PATE'S GRAMMAR SCHOOL

COMPUTING DEPARTMENT

Unit 3/4 – Programming Project

CANDIDATE NAME

Pranay Begwani

EXAM NUMBER

_____

Contents

(1) ANLYSIS OF THE PROBLEM (10 MARKS)

(I) PROBLEM DEFINITION

(a) Describe and justify the features that make the problem solvable by computational methods.

Describe your solution. What is it? Why do they need a solution? Why is better than the solution that they currently have or why am I going to emulate a current solution for my clients?

The Problem and my proposed solution:

The problem I am trying to solve is the lack of a suitable science laboratory management system in Pate's Grammar School Science Department. The department needs an organized way to manage bookings of labs and experiments and keep track of all equipment that is currently stored in the labs. The system will allow the science department's technicians to lend equipment to teachers and it will create an online register of all equipment that is lent. The equipment will be lent based on requests made and based on the practicals that the teachers wish to conduct in their lessons. This system will allow the staff to book labs and equipment in advance of the lesson.

Currently this is basically a 'word of mouth process' as described by my stakeholder and requires some serious attention given the problems faced. These problems include loss of equipment, clashes in booking labs and having to look out for and find equipment during lessons. The 'booking' process does not take place until almost right before the lesson and is based on handwritten slips or emails.

A solution is crucial to this problem as it means that students and teachers do not need to go out searching for lab equipment. This eliminates error from word of mouth and regular emails – which is the current way in which the task is performed. This system will enable staff to send an automatic email to the technician as soon as the teacher selects the practical and the number of students.

Overview of the solution:

My solution will consist mainly of 2 databases. One of them will be a main school inventory and another will be a database of all the practical that the school science department conducts. These will then be accessed using a web app. The web app will allow teachers to login using their school credentials and they will then be able to select the practical that they will perform in their next lesson and the number of students that will be doing the practical.

Doing so, accesses the database with the practical list and emails a request of all the needed equipment to the technician. This will schedule a temporary reduction in the equipment in the inventory database. This means that any other teachers will not be able to book that equipment for that particular time.

I am aiming to also make a simple android app – which will use either an SQL database or Google Firestore for database storage and management – for this system which will allow the staff to pre-book in advance from their homes at their convenience. Firestore is an online, google based storage platform which provides real-time cloud-based storage.

****** I will add a screenshot of an example current email that my stakeholder might have sent before at some point******

Currently, ordering and setting up labs requires the teacher to email or speak to the technician to request for equipment. This method has a lot of room for error – the technician remembering what was ordered, the teacher calculating the total number of equipment needed based on number of students, and the fact that there is nothing to indicate if the equipment is in use or not. The web-based solution will be on the school servers allowing the teachers to book for labs and equipment remotely and in advance. Using databases to store equipment mean that staff will be able to report permanent loss/ breakages, add new stock, store information about every practical (eliminating the need for checking and remembering), and place requests at their convenience.

(b) Explain why the problem is amenable to a computational approach.

A detailed calendar-based storage which is updated automatically: Computer based solution will allow me to create an importable calendar that connects to the Science Department's Calendar making the selection for rooms and number of students even easier.

Automation/ Error Reduction: Automatic emails being sent to the technician make the whole process less time consuming and easier – once all practical have been added to the database. Because normally, this process would be carried out by a word-of-mouth system where teachers would type out their list manually and then send it off to the technician. Using a computer, the teachers can make sure that the data added initially is correct and then all they need to do is select the practical and send request to technician. This means that the room for error is very low.

Storage: The high storage capacity of the computer will allow staff to store multiple practical for each year group and book for them in advance. This storage power will also allow users to access booking records and histories to track lost equipment. Storage on the computer databases will be based on the school's virtual server. This helps teachers in using the

application and placing requests remotely and well in advance. Apart from this, the fact that data is stored on a virtual server means that it will all be automatically backed-up regularly, hence, enhancing safety of the data and protecting it from accidental damage.

Visualisation: Visual power and processing ability will enable me to add an option to visualise the databases. This will help staff keep track of and watch the count of all equipment.

Processing Power: High processing power means that the execution of requests will be much quicker, and scheduling will be implementable to ensure that there are no clashes when booking rooms and equipment. This will avoid the problems faced currently.

Multimedia: Given one of my stakeholder's request about adding images to each item in the database, this will only be possible because of a computer's multimedia capabilities. The fact that using SQL makes data retrieval and storage easier means that all processes will be much faster and efficient. Data on a server will allow staff to use the system remotely.

Development: Using libraries in python such as SQLite, database management becomes a lot easier. This means that my code is modular and there will be less repeated code. Code is concise due to using library methods. Front-end development will be done using HTML/ CSS. OOP in python, though difficult to manage, does add readability to the program, this means that there can be a class known as practical and the attributes can be the equipment needed, each different/ new practical is therefore, just an instance of that class/ an object. Doing so will make the application easier to update and modify the application.

(II) STAKEHOLDERS

(a) Identify and describe those who will have an interest in the solution explaining how the solution is appropriate to their needs (this may be named individuals, groups or persona that describes the target end user).

My application is for the Pate's Grammar School Science Department as requested by Mr. Geoff Worth who is the Head of Science and the Data Manager at the school and will be one of my stakeholders. The application will essentially be a system for managing the school science laboratory equipment. The school science department conducts practical demonstrations nearly every lesson and required practical nearly once a fortnight for EVERY class. This means that there is quite a big chance of the equipment being misplaced and clashes occurring when different classes and teachers are planning to use the same equipment.

Another one of my stakeholders is Mr. Naveen Begwani who is the head of finance for the Europe, Middle East, and Africa region for Kohler Co. He is experienced at using management softwares like Tally and will be giving me feedback for the various stages of the application development and will carry out some of the user testing.

Currently, Pate's uses word of mouth and simple emails to the technician for requesting setting up labs. This method is clearly vulnerable to inaccuracies. My system will help the school maintain an inventory of all the equipment available to them and will allow them schedule practical lessons and book equipment in advance. This will enable the staff to keep track of all available things and will allow them to avoid clashes in the lab sessions. An importable calendar function will make it very easy for staff to select the lesson time, room, and class (which will do the activity, and hence the number of students).

INITIAL QUESTIONNAIRE FOR MAIN STAKEHOLDER:

1. What is the problem in the way in which the task is currently performed?
2. Any particular issues or tasks that you would like the solution to solve?
3. What should the application look like visually?
4. Do you need a way to visualize what the practicals and equipment look like when stores in the database?
5. Are there any visual requirements, color scheme, and/or stylistic elements that the application must meet?
6. Does the application need to be compatible with the school servers?
7. Would you like the application to quantify equipment under practical names based on the year group or can they just be stored as separate practicals? (Eg: Year 8<Hooke's Law<Equipment or Hooke's Law<Equipment)

Initial Conversation with stakeholder:

19 August 2020

19/08 12:39
Hello Mr Worth,

Hope you are doing well.

I am pleased to inform you that Mr Read has approved my proposal of making the school science lab system.

I would request you to kindly give me details of the functions or features that you would like this web app to be able perform. I will soon share with you a prototype of the homepage of the application.

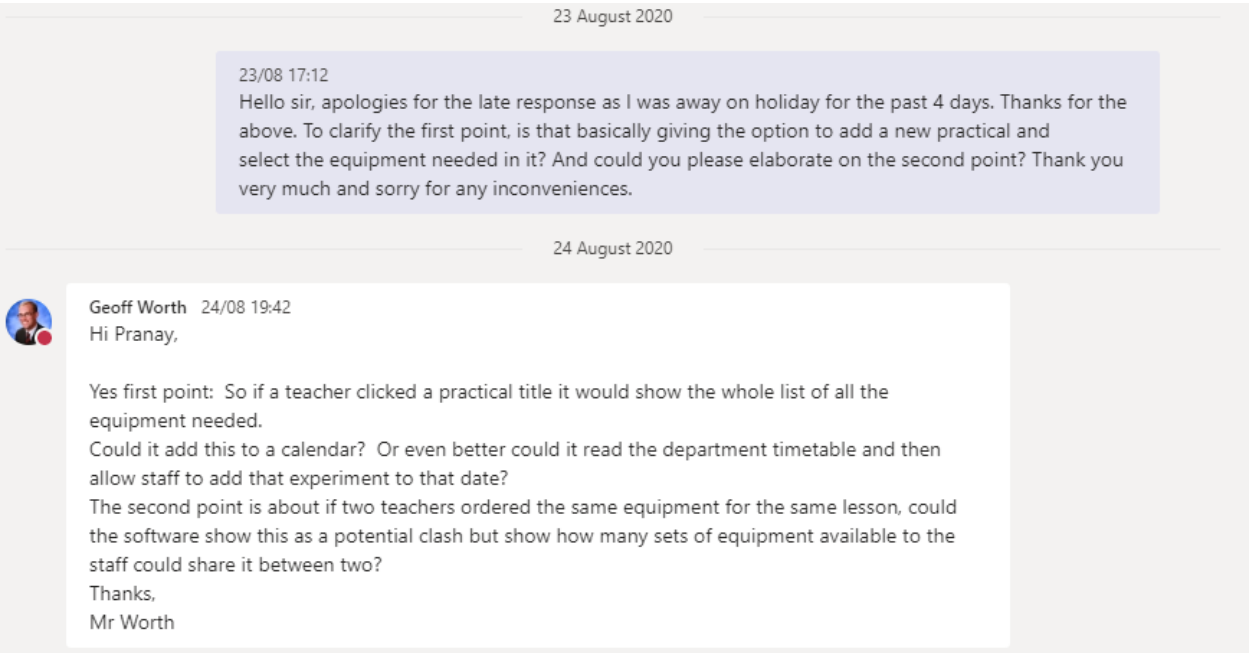
Looking forward to hearing from you.
Thanks,
Pranay Begwani

Geoff Worth 19/08 20:53
Hi Pranay

- To be able to create lists of equipment under a title
- To quantify how many sets of everything we have and so how splitting it between different groups would work
- To be able to order an experiment (eg Hookes law) and all the correct equip for that exp to come in a list.

How is that for a start?

Following this, I needed some more clarifications and some further details about the stakeholder's requirements.



23 August 2020

23/08 17:12
Hello sir, apologies for the late response as I was away on holiday for the past 4 days. Thanks for the above. To clarify the first point, is that basically giving the option to add a new practical and select the equipment needed in it? And could you please elaborate on the second point? Thank you very much and sorry for any inconveniences.

24 August 2020

Geoff Worth 24/08 19:42
Hi Pranay,

Yes first point: So if a teacher clicked a practical title it would show the whole list of all the equipment needed.
Could it add this to a calendar? Or even better could it read the department timetable and then allow staff to add that experiment to that date?
The second point is about if two teachers ordered the same equipment for the same lesson, could the software show this as a potential clash but show how many sets of equipment available to the staff could share it between two?
Thanks,
Mr Worth

RESPONSES AND ANALYSIS OF THE STAKEHOLDER'S INITIAL QUESTIONNAIRE:

Me: What is the problem in the way in which the task is currently performed?
Mr. Worth: We don't have an up-to-date database / inventory of all our equipment so very much it works on a word of mouth type process.
Analysis: As previously stated, the system will simplify much of this process – there will be a database with all practicals already stored in it, there will be an inventory database. Computers will allow to implement scheduling such that there are no clashes between teachers ordering equipment or booking labs. Technicians will not need to maintain an excel/ paper register record for all the equipment. Loss/ addition of new equipment can be reported easily and changes will be made on their own in the master database.

Me: Any particular issues or tasks that you would like the solution to solve?
Mr. Worth: Having an easy to use piece of software that would enable any member of staff find a piece of equipment quickly.
Also, to enable the user to pick the name of a practical and for it to produce a list for the technician of all the equipment required for it.
If we could put in the number of sets of equipment that would help too –ie. 1 set as a teacher demo. 8 sets for a class practical.
Analysis: Okay, this makes sense. In order to implement **this, I will now add another text box** to the initial homepage of my application. The three textboxes will now be: 'Name of Practical', 'No. of Teacher Sets', and 'Number of Student Sets'. For the practical textbox, I will implement a drop down which will be sorted by year group and practicals in alphabetical order allowing staff to select the practical easily. The calendar functionality will make year and class selection easier.

Me: What should the application look like visually?
Mr. Worth: Really does not matter. But easy to use.
Analysis: I am considering using the school website's color scheme i.e. red, white, dark grey. To enhance the ease of us, I am implementing drop down menus as opposed to letting the user type the name of a practical.

Me: Do you need a way to visualize what a practical and its equipment look like when stored in the database?
Mr Worth: Yes, that would be hugely helpful. Could we insert photographs?
Analysis: Inserting full photographs maybe a challenge given the time constraints and also because databases are not actually meat to store images in the table. I will try to at least incorporate 'links' to photographs in the table so that each photo can be accessed via its link, making the database and program as whole, more efficient. Although this wasn't initially a part of my plan, I will implement this given that it is a request by the user. Direct addition of images in a database slow down databases and make the application inefficient, therefore, I will add references to images only.

Me: Are there any visual requirements, color scheme, and/or stylistic elements that the application must meet?

Mr Worth: No

Me: Does the application need to be compatible with the school servers?
Mr. Worth: Yes – could it be web based?
Analysis: Yes, the application will be web based and it will run on the school servers.

Me: Would you like the application to give the user a chance to quantify equipment under practical names based on the year group or can they just be stored as separate practical? (Eg: Year 8<Hooke's Law<Equipment or Hooke's Law<Equipment)
Mr. Worth: Yes, the more flexibility the better.
Analysis: The application will sort the practicals under year groups. As mentioned by the stakeholder, this will improve the flexibility of the application and make bookings easier if not faster.

Me: Would you like to be able to edit source code at your convenience?
Mr. Worth: No this would be more than we need.
Analysis: Although the stakeholder doesn't need the source code, the source code will be available to edit because the application will be live on the school servers.

Me: One more thing I wanted to clarify; you do need an importable calendar function on the app's homepage, right?
Mr. Worth: What I really want is to be able to import the physics department calendar for the year with all lessons on it. Then allow staff to book the equipment linked to that lesson exactly. I.e. so the resources are tagged to that lesson. Then for the complete overview of all the teacher and lessons and resources be outputted to a calendar so that all lessons are viewable by anyone.
Analysis: Initially, I thought that the calendar functionality was to be restricted to the application being able to take information from the calendar (like students number, time of lesson, room number). This message from the stakeholder means that I will try to expand the use of the calendar in my application to be able to receive data 'tags' from the application – as requested by the stakeholder.

POST QUESTIONNAIRE ANALYSIS SUMMARY:

Based on the response from the user I have decided on adding the following to my previous ideas:

1. More functionality to the calendar feature:
    a. Add tags such that all staff can view any bookings made.
    b. Selecting a lesson and being able to send a request – without having to enter room number and no. of students manually.
2. Better and more sophisticated sort and filter options to allow ease of booking.
3. Adding a column for image references of every item in the inventory
4. Separating the teacher bookings and students bookings for sending request to technician.

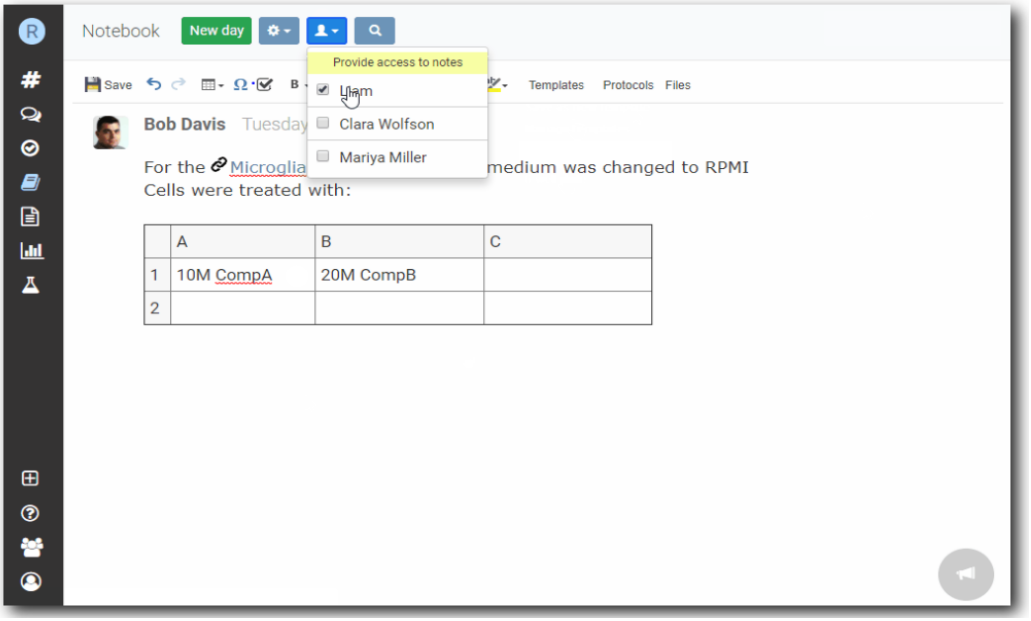
(III) RESEARCH THE PROBLEM

(a) Research the problem and solutions to similar problems to identify and justify suitable approaches to a solution.

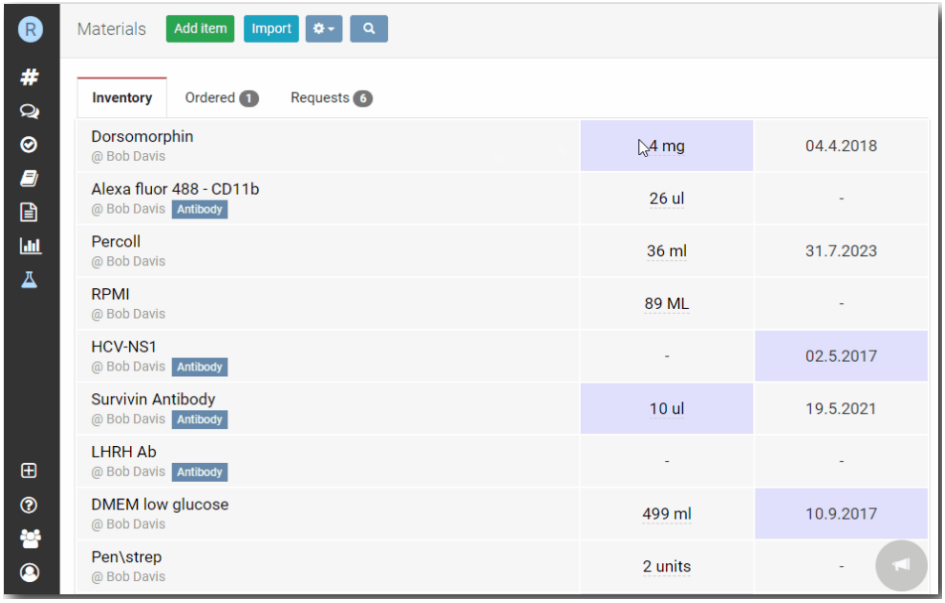
(b) Describe the essential features of a computational solution explaining these choices.

The major problem with currently existing solutions tends to be the cost! This problem is not that major and so spending thousands of pounds a year is not worth it. It tends to make much more sense to just replace lost/ broken equipment because it is much cheaper than spending thousands of pounds on the management system. Some of the existing solutions that I have researched into are below. I have also pasted some screenshots of the applications using their demo versions. The description of each of these images is below the image.

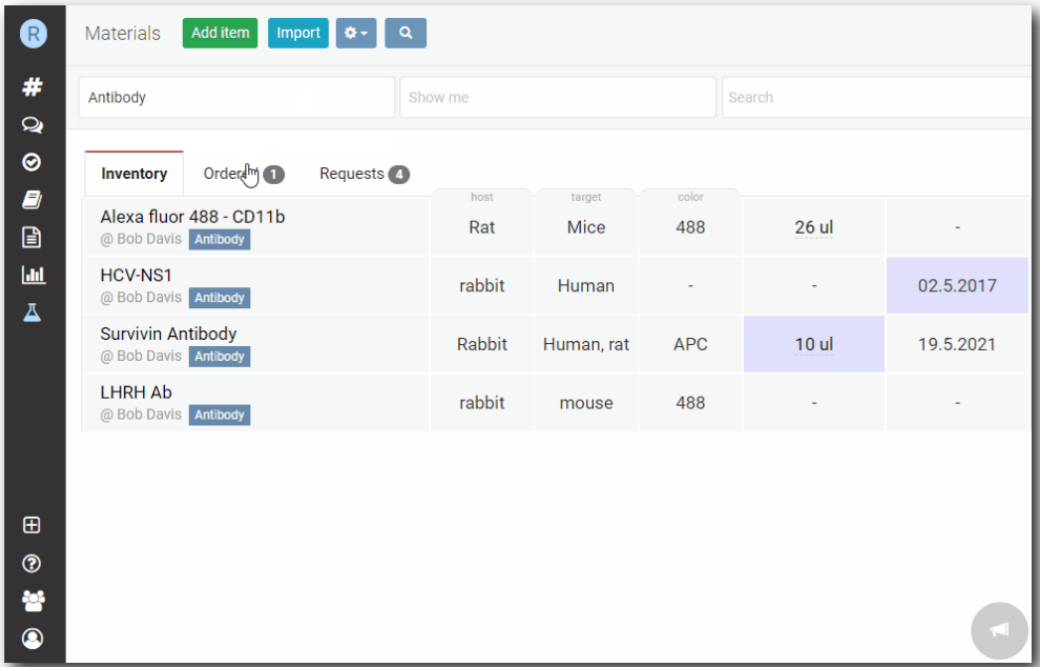
1) LABSPACE.IO



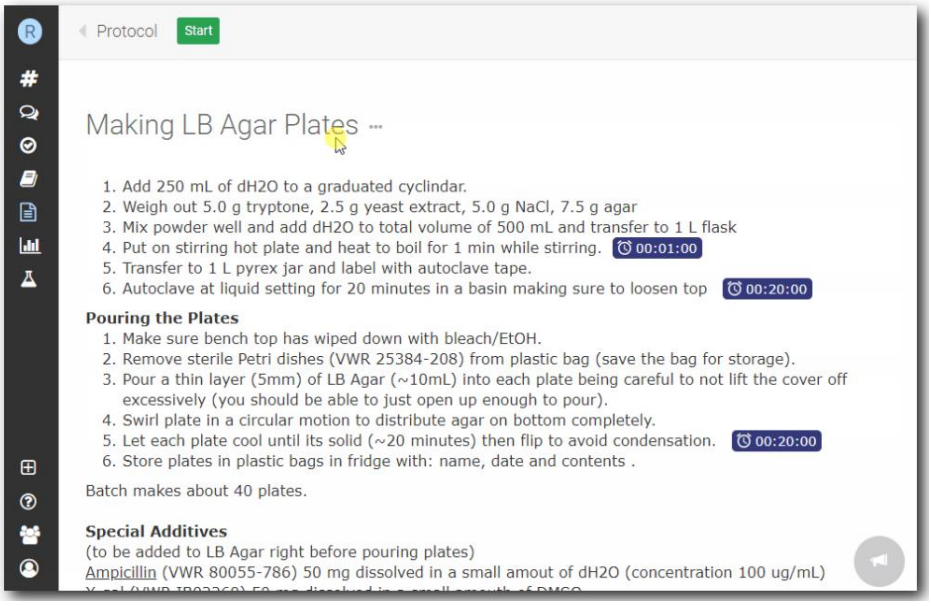
This solution has a notebook system which doubles as chat system between teachers and an area to store notes and 'procedures' for practicals. This notebook system is very sophisticated allowing user to add things like tables, change font, attach documents, etc. I really like this feature as provides a lot of flexibility. This notebook doubles as a Google Docs equivalent where staff can collaborate an work on a single document together. This allows the teachers to prepare standard methods for practicals and is very handy for future references. Although this will be helpful, time constraints mean that I will not be able to implement this.



This system shows the full available inventory. Clicking on an item leads to the page with details of availability, history, and practicals in which that equipment is needed. This page has an 'Add item' button which add the selected item(s) to the 'basket' from where you can select the quantity and place the request. I really like this option to be able to view the full inventory and then being able to view details of each item just by clicking on it. Initially, I had planned to use the inventory database of my application to just view what the school has and to add new stock or report breakages. Looking at this, I might add the optionality to click and select an item and view information such as which experiments need that item, how many are available, how many are in use and their booking details. One feature my application will contain is an image reference to the particular equipment showing the location of the equipment.

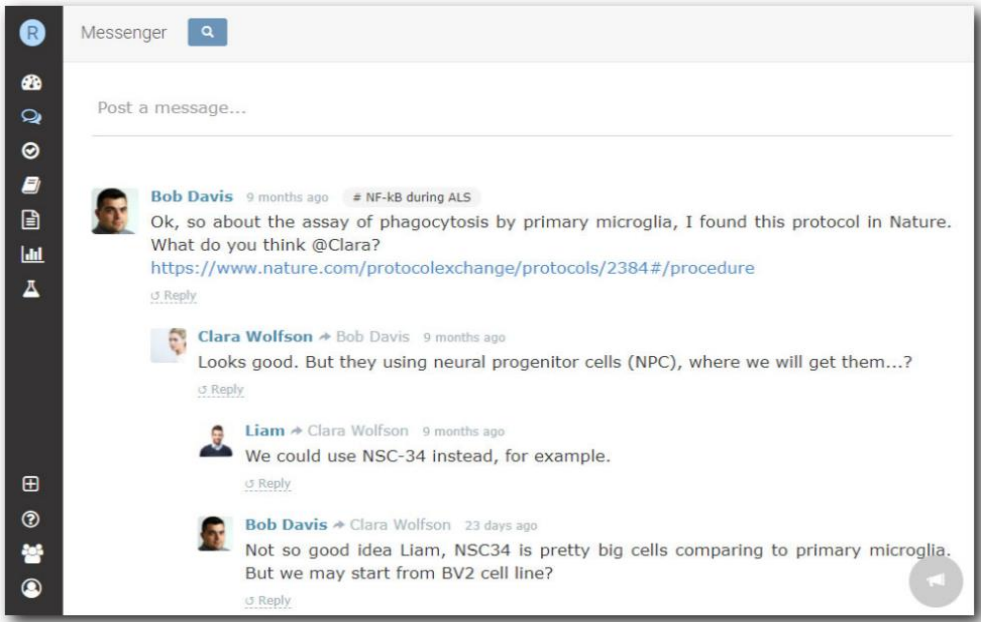


This is the page that opens up when the user searches for a particular item by clicking on the search button on the previous page. This page will give the option to making changes to the quantity and details of a particular item. The user can add the item to their 'basket' from where they can place a request. Item can be added from top button. From here user can send request and receive quote. Although I like this feature and it adds a layer of convenience, I am not sure how easy it will be to implement a searching option in the application.



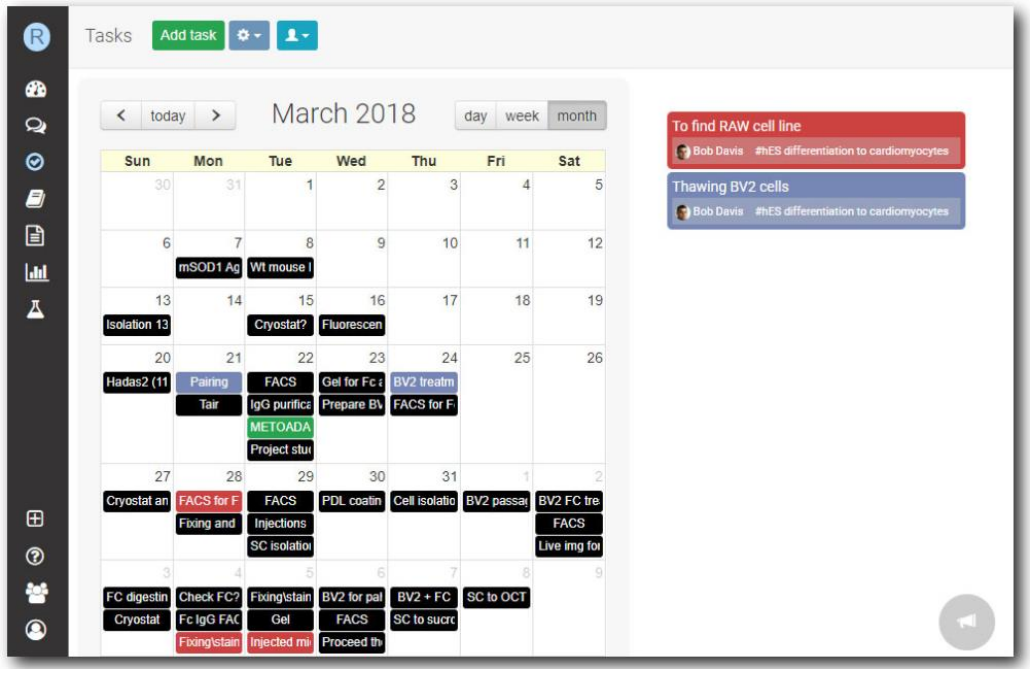
Gives user the choice to include method of performing practicals and set 'protocols.' There is also an option to record notes and make amendments to the notes made. These notes can then

be accessed by anyone in the organisation. Although an attractive feature that I like, time constraints make it difficult to implement. This is an extension to the notebook feature described above.



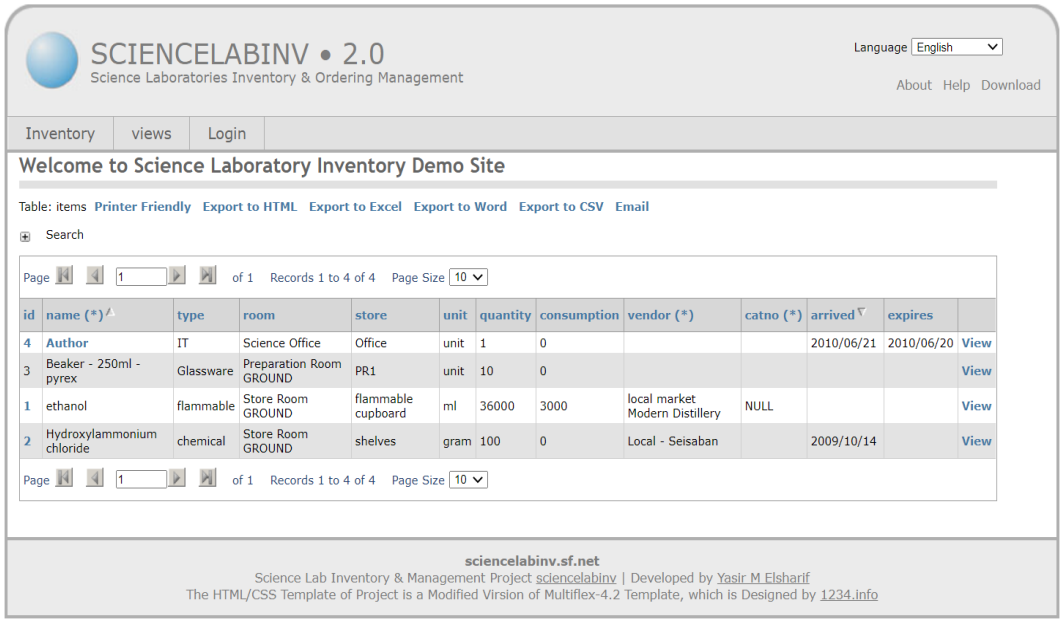
There is also a chat function in this application which allows the user to coordinate with the different members of the departments. This is a very good feature. However, it is something not needed in the school. All conversations take place on Microsoft Teams and Outlook which is much more sophisticated for having conversations. Also, implementation of a chat system

requires knowledge of networking and port systems and communication infrastructure. This is not feasible to make in the given time. Although, I would like to work towards this.

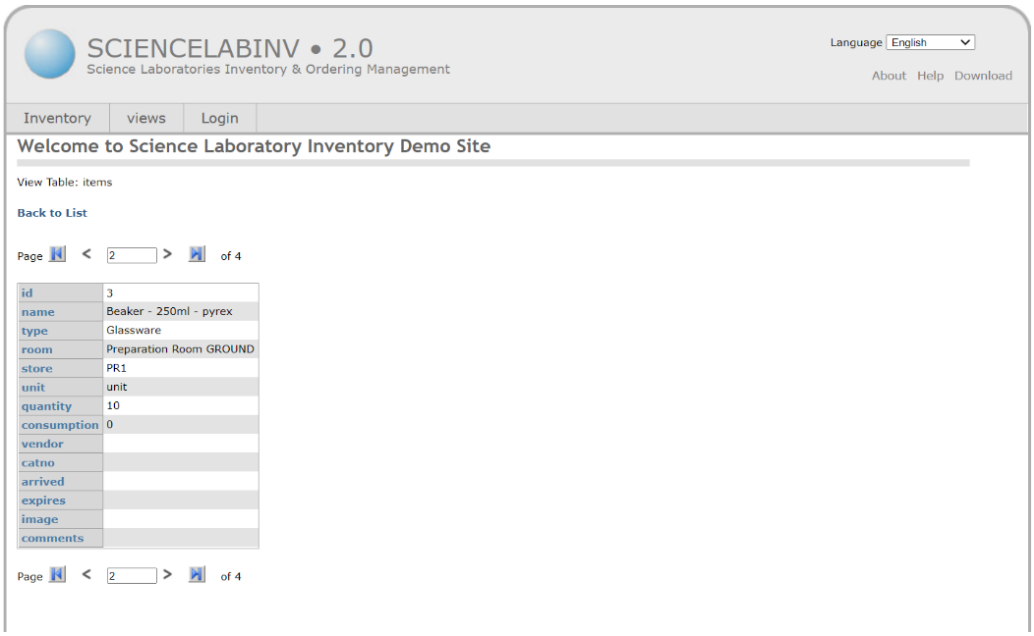


This website has a calendar function which shows the rooms that have been booked and the experiment which will be performed at any given time. I like this feature and based on the stakeholder specification I will be making a similar feature in my app. I really like the functionality of viewing 'tags' – the horizontal bars on each day in the calendar. This is a good place to consolidate the bookings mage. Teachers and members of staff will be able to select a tag and view the associated booking details: room number, students, experiment name, and equipment. This calendar in my app will be connected to the school's Physics Department's calendar.

2) HTTP://SCIENCELABINV.SOURCEFORGE.NET/DEMO/ITEMSLIST.PHP

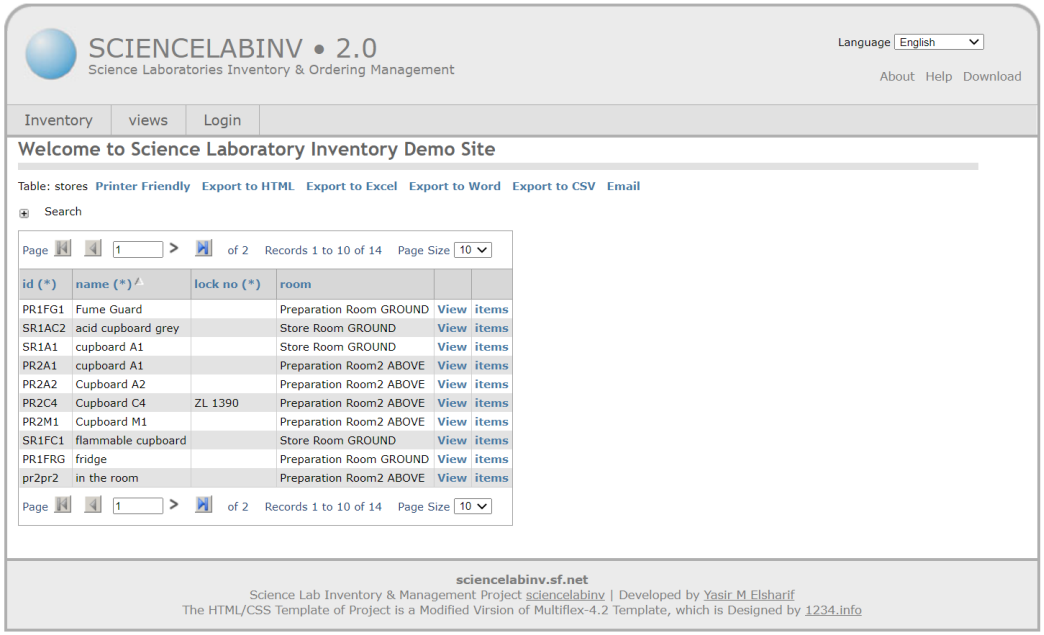


This is the homepage of the application, shows equipment, dept, room, location, units, quantity, etc., of the equipment available. A similar database table will be a part of my application but the access of the same will be restricted only to certain individuals. The equivalent of this page in my application will enable the user to track loss/ broken items, add new stock, track quantity of a certain equipment, have the image link, storage area, etc.



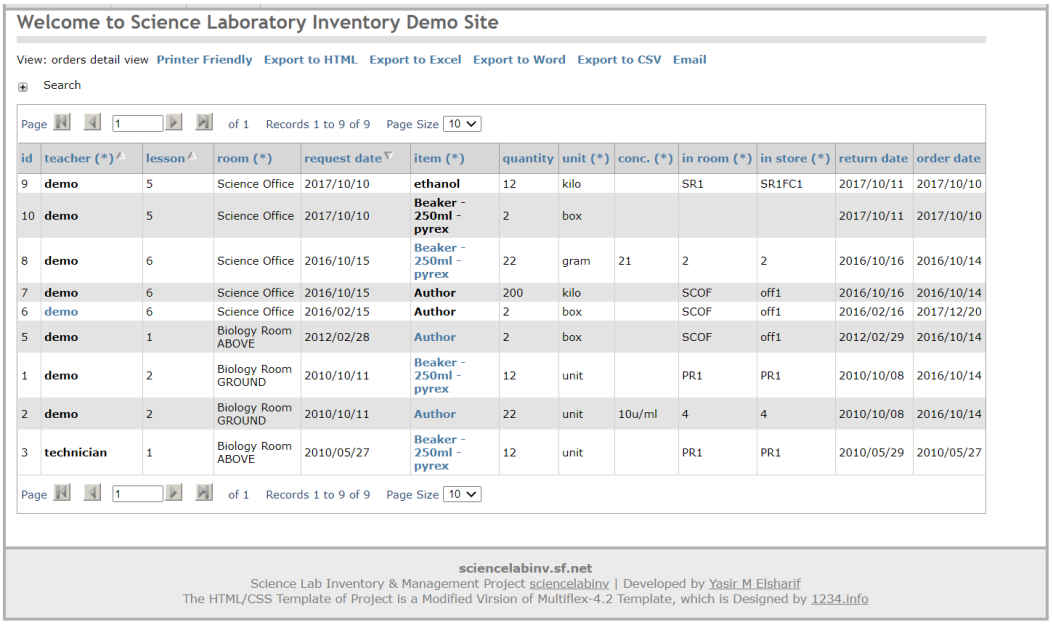
This screen appears when one of the equipment is clicked on. This displays further details of about the piece of equipment selected. My application will contain a similar feature, but except

I will not create a separate page for each item. The table in the database, described above, will contain all this information. Although I like this feature, it is not particularly helpful and won't satisfy any needs of the stakeholder.



This page is one of the tabs under the 'Inventory' button. Displays all the locations where the equipment is stored and the equipment stored in that area. This feature will be implemented in the main table. All the items will have next to them their locations and an image reference. This feature may seem helpful, but this application seems to be repeating quite a few things and that only adds a layer of confusion in my opinion.

Another tab under the ‘Inventory’ button. Displays all the lab locations. Labs is not a major focus of my application. Given the time constraints and stakeholder’s initial requirements, it does not make sense to implement lab locations. What my application will do is, is that it will give an option to book a room or lab when the teacher is booking for a particular lesson.



Welcome to Science Laboratory Inventory Demo Site

View: orders detail view Printer Friendly Export to HTML Export to Excel Export to Word Export to CSV Email

Search

Page 1 of 1 Records 1 to 9 of 9 Page Size 10

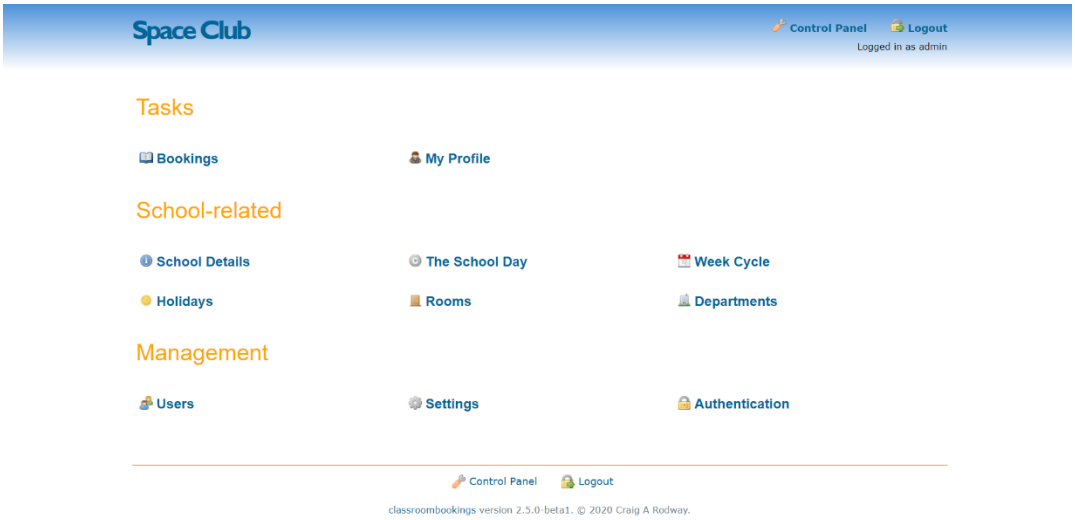
id	teacher (*)	lesson	room (*)	request date	item (*)	quantity	unit (*)	conc. (*)	in room (*)	in store (*)	return date	order date
9	demo	5	Science Office	2017/10/10	ethanol	12	kilo		SR1	SR1FC1	2017/10/11	2017/10/10
10	demo	5	Science Office	2017/10/10	Beaker - 250ml - pyrex	2	box				2017/10/11	2017/10/10
8	demo	6	Science Office	2016/10/15	Beaker - 250ml - pyrex	22	gram	21	2	2	2016/10/16	2016/10/14
7	demo	6	Science Office	2016/10/15	Author	200	kilo		SCOF	off1	2016/10/16	2016/10/14
6	demo	6	Science Office	2016/02/15	Author	2	box		SCOF	off1	2016/02/16	2017/12/20
5	demo	1	Biology Room ABOVE	2012/02/28	Author	2	box		SCOF	off1	2012/02/29	2016/10/14
1	demo	2	Biology Room GROUND	2010/10/11	Beaker - 250ml - pyrex	12	unit		PR1	PR1	2010/10/08	2016/10/14
2	demo	2	Biology Room GROUND	2010/10/11	Author	22	unit	10u/ml	4	4	2010/10/08	2016/10/14
3	technician	1	Biology Room ABOVE	2010/05/27	Beaker - 250ml - pyrex	12	unit		PR1	PR1	2010/05/29	2010/05/27

Page 1 of 1 Records 1 to 9 of 9 Page Size 10

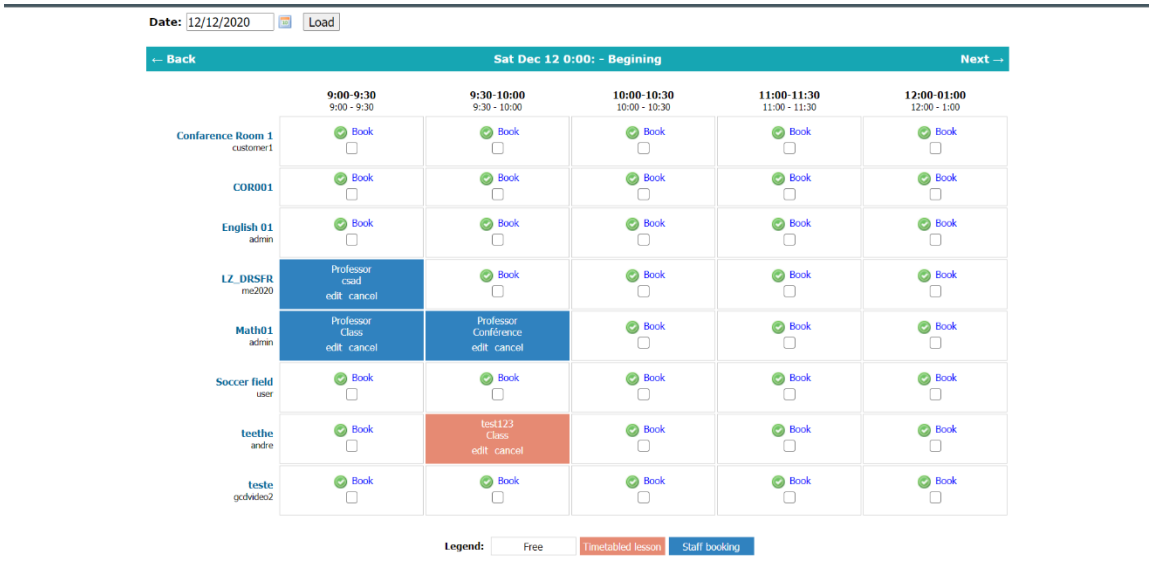
sciencelabinv.sf.net
Science Lab Inventory & Management Project sciencelabinv | Developed by Yasir M Elsharif
The HTML/CSS Template of Project is a Modified Virsion of Multiflex-4.2 Template, which is Designed by 1234.info

This page shows all the booked room and equipment and the booking history. This is a feature I had not thought about implementing initially, but thinking about its potential usefulness such as, tracking of lost equipment, etc., I am considering creating a new button in the app’s homepage named ‘History’ and this will contain every booking history.

3) HTTPS://DEMO.CLASSROOMBOOKINGS.COM/

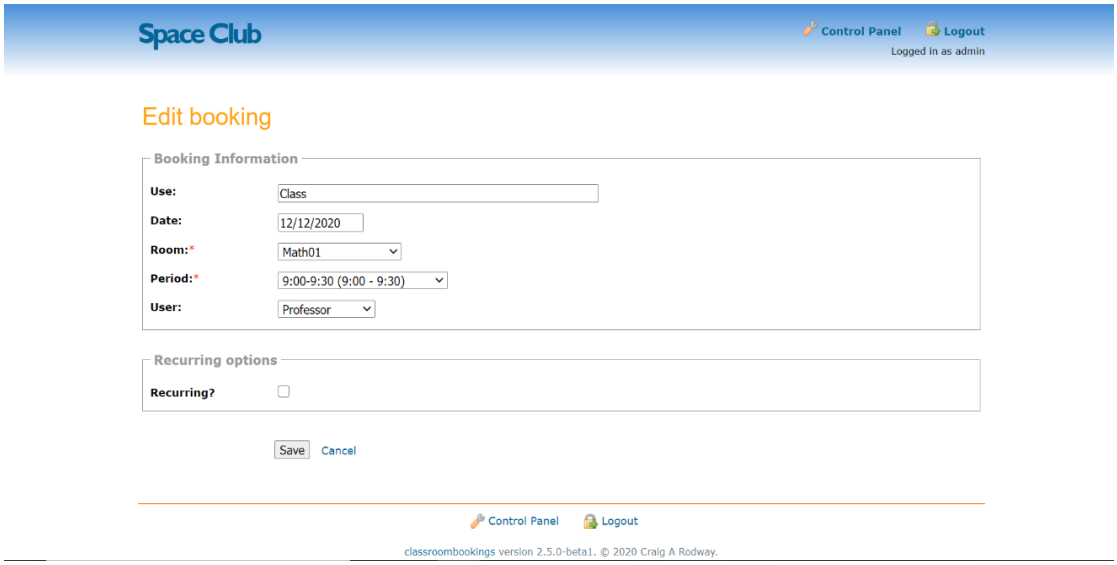


This is the homepage of the application. I like the simplistic layout of the application as it clearly shows what each link/button will lead to. Although the UI is not very pleasing, the application appears easy to use and that is the primary and most important condition to be met.

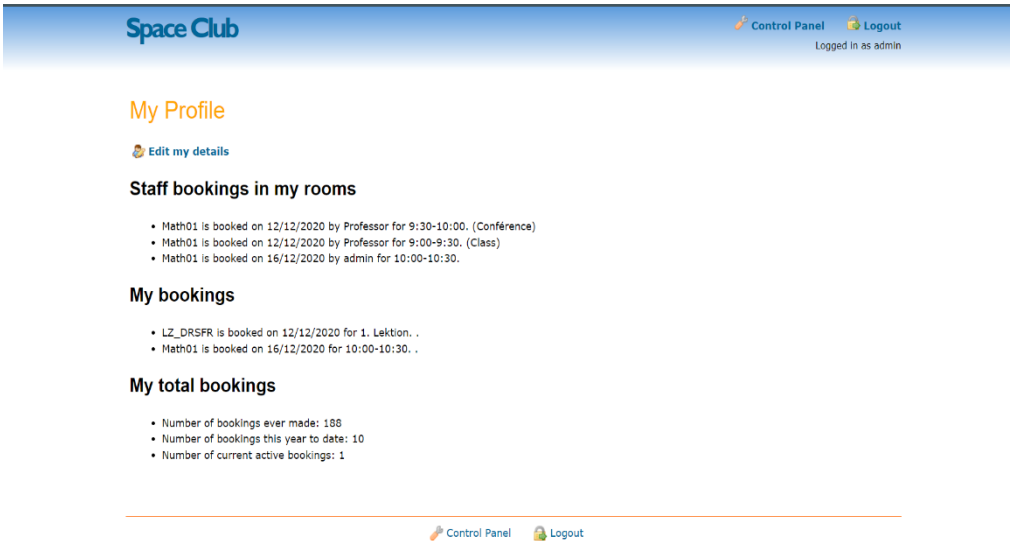


This page is the page that appears when the Bookings option on the homepage is clicked. It shows a calendar layout which has bookings for every 30min timeslot. This is again a very simple layout and each block clearly shows the name/ event booked and gives the option to edit the booking or cancel it. I like the simplicity of this feature and given that a calendar is something that I am trying to implement, it gives me a good idea for a potential implementation of the calendar. The calendar in my application will have a ‘tag’ feature, this

will be very similar to the ‘edit’ button on this calendar’s block. Each tag will show details of a booking that has been made. Currently, I am thinking of only allowing the user to view booking details using the calendar, but I am considering to expand this functionality to allow the user to edit a booking by clicking the tag.



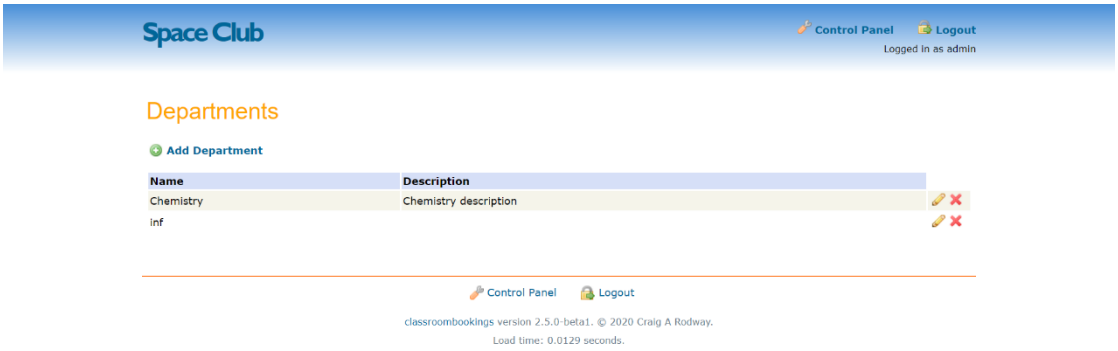
The above shows the page that appears when the edit button is clicked on the calendar block of a booking. As stated above, I am considering implementing a similar feature. In case of my application, the editable details via the calendar will be the practical name (the details of a practical will be edited separately) and the number of students.



This is the page that opens when I click the ‘Profile’ button on the homepage of the website. This page shows the booking history of the user I like this feature, and based on analysis from

the previous solution, I will be implementing this in my solution. My solution will have a slightly different approach though: it will show ALL bookings that have been made, no matter who the user. Therefore, the booking history will not be user specific.

This page gives a list of all the rooms available. My website will implement this feature indirectly, the rooms have already been allotted to each lesson and my website will simply reinforce the booking already made, rather than giving an option to book a new room.



This is the Departments section of the homepage. Simply shows all the departments using the app. This would have been helpful to filter the booking history based on the department – Physics, Chemistry, or Biology. My application is limited in scope to only the Physics dept., however, I will consider this for a potential upgrade to the application – if I have time towards the end.

(c) Explain the limitations of the proposed solution.

POST RESEARCH ANALYSIS:

After researching, I am thinking of expanding the use of the calendar function to include booked labs clicking on which will give further details about that time. Not only is this a request from my stakeholder but the sophisticated use of the calendar in the first researched solution is a good representation of what I am striving for.

Looking at both the solutions above, I am also thinking of including a booking history which shows the information of booked equipment. The potential uses of this, such as tracking down lost equipment and looking a booking history for rooms, are clearly very beneficial in the future.

First solution is very sophisticated, and I probably do not have enough time to include chat system, notebooks, and as sophisticated a UI. One thing, which according to my research, both systems did not have, is a very helpful system of ordering equipment. Both systems being demo versions, maybe less sophisticated as I tested them. Having an option to just select the practical and automatically order all equipment needed is much easier than individually selecting equipment and adding it one by one to the cart – I am planning to give the option to just select an experiment, add number of students and the system orders all equipment in one go, no need to individually add equipment to a basket and then order it. This means that the teachers/ technicians only need to add the equipment needed under a practical name once and every time an experiment is booked, the total equipment needed is calculated automatically and sent to the technician.

After careful look at the second application, I feel that although it is a very detail-oriented approach to solving a problem and managing an inventory, it is very repetitive. Details such as the name of equipment, the locations, bookings made, etc., are repeated and this, in my opinion, adds a layer of complexity to the usability of the application.

The third solution is a very simple approach to a similar problem. Although it is not used to book science practical lessons, it is used to book meeting and rooms for lessons. The app has a very simple UI which I like a lot due to ease of navigation as compared to solution 2. This app's color scheme is not very appealing to me but the fact that UI elements are placed conveniently and can be found easily, it makes up for that! The amount of customizability offered is also good i.e. a person can make a booking, they can amend it, delete it, etc. This is very similar to my application's approach – the staff can add new practical, amend their details, delete them, select a practical and make a booking!

ABSTRACTION AND VISUALISATION

Certain elements of the application will need to be abstracted to avoid slowing down the development process and make the application unnecessarily complicated. Based on the user's requirements, these elements are not very essential and can hence be reduced in complexity or ignored entirely.

1. A chat system built-in to the application – I need not add a chat system to application because of the already prevailing school network on Microsoft Outlook and Microsoft Teams, therefore, the chat system is basically irrelevant.
2. A notebooks section – I do not need to add a notebook system to the application, this system is normally helpful for students but because they do not have access to tablets and/or laptops in class this system is pretty useless. As things stand, each student gets a paper hand-out for each experiment done making this system even more redundant.
3. The process taking place in the background, such as, scheduling for the lab booking, making amendment to the quantity of a particular object.
4. Details of physical abstraction like the storage/ computer memory used by storing one practical, size of memory, and implementation of data structures. Parts of the algorithm will need to be given to the organization to give them more room to upgrade in the future.
5. Logical details such as relationship between different tables/ entities need not be visible to the user although they will be available for the user to view.
6. Access levels will be established so that some people in the system have no access to the database at the background, other have partial access and the remaining have full access allowing them to change anything they want to.

THINKING AHEAD

Initially, when setting up the application, the inputs will be each practical and the equipment needed to perform that practical. The user will need to click the 'Add Practical' button and this will prompt them to enter a name, select the equipment needed, and the quantity of that equipment. This will then add that practical to the practical database. Another initial input that the technicians will have to enter is the quantity/ amount/ number of each equipment that the lab already has – this will then be copied into 2 tables: one for temporary changes when lab is booked; second for permanent changes like new stock/ breakages.

The outputs received by the user when running this application are limited to things like error messages and prompts. The other major output is the message that is sent off to the technician once the user places their request.

Other inputs include:

1. Option to edit an already added practical's details
    a. Change_equipment
    b. Equipment quantities
    c. Add new equipment
    d. Delete equipment
2. Equipment needed
3. Quantity of needed equipment
4. Adding/ deleting experiments
5. Buttons of the navbar
6. Details of a new equipment being added to the inventory
7. Adding/ Decreasing quantity of equipment

THINKING PROCEDURALLY

The application needs various tabs:

1. A home page where the user can order equipment from: this is where user selects name of the practical from a dropdown menu and gets the option to select number of students.
2. The home page will have buttons namely: Add Practical, Edit Practical, Delete Practical, Add Items to inventory, Report loss/ breakages. **Each of which will have a separate module/ screen.**
3. I will try to include an importable Microsoft Calendar so that teachers can book labs based on their schedules and classes, this will also be like a separate module incorporated with another module.
4. The main procedures needed across the application are as follows:
    a. Add_new_practical(): Add a new practical to the database and enter the equipment needed to book it.
    b. Edit_Practical(): Edit an existing practical – changing equipment/ quantity/ both.
    c. Delete_Practical(): Delete an existing practical so that it can't be booked.
    d. Add_New_Equipment(): Add new stock/ new equipment/ both to the inventory; these can be used in practicals.
    e. Report_Loss(): Permanently remove equipment from the inventory – due to loss or breakage.
    f. Calculate_Total_Equipment(): Calculate the total equipment needed to perform a practical – when booking is being made. Essentially multiply equipment needed in one set of practical by the number of students. And compare if needed equipment is available or not.
    g. Create_Message_Request(): Compiles the calculated total equipment into a message which will be sent to the technician.

h. Email_Request(): Actually emails the created message to the technician
i. Create_calendar_tag(): Summarizes a booking made and places it in a tag on the calendar of the website.

THINKING LOGICALLY

1. Critical conditional statements:
   a. If statement to check whether the HTML template is POST or GET method.
   b. If statement to check if a form being submitted is valid or not.
   c. If statement to check if a room is booked already for a given date and lesson time.
   d. It statement to check if a teacher is busy for a given lesson time and date.
   e. It statement to check if a field in a form is null or not.
   f. If statement to check if a practical exists already in the database
   g. If statements to check that the types of data entered in a field are correct
2. Critical iterations:
   a. For loop to retrieve all equipment needed in a practical (when booking) in a list
   b. For loop to store quantities of all retrieved equipment in a list
   c. For loop to multiply all quantities by number of students and store that in a list
   d. For loop to add equipment and quantities in a list for the message that needs to go to the technician.
   e. Loop to store every individual equipment and its quantity when adding them to a practical.
   f. Loop to iterate through bookings made and display them.
   g. Loop to iterate through inventory table and display all equipment in a table.

THINKING CONCURRENTLY

There are some minor elements of concurrency involved in my application:

1. When a booking is being made, several tests are simultaneously undertaken to calculate total equipment, create a message, look for clashes, email the message, and update the booking history.
2. Next, when following the CRUD operations on data, all the database tables need to be consistent and will be updated simultaneously. So that the database is consistent.

(IV) SPECIFIY THE PROPOSED SOLUTION

(a) Specify and justify the solution requirements including hardware and software configuration (if appropriate).

HARDWARE AND SOFTWARE REQUIREMENTS:

1. Given that the application is light weight, a reasonably modern CPU should suffice.
2. For jobs like visualization, the user will need a monitor.
3. For data entry and placing requests a mouse and keyboard will be needed.
4. Given that this is an application for the school, I will need access to the school server to launch this application so that it can be accessed by staff members. This will also mean that the application can only be run on the school server.
5. Given that this is a web app, it will need a browser to run. This must preferably be Chrome because that is what the application will be tested on.
6. The app will not be made keeping in mind touch screen devices like phones and tablets and therefore, the application may not be compatible with such devices.
7. Flask library of python will need to be installed – this library will perform the bulk of my application's information and data processing.
8. A python interpreter may also be needed on the school server to be able to do the processing.
9. Windows OS or even MacOS – these are operating systems both supported by the languages and browsers that will be needed to run the application.
10. Need a decent amount of storage on the servers so that the applications databases can be stored easily on the server for data retrieval and storage.

OBJECTIVES:

Following this feedback, I have made some changes to the design objectives. The new objectives can be seen below. The altered parts have been highlighted.

1. Aesthetic objectives
    a. The main window size is 95% of the user's screen, therefore, it changes based on the display. The user can't resize the screen though.
    b. The colour scheme is red, grey, white – this is the colour scheme of the school website.
    c. There will be a school logo on the top right of the screen – clicking this takes to the school's main website.
    d. Text boxes
        i. They will be white with black outlines.
        ii. Password for login will be entered as dots.

iii. Practical will be selected from a drop-down menu.
iv. The text will be black.

e. The font used will be arial – it will be consistent throughout.
f. Buttons:
   i. White background.
   ii. Black border
   iii. Uniform padding
   iv. Text becomes red in colour and mouse turns to finger when hovered over.
g. Calendar:
   i. This will be covering 60% of the application's main window (which is 95% of screen).
   ii. This will be implemented via the 'Share' link on OUTLOOK.
   iii. The calendar objects will be blue – given that this is from outlook link, it will be difficult to get this to match the colour scheme.
   iv. Clicking an individual day will preview the day in the right of the calendar.
   v. Clicking an individual event will preview the details of that event.

2. Inputs
   a. Screens are navigated and actions are carried out by clicking on-screen buttons
   b. There are text entry fields on the following screens:
      i. Staff Log In
         1. Information to be entered – username, password – this will be connected to the school server.
      ii. Homepage
         1. Information to be entered – Name of experiment being done, number of student sets needed, and number of teacher sets needed.
         2. This page will have a send request to technician button.
         3. There will be other buttons as well – each of which will lead to another page in the application.
      iii. Add Practical
         1. Information to be entered – Name of the practical, the name of the equipment you want to add, quantity of the equipment (these last two will be from a drop down menu), and an add button which will one more field for more equipment.
      iv. Edit Practical
         1. Information to be entered – Name of the practical, and anything in that practical that needs to be updated.
      v. Delete practical
         1. Information to be entered – Year group and name of the practical both from a drop down menu.
      vi. Add to Inventory
         1. Information to be entered – Name of what is to be added – this is either selected from the drop down or a new field is created if you want to add a totally new thing – and the quantity of the item.
      vii. Loss/ Breakages to Inventory

1. Information to be entered – Select the name of the equipment and the number of broken ones.

c. Drop down menus are used to select practical names. These will be present on the Homepage, Edit Practical, Delete Practical, Add to Inventory, Loss/ Breakage page.
d. The calendar is also another major form of input, this will be located on the Homepage of the application.
    i. It will provide the following information:
        1. The date for booking the lab
        2. The class and therefore the number of students
        3. The room number to book
        4. All this information will be sent automatically for python to process and work with.

3. Processing
    a. The application will need to process inputs from the calendar. The input for this will just be selecting the class – the app will then need to retrieve the class size and room number from that.
    b. It will then multiply the number of students with the fixed number of equipment that will have been stored by the technician.
    c. The request of this will then be sent to the technician as an outlook email – this might need there to be an outlook account for the application. I will need school permissions for this.
    d. Some other processing will be adding or subtracting from inventory losses or breakages and for new stock, deleting old or adding new practicals.
    e. A basic scheduling algorithm will have to be implemented – probably first come first served – to avoid clashes in booked rooms and hence, equipment. This will mean that teachers will know if equipment or room is available or not.
    f. Need an algorithm such that all changes made to the practicals database are temporary and reverted to normal once technician acknowledges the return of equipment. The changes to main inventory, however, will be permanent.
4. Outputs
    a. The database table for the practical list and the inventory will be available to the user on request (an option on the app).
    b. The calendar implementation of the application will show ‘tags’ – these are essentially markers which indicate bookings that have been made.
    c. Booking History
        i. Shows all labs and equipment that have been booked over the past month (might change timeframe).
    d. A major and perhaps the most important output will be the message request that goes to the technician.
        i. Contains the teacher’s name (the person who made the booking)
        ii. The room that has been booked
        iii. The equipment needed and date it is needed for.
    e. Other outputs:
        i. Minor prompts for input

ii. Prompts for errors input
iii. Confirmations for any amendments made to any experiments – such as, changes to equipment needed, additions of new experiments, deletions of existing equipment – and changes to the inventory – losses or additions.

(b) Identify and justify measurable success criteria for the proposed solution.

LIMITATIONS OF THE SOLUTION:

1. Adding a chat and messaging system like some of the applications I had researched into. This will be very difficult to implement in the given timeframe. Setting up a chat system also requires knowledge of network protocols and that is not possible to thoroughly achieve in the given time.
2. Given that the application is running on school servers and that I am using python libraries like Django for the server-side processing, my stakeholder will need to install this library on their machines for some of the testing. Although relatively simple to install it is still a drawback of the system. Even the school IT technicians will need to enable Django to run on their server.
3. Given my limited time and expertise, implementing sophisticated user interface elements like those of one of my research examples will not be possible.
4. Implementation of a sophisticated scheduling algorithm to avoid clashes in database manipulation, for example during data entry when reporting loss or when adding new stock, multiple people might not be able to do this simultaneously.

SUCCESS CRITERIA

1.	Successfully login to the system	Successfully being able to login in the system using the school credentials.
2.	Physics Department/Any required calendar visible on the homepage	Successfully displaying a required calendar on the homepage of the application.
3.	Functional User Interface elements and simple layout of the application	Navbar of the application works properly, user can navigate between all pages of the application easily. The layout of all the UI elements like textboxes, dropdowns, buttons is simple and can be navigated easily.
4.	Ability successfully to add a practical experiment	Successfully adding a practical to the database. This will involve selecting a piece of equipment/creating a new equipment, selecting quantity of this equipment, confirming the addition, being able to visualise this change in the database, and receiving an output that confirms the amendment.
5.	Ability to amend an existing practical	This involves, successfully entering the page to make the change, select the name of the practical, make the needed change – this could be the name, equipment, or the quantity – save it, receive a confirmation, and view it in the database table.
6.	Delete an existing practical	This involves successfully selecting a practical from a drop-down menu, deleting it, receiving a confirmation and viewing it in the database.
7.	Add new stock	Select an equipment or type in a new equipment, add the quantity, and be able to view the change in the stock/inventory database.
8.	Report a loss or breakage	This involves successfully selecting a piece of equipment, entering quantity of the broken equipment, clicking the button which then reduces the quantity of the equipment from the school stock/inventory, and this change should be visible in the database that the viewer views.
9.	Viewing the details of a selected equipment in the inventory	This involves viewing quantity, availability and bookings of an item selected by the user from the inventory.
10.	Edit details of selecting an item from inventory	Successfully editing the details of a selected item in the inventory. This includes the name, qty, location, and image reference.

11.	Successfully calculate the total equipment needed to be ordered	This involves, selecting the name of the practical, adding number of teacher and student sets, calculating the total equipment needed.
12.	Successfully make a list that needs to be emailed to the technician	This involves making a list of all the equipment that was calculated and placing them like a message.
13.	Successfully emailing the created list to the technician	This involves successfully sending an outlook email of the list created to the technician.
14.	Select a lesson in the calendar on the homepage	This involves clicking a lesson, the name of the room, and the number of students automatically appearing on the enter information section, and the request being placed accordingly from there.
15.	Adding a tag to the application's calendar	This involves successfully adding a 'tag' to the app's calendar This will show all details of the booking – teacher name, room number, lesson time, practical name, equipment details - when clicked on. It will be visible to the whole of physics department so that clashes maybe avoided. Clicking will give a chance to edit/ amend booking.
16.	Successfully Edit/modify bookings using a tag	This involves successfully being able to amend a booking made by clicking on the calendar tag for that particular booking.
17.	Viewing a comprehensive booking history	This involves successfully being able to view the booking history.
18.	Log out	This involves successfully logging out of the application once the logout button is clicked.

(2) DESIGN OF THE SOLUTION [15 MARKS]

(I) DECOMPOSE THE PROBLEM

(a) Break down the problem into smaller parts suitable for computational solutions justifying any decisions made.

CHOICE OF PROGRAMMING LANGUAGE AND TOOLS:

Web-based application will be divided into two stages: client-side and server-side. **HTML/CSS/ JavaScript** will be used to render webpages. They will allow me to implement simple user interface for web-based app and allow me to make some decently sophisticated UI. **Python** will form the main framework of my application. Python provides access to very sophisticated web development libraries and frameworks such as **Django** – this is a web framework which makes server-side processing such as storing and receiving data from databases much simpler. Another crucial library that python provides and will be used heavily in project is **MySQL** – this will allow me to write SQL code in my python scripts and therefore allow me to manipulate databases. Each SQL entity could be representative of an OOP class in python where the attributes for the entity will be the attributes of the class. Therefore, python will also allow me to use **OOP paradigm** for development. **Flexibility** of data structures means that python will make it simpler to implement simple scheduling algorithms. Using SQL through python means that I will also use python as a procedural language – to be able to work with databases. Popularity of python and the wide programmer network, although less important, are worth noting because they make development substantially simpler and are very powerful development tools.

(II) DESCRIBE THE SOLUTION

(a) Explain and justify the structure of the solution.

(b) Describe the parts of the solution using algorithms justifying how these algorithms form a complete solution to the problem.

(c) Describe usability features to be included in the solution.

(d) Identify key variables / data structures / classes justifying choices and any necessary validation.

(e) Identified and justified the test data to be used during the iterative development of the solution.

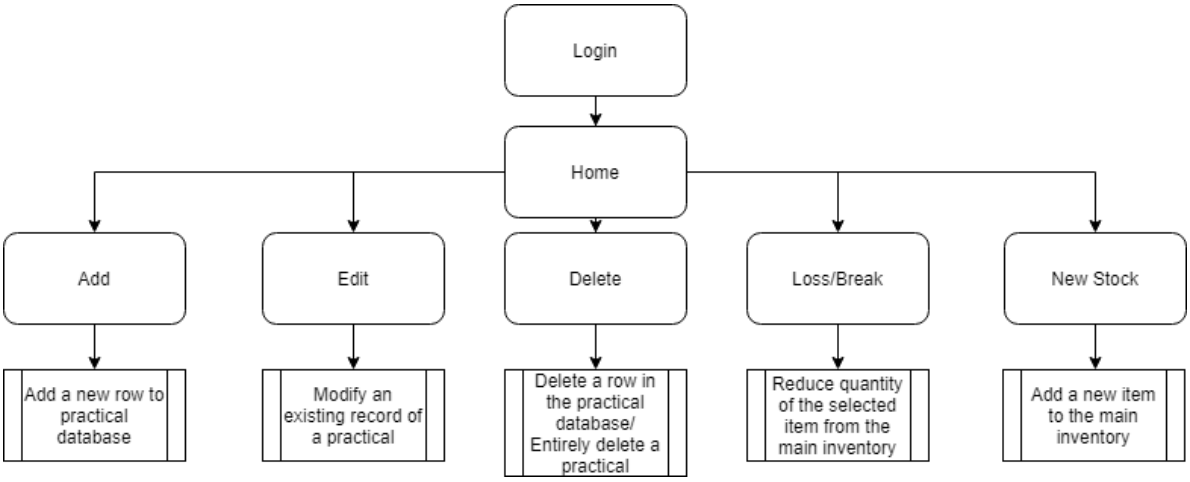
(III) DESCRIBE THE APPROACH TO TESTING

(a) Identify the test data to be used during the iterative development and post development phases and justify the choice of this test data.

APPLICATION DESIGN AND STRUCTURE:

TOP-DOWN DESIGN DIAGRAM FOR THE WHOLE APPLICATION:

Below is the rough top-down design of my application. The homepage of the application is far more complicated and involves a lot more methods and hence, I have a separate top down design for the homepage.



Justifications:

