (in the format ‘<admin username>:<show admin [0/1]>:<message>’).

\*\*\*\*5 digits total, 2 to the right of the decimal point, representing a percentage

\*\*\*\*\*A 16-digit base32 TOTP secret, encrypted using Fernet, is always 120 characters in length.

#### 2.1.1.2 SQL

The users table can be created using the following query:

**CREATE TABLE IF NOT EXISTS users ( username VARCHAR( 64 ) NOT NULL, password CHAR( 60 ), forename TEXT, surname TEXT, disabled TEXT, attendance DECIMAL( 5, 2 ), otp CHAR( 120 ), PRIMARY KEY ( username ) )**

The users table can then be searched using the following queries:

By username: **SELECT <columns> FROM users WHERE username = “<username>”** #USAGE: This would be used anywhere where a specific user’s information needs to be retrieved. This would be used when a user attempts to log in. #NOTE: When searching for a user in a list, such as staff members searching for a student through the web application, the search should use ‘LIKE’ instead of ‘=’.

By name: **SELECT <columns> FROM users WHERE username = “<f><surname>” OR username LIKE “4004<fo><su>\_\_”** #USAGE: This would be used to search by a given name, as long as the required length of each name is given. If, for example, the length of the given forename was one character, then the ‘4004<fo><su>\_\_’ search would be changed by the code it is called from to ‘4004<f>\_<su>\_\_’. #NOTE: This is much more efficient than searching the forename and surname fields, as TEXT fields cannot be indexed in MySQL, so searching within them is slow. Having the web app or other code format a name into the two allowed username formats, then search based on those will be much quicker.

By whether account is disabled: **SELECT <columns> FROM users WHERE disabled <> NULL** #CONTINUE: UPDATE THIS TEXT #USAGE: This would be used to retrieve information on all disabled accounts, such as by staff members on the web app with the appropriate permissions to see which accounts are disabled, and . The text itself will follow the format ‘<admin username>:<show admin [0/1]>: <message>’, where ‘<admin>’ would be the admin that disabled the account, ‘<show admin [0/1]>’ would be a 1 or 0 (representing true or false) that would determine whether the admin that disabled the account would be shown to the user when they attempt to log in. ‘<message>’ would always be shown to the user if they attempt to log in. If ‘<message>’ is blank, then it will simply show that the account is disabled (and the admin if ‘<show admin>’ is 1).

By empty information: **SELECT <columns> FROM users WHERE <column> = “” OR <column> IS NULL** #USAGE: This would be used by the scripts used to upload information from scripts to check if, for example, a user’s name has not been retrieved or set yet, and then any users with missing names would have their info added by an UPDATE query. #NOTE: This statement assumes that both an empty string and a properly NULL value are possible. If possible, using the check for an empty string alone would be quicker.

Users created using the web app would be created using the following query:

**INSERT INTO users ( username ) VALUES ( “<username>” )** #USAGE: Once a user clicks on the link sent to their email, validating their new account, this query would be used to create their permanent account. This could also be used by staff with the appropriate permissions to create accounts within the web app for new users. Once users set a password either through the page opened by the validation link, or by attempting to log in and being sent a password reset email due to a lack of password, their password will be set by an UPDATE query.

Updating data within the users table would be done using the following query:

**UPDATE users SET <column> = <data>, <column> = <data>, … WHERE <condition>**

Including:

Setting a new password: **UPDATE users SET password = “<password>” WHERE username = “<username>”** #USAGE: This would be used for when a user resets their password successfully, or changes their password.

Setting a user’s forename and surname: **UPDATE users SET forename = “<forename>”, surname = “<surname>” WHERE username = “<username>”** #USAGE: This would be called by a script that automatically checks if users’ names are up to date with the latest SIMS report of names.

Enabling/disabling an account: **UPDATE users SET disabled = “<admin username>:<show admin [0/1]>:<message>” WHERE username = “<username>”** #USAGE: This would be called by the web app when a user with the appropriate permissions disables or enables an account. The user disabling an account would do so through the web app, with ‘<admin username>’ being their own username automatically added, ‘<show admin [0/1]>’ being a 0 or 1 (representing false or true), set by the user disabling their account by a checkbox, and ‘<message>’ being entered by the user into an optional text box, with all leading and trailing spaces being removed upon submission.

Updating a user’s attendance: **UPDATE users SET attendance = <attendance decimal>** #USAGE: This would be called by a script that would automatically update every user’s attendance to the latest SIMS report’s.

Setting a new 2FA secret: **UPDATE users SET otp = “<new encrypted otp secret>” WHERE username = “<username>”** #USAGE: This would be called by the web application when a user sets up 2FA successfully (‘2fa’ itself would be a random base32 string of length 16, encrypted using Fernet from the Cryptography library[[5]](#footnote-5) and a key hardcoded into the web app to prevent people with access to the database from simply reading the secrets. This way, if the database is breached, as long as the web server is not also breached, it would take too long to find the key to decrypt them before users could be warned by the web app and forced to update their 2FA).

Deactivating 2FA: **UPDATE users SET 2fa = NULL WHERE username = “<username>”** #USAGE: This would be called by the web app when a user disables 2FA on their account after the user confirms their login information.

Deleting users from the table would be done using the following queries:

By username: **DELETE FROM users WHERE username = “<username>”** #USAGE: This would be called by the web app when a staff member with the appropriate permissions deletes a user’s account. At the same time, other deletion queries would be called to remove all data related to that user and their account, and any files related to the user (e.g. timetables, reports) would be deleted.

By year: **DELETE FROM users WHERE username LIKE “4004%<year id number>” AND NOT username IN ( “4004<fo><su><year number>”, … )** #USAGE: This would be called by the web app when a staff member with the appropriate permissions deletes an outgoing year of student’s accounts with the exception of specified users that are being held back a year.

### 2.1.2 Requests

The requests table will store the username, link ID, link creation time, and browser user-agent of new user and password reset requests. Rows will automatically be deleted a certain amount of time after creation if a link is not used, or once a link is used. The primary key will be the linkid, which must be unique to work, but systems should be in place to limit the amount of requests a user can make (i.e. a cooldown should be in place, if a certain number of links have been created, no more can be created until one expires).

#### 2.1.2.1 Structure

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Field Name | username | requestid | creationtime | useragent | ip |
| Type | VARCHAR | CHAR | DATETIME | TEXT | INT UNSIGNED |
| Size | 64 | 32 | N/A | N/A | 4\* |

\*IPv4 addresses can be stored as 4 byte unsigned binary integers. This is the most efficient way to store them.

#### 2.1.2.2 SQL

The requests table will be created using the following query:

**CREATE TABLE IF NOT EXISTS requests ( username VARCHAR( 64 ) NOT NULL, requestid CHAR( 32 ) NOT NULL, creationtime DATETIME DEFAULT CURRENT\_TIMESTAMP, useragent TEXT, ip UNSIGNED INT( 4 ) PRIMARY KEY ( linkid ) )**

Alongside this statement, the following should be run to create an event that wipes unused links on a schedule:

**CREATE EVENT remove\_unused ON SCHEDULE EVERY 1 DAY DO DELETE FROM requests WHERE creationtime < DATE\_SUB( NOW(), INTERVAL 30 MINUTE )** #USAGE: This event would run once a day, every day, and remove any requests that are more than 30 minutes old at the time of the event running. #CONTINUE: TEST THAT THIS WORKS

Searching this table can be done using the following queries:

By username: **SELECT <columns> FROM requests WHERE username = “<username>”** #USAGE: This would be called by the web app to find information on requests relating to a given username. This would be used on a page within the user’s settings to show the user their own active requests, and by a page that staff with the appropriate permissions could access to check other users’ requests (usernames that match this condition would be displayed first, followed by those that match ‘WHERE username LIKE “%<username>%”’).

By user classes: **SELECT r.<columns> FROM requests AS r INNER JOIN classes AS c ON r.username = c.username AND c.classid = “<classid>”** #USAGE: This would be used in a page staff with appropriate permissions could access to show all requests within a given class. (ClassIDs that match this condition would be displayed first, followed by those that match ‘WHERE c.classid LIKE “%<classid>%”’).

By linkid: **SELECT <columns> FROM requests WHERE requestid = “<requestid>”** #USAGE: This would be used when a request link is clicked or the ID is entered manually to find if that specific request does exist, or if the request ID is searched by a staff member, this would be used to return the matching requested. (requestids that match this condition would be displayed first, followed by those that match ‘WHERE requestid LIKE “%<requestid>%”’).

By ip: **SELECT <columns> FROM requests WHERE ip = <ip int>** #USAGE: This would be used

Requests created through the web app would use the following query:

**INSERT INTO requests ( username, requestid, useragent, ip ) VALUES ( “<username>”, “<random 32-character id>”, “<useragent>”, <ip int> )**

The following queries would be used to delete data based on different conditions:

All expired: **DELETE FROM requests WHERE creationtime < DATE\_SUB( NOW(), INTERVAL 30 MINUTE )** #USAGE: This would be used whenever a staff member attempts to search the list of active requests, with this query being called before the SELECT query is called to remove all expired queries. It would also be used whenever a user attempts to confirm a request, being called before the SELECT statement for the request is called.

By requestid: **DELETE FROM requests WHERE requestid = “<requestid>”** #USAGE: This would be called when a user successfully confirms a request, and only once the request is fully complete, so if the process is interrupted (e.g. by a closed browser or an internet connection getting interrupted), it can be restarted within the allowed time, not cause any problems with the system, and not cause inconvenience to the user. This would also be called if a staff member with the appropriate permissions manually deleted a request

By username: **DELETE FROM requests WHERE username = “<username>”** #USAGE: This would be used where a user, through the web app or a link within the email sent upon a request being served, requests that all active requests be deleted. A staff member with the appropriate permissions could also run this on users they select through the web app.

By ip: **DELETE FROM requests WHERE ip = <ip int>** #USAGE: This would be called when a staff member with the appropriate permissions enters an IP address to delete the requests from (e.g. in the event that an IP [not the school IP] is generating too many requests).

### 2.1.3 Permissions

The permissions table will store pairs of usernames and permissionids. As this table describes a relationship between users and permissions, it will use a composite primary key of username and permissionid. There should be documentation stored and updated by admins to reflect what each permissionid should allow and who should have access to it.

#### 2.1.3.1 Structure

|  |  |  |
| --- | --- | --- |
| Field Name | username | permissionid |
| Type | VARCHAR | VARCHAR |
| Size | 64 | 16 |

#### 2.1.3.2 SQL

The permissions table can be created using the following query:

**CREATE TABLE IF NOT EXISTS permissions ( username VARCHAR( 64 ) NOT NULL, permissionid VARCHAR( 16 ) NOT NULL, PRIMARY KEY ( username, permissionid ) )**

To test if a user has a specific permission, use the following query:

**SELECT EXISTS ( SELECT 1 FROM permissions WHERE permissionid = “<permissionid>” AND username = “<username>” LIMIT 1 )** #USAGE: This would be used to check if a user has a permission. Returns TRUE/FALSE [1/0]. #NOTE: This is the most efficient way to check this, as the limit stops the database from searching needlessly once one instance is found, and only one instance can exist. The EXISTS statement then returns 1 if an instance is found, and 0 if one is not.

To find all users with a given permission, the following query can be used:

**SELECT username FROM permissions WHERE permissionid = “<permissionid>”** #USAGE: This would be used for a page in the web app where staff with the required permissions could search for all users with a given permission (this would list the users returned by this query, followed by a query using a LIKE condition rather than an equals), and for certain screens through the web app where users wish to request an action (e.g. changing their name), where the users with the permissions to make the change could be listed.

To find all permissions a user has, the following query can be used:

**SELECT permissionid FROM permissions WHERE username = “<username>”** #USAGE: This would be used in the web app to show a user their own permissions (if they have permissions to view their own permissions), and to show other users’ permissions on their profile pages (if the user requesting the page has permissions to view these permissions).

To add new pairs to the table, the following query can be used:

**INSERT INTO permissions ( username, permissionid ) VALUES ( “<username>”, “<permissionid>” )** #USAGE: This would be used when an account is either automatically identified as needing certain permissions (e.g. a student getting all student permissions), or when a user with the appropriate permissions manually adds permissions to other users.

If a permissionid is being changed, but all the users need to continue to have that permission, the following query should be used:

**UPDATE permissions SET permissionid = “<new permissionid>” WHERE permissionid = “<old permissionid>”** #USAGE: This would be used when a user with sufficient permissions in the web app changes a permissionid and checks a box for all current users to keep the permission.

If a user is changing username or a different user is taking over their permissions, the following query should be used:

**UPDATE permissions SET username = “<new username>” WHERE username = “<current username>”** #USAGE: This would be used whenever a username is changed within the web app to update the username within the permissions table, and whenever a user with the appropriate permissions transfers a specified user’s permissions to another user (e.g. a leaving member of staff’s permissions getting transferred to their replacement).

To delete permission pairs, the following queries can be used based on different conditions:

Individual pairs: **DELETE FROM permissions WHERE username = “<username>” AND permissionid = “<permissionid>”** #USAGE: This would be used whenever a permission needs to be removed from a specified user, either manually through the web app or automatically.

All from username: **DELETE FROM permissions WHERE username = “<username>”** #USAGE: This would be used to remove all permissions from a specified user (e.g. if the user’s account is being deleted, or if their permissions are being removed manually through the web app)

All from permissionid: **DELETE FROM permissions WHERE permissionid = “<permissionid>”** #USAGE: This would be used to remove a specified permission from all users (e.g. if the permission is being removed, or reset so that no users have it yet). It would be called from the web app by users with the appropriate permissions.

### 2.1.4 Sessions

The sessions table will store information on active sessions. The primary key will be the sessionid, which will be stored as a cookie in users’ browsers to identify individual browsers. The table will also store the username the session belongs to, the last time the session was active, the IP address it is active on, and the browser user-agent. Once the user logs out of the session, the IP address or user-agent changes from the original, or the session has been inactive for more than two hours, the session will be deleted from the table.

#### 2.1.4.1 Structure

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Field Name | sessionid | username | lastactivity | ip | useragent |
| Type | CHAR | VARCHAR | DATETIME | INT UNSIGNED | TEXT |
| Size | 64 | 64 | N/A | 4\* | N/A |

\*IPv4 addresses can be stored as 4 byte unsigned binary integers. This is the most efficient way to store them.

#### 2.1.4.2 SQL

The sessions table can be created using the following query:

**CREATE TABLE IF NOT EXISTS sessions ( sessionid CHAR( 64 ) NOT NULL, username VARCHAR( 64 ) NOT NULL, lastactivity DATETIME NOT NULL, ip UNSIGNED INT( 4 ) NOT NULL, useragent TEXT NOT NULL, PRIMARY KEY ( sessionid ) )**

Alongside the creation of the table, the following queries should be run to schedule automatic events:

The deletion of old sessions: #CONTINUE: UPDATE THIS TO NEW SESSIONS SYSTEM

**CREATE EVENT remove\_old\_sessions ON SCHEDULE 12 HOUR DO DELETE FROM sessions WHERE lastactivity < DATE\_SUB( NOW(), INTERVAL 45 MINUTE )** #USAGE: This would be in place to remove any sessions from the table that were last active more than 45 minutes ago that haven’t been removed by the web app (e.g. when a user attempts to access a new page, their expired sessions would be removed and then their current sessionID would be checked against the existing ones in the table).

To retrieve information about sessions, the following queries would be used for their relevant conditions:

By sessionid: **SELECT <columns> FROM sessions WHERE sessionid = “<sessionid>”** #USAGE: This would be used to retrieve information about a specific session. It would be used when a user with “<sessionid>” in their browser cookie attempts to load a page, after the query to delete expired sessions for that user is run, to make sure that the session cookie is valid, and if so, if the user’s information matches up with the session’s (e.g. that their IP address and browser user-agent hasn’t changed, else the session would be deleted and the user shown a page that explains that they have been logged out, and allows them to login again to resume their session [where possible]).

By username: **SELECT <columns> FROM sessions WHERE username = “<username>”** #USAGE: This would be used to retrieve information on all of a user’s active sessions. This would be used in a page that each user would be able to access showing their own active sessions, and within a search page available to staff with the appropriate permissions to show other, specified users’ sessions information (this search would show the results from this query, followed by one using a LIKE condition).

By ip: **SELECT <columns> FROM sessions WHERE ip = <ip int>** #USAGE: This would be used to retrieve information on all sessions active on a given IP address, and would be used within a search page that staff with the appropriate permissions would have access to (sessions retrieved by this query would be displayed first, followed by those retrieved by a similar query using a LIKE condition). When used here, this would not show sessionIDs, as that could potentially allow users to attempt to forge sessions and access other users’ accounts. The search page would be used by staff to find where users are potentially accessing other accounts without permission.

To update a session’s ‘lastactivity’ column, the following query should be used:

**UPDATE sessions SET lastactivity = NOW() WHERE sessionid = “<sessionid>”** #USAGE: This would be used whenever a user in an active session, after their session is verified, accesses a new page or is otherwise active, to keep the ‘lastactivity’ of the session up to date to prevent the session from timing out too early.

To delete sessions based on different conditions, the following queries should be used:

By sessionid: **DELETE FROM sessions WHERE sessionid = “<sessionid>”** #USAGE: This would be used to log out a specific session. This would be called if the user themselves manually logs out, the session times out, the user’s password changes (which will be verified every time the user loads a new page or attempts various tasks such has changing settings), the user is forcefully logged out by a staff member, or the user is otherwise logged out for any reason.

#CONTINUE: FROM THIS POINT

By username: **DELETE FROM sessions WHERE username = “<username>”**

If end at least a week ago: **DELETE FROM sessions WHERE end < DATE\_SUB( NOW(), INTERVAL 1 WEEK )**

By ip: **DELETE FROM sessions WHERE ip = <ip int>**

By useragent: **DELETE FROM sessions WHERE useragent = “<useragent>”**

### 2.1.5 Detentions

The detentions table will store detention information for every student. ??? CHECK HOW DETENTIONS ARE STORED IN SIMS

#### 2.1.5.1 Structure

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Field name | username | start | end | teacher | room | info |
| Type | VARCHAR | DATETIME | DATETIME | VARCHAR | CHAR | TEXT |
| Size | 64 | N/A | N/A | 64\* | 4 | N/A |

\*This will store the username of the teacher, and so will need the maximum of 64 characters of all other usernames

#CONTINUE: Research how detentions are stored.

#### 2.1.5.2 SQL

The detentions table can be created using the following query:

**CREATE TABLE IF NOT EXISTS detentions ( username VARCHAR( 64 ) NOT NULL, date DATE NOT NULL, period VARCHAR( 5 ) NOT NULL, teacher VARCHAR( 64 ) NOT NULL, length UNSIGNED TINYINT, room CHAR( 4 ), info TEXT, PRIMARY KEY ( username, date, period, teacher ) )**

### 2.1.6 Grades

The grades table will store all uploaded student grades, including from individual tests and for an overall course.

#### 2.1.6.1 Table Structure

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Field Name | examid | username | grade | examdate |
| Type | TEXT | VARCHAR | VARCHAR | DATE |
| Size | N/A | 64 | 6 | N/A |

#CONTINUE: Research how grades are stored.

#### 2.1.6.2 SQL

The grades table can be created using the following query:

**CREATE TABLE IF NOT EXISTS grades ( examid TEXT NOT NULL, username VARCHAR( 64 ) NOT NULL, grade VARCHAR( 6 ), examdate DATE, PRIMARY KEY ( examid, username ) )**

### 2.1.7 Classes

The classes table will store all classes a user is/has been a part of, and whether they are currently part of that class.

#### 2.1.7.1 Structure

|  |  |  |  |
| --- | --- | --- | --- |
| Field Name | classid | username |  |
| Type | VARCHAR | VARCHAR |  |
| Size | 10 | 64 |  |

#CONTINUE: Research max length of classid

#### 2.1.7.2 SQL

The userclasses table can be created using the following query:

**CREATE TABLE IF NOT EXISTS userclasses ( classid VARCHAR( 10 ) NOT NULL, username VARCHAR( 64 ) NOT NULL, PRIMARY KEY ( classid, username ) )**

To retrieve information about users’ classes, the following queries can be used:

<username>’s classes: **SELECT classid FROM userclasses WHERE username = “<username>”** #Usage: This would be used to retrieve all classes that a given user is a member of.

<classid>’s users: **SELECT username FROM userclasses WHERE classid = “<classid>”** #Usage: This would be used to retrieve all users that are a member of the given class.

If <username> has <classid>: **SELECT EXISTS ( SELECT 1 FROM userclasses WHERE classid = “<classid>” AND username = “<username>”** #Usage: This would be used to check if a user has a specified class.

To update user classes

### 2.1.8 #REMOVED

### 2.1.9 Feedback

The feedback table will store all feedback given by users on different teachers, subjects and facilities. Each instance of feedback will be given a unique ID, The usernames of the users giving the feedback will be encrypted if the feedback is not set to public, with the key being part of the web app, and only be decrypted if a user with the correct permission views the username.

#### 2.1.9.1 Structure

#### 2.1.9.2 SQL

The feedback table can be created using the following query:

### 2.1.10 Feedback Access

The feedbackaccess table will store all the times a user reveals a hidden username on feedback, which user did it, the feedback ID, the session ID and time.

#### 2.1.10.1 Structure

### 2.1.11 Announcements

The announcements table will store active announcements, who sent them, and their target.

#### 2.1.11.1 Structure

### 2.1.12 Homework

The homework table will store due homework and information relating to it, with each row being deleted after a certain amount of time has passed since the due date to prevent the table storing too much irrelevant data.

#### 2.1.12.1 Structure

### 2.1.13 Previous Sessions

The prevsessions table will store the data of recent sessions for a reasonable amount of time to allow users to see if their account has been accessed without their permission. The user will then be able to mark a session as suspicious, at which point the automatic deletion of the session information will be delayed, and admins/technical staff will be able to see suspicious sessions if they have permission.

#### 2.1.13.1 Structure

#### 2.1.13.2 SQL

## 2.2 Report Handling

The report handling component will include scripts, application usage and task scheduling to generate reports from SIMS, convert the data into the format required for use where applicable, and then upload the data or the report to the web server or database.

### 2.2.1 Report Scheduling

To schedule reports to be exported from SIMS, this component would need to be set up on the SIMS SQL server, with access to a SIMS account with either the School Administrator or System Manager user group and any other permissions required to access the appropriate data, under a separate Windows user account to anyone else, and have access to the SQL server drives.

#### 2.2.1.1 Task Scheduler

To setup a schedule for reports to be exported, a Windows Scheduled Task will be created per report which runs AutoReports.exe[[6]](#footnote-6) daily, outside of school hours to prevent overloading the server.

The parameters for exporting reports should look something like:

**AutoReports.exe /U:<SIMS username> /P:<SIMS password> /S:<SIMS SQL server name> /D:<SIMS SQL database name>**

#### 2.2.1.2 Report Exporting

To set which reports should be exported on the schedule, these reports should be selected under Focus 🡪 Alerts 🡪 Schedule Reports with the SIMS user for the schedule selected as the recipient.[[7]](#footnote-7)

### 2.2.2 Command Reporter

Included in SIMS is an application called Command Reporter which converts SIMS reports to CSV files.[[8]](#footnote-8) Having the reports in CSV format would allow the scripts within this component to easily grab the required information and upload it to where it is required. Command Reporter will be run by a scheduled task set after the task to export the reports to allow them to be made, with parameters like:

**CommandReporter.exe /USER:<SIMS username> /PASSWORD:<SIMS password> /SERVERNAME:<SIMS SQL server name> /DATABASENAME:<SIMS SQL database name> /REPORT:<SIMS report name> /OUTPUT:<CSV file output filepath>\<CSV filename>.csv**

### 2.2.3 Export Scripts

Finally, within this component, any information from files, and in some cases, raw files, will need to be uploaded to either the database or the web server. There will be a script for each scenario which will be run by Windows Task Scheduler after the two previous tasks are complete.

#### 2.2.3.1 Raw File Export

The timetable for each user will be exported to the web server as a raw file (in XLS format). This should follow the following pseudocode:

**domain 🡨 “FTP.EXAMPLE.COM”**

**port 🡨 21**

**directory 🡨 “timetables”**

**username 🡨 “FTPUSER”**

**password 🡨 “FTPPASS”**

**ftp = FTP.CONNECT( domain, port )**

**ftp.LOGIN( username, password )**

**ftp.DIR( directory )**

**files 🡨 FILES.GETFROMDIR( directory )**

**FOR file IN files DO**

**ftp.UPLOAD( FILE( file ) )**

**END**

**ftp.DISCONNECT()**

#### 2.2.3.2 Attendance Export UPDATE WITH SIMS MICROSOFT INFO

The script to export attendance will retrieve every student’s attendance percentage from the CSV file the report is stored in and upload it to the users table within the database (where students have an account within the web app).

This should follow the following pseudocode:

**file 🡨 FILE( “C:\path\to\attendance.csv” )**

**domain 🡨 “DB.EXAMPLE.COM”**

**database 🡨 “users”**

**username 🡨 “SQLUSER”**

**password 🡨 “SQLPASS”**

**db 🡨 SQL.LOGIN( domain, database, username, password )**

**FOR line IN file.LINES DO**

**values 🡨 line.SPLIT( “,” )**

**db.QUERY( “UPDATE users SET attendance = “ + values.GET( 1 ) + “ WHERE username = “ + values.GET( 0 )**

**END**

**db.LOGOUT()**

#### 2.2.3.3 Detention Export

The script to export detention information will retrieve every student’s detention data from the CSV files the detention reports are stored in and upload the information to the detentions table within the database (where students have an account within the web app).

This should follow the following pseudocode:

**file 🡨 FILE( “C:\path\to\detentions.csv” )**

**domain 🡨 “DB.EXAMPLE.COM”**

**database 🡨 “detentions”**

**username 🡨 “SQLUSER”**

**password 🡨 “SQLPASS”**

**db 🡨 SQL.LOGIN( domain, database, username, password )**

**FOR line IN file.LINES DO**

**values 🡨 line.SPLIT( “,” )**

**db.QUERY( “UPDATE detentions SET ( teacher, length, room, info ) VALUES ( “ + values.GET( 3 ) + “, “ + values.GET( 4 ) + “, “ + values.GET( 5) + “, “ + values.GET( 6 ) + “ ) WHERE username = “ + values.GET( 0 ) + “ AND date = “ + values.GET( 1 ) + “ AND period = “ + values.GET( 2 ) + “**

**IF @@ROWCOUNT=0 INSERT INTO detentions ( username, date, period, teacher, length, room, info ) VALUES ( “ + values.GET( 0 ) + “, “ + values.GET( 1 ) + “, “ + values.GET( 2 ) + “, “ + values.GET( 3 ) + “, “ + values.GET( 4 ) + “, “ + values.GET( 5) + “, “ + values.GET( 6 ) + “ );” )**

**END**

**db.LOGOUT()**

#### 2.2.3.4 Grades Export

The script to export grades from SIMS will retrieve all of the grading information from the CSV files the reports are stored in and upload it to the grades table in the database for the students that have an account on the web app.

This should follow the following pseudocode:

Flask Bcrypt, 72 chars password max unless workaround

User agent, use from user\_agents import parse, parse(useragent), store parsed form

## 4 Testing

A lot of testing will take place throughout the creation of my technical solution and some of my design. More testing will then take place once significant sections of my technical solution are complete, and then for the final product.

1. <http://www.edugeek.net/forums/mis-systems/63879-sims-database-structure.html> [↑](#footnote-ref-1)
2. <https://blogleedrury.files.wordpress.com/2019/03/command-reporter-user-guide.pdf> [↑](#footnote-ref-2)
3. <http://nuast.org.uk/page.php?p=pupil> [↑](#footnote-ref-3)
4. <https://support.google.com/transparencyreport/answer/7381230?hl=en> [↑](#footnote-ref-4)
5. <https://cryptography.io/en/latest/fernet/> [↑](#footnote-ref-5)
6. <http://faq.scomis.org/wp-content/uploads/2012/08/Setting_up_the_Report_Scheduler_11982.pdf> [↑](#footnote-ref-6)
7. <https://static1.squarespace.com/static/50e965eae4b0e6a1b5e24274/t/57e0e4f429687f793d09ff0b/1474356481627/Schools.pdf> [↑](#footnote-ref-7)
8. <https://blogleedrury.files.wordpress.com/2019/03/command-reporter-user-guide.pdf> [↑](#footnote-ref-8)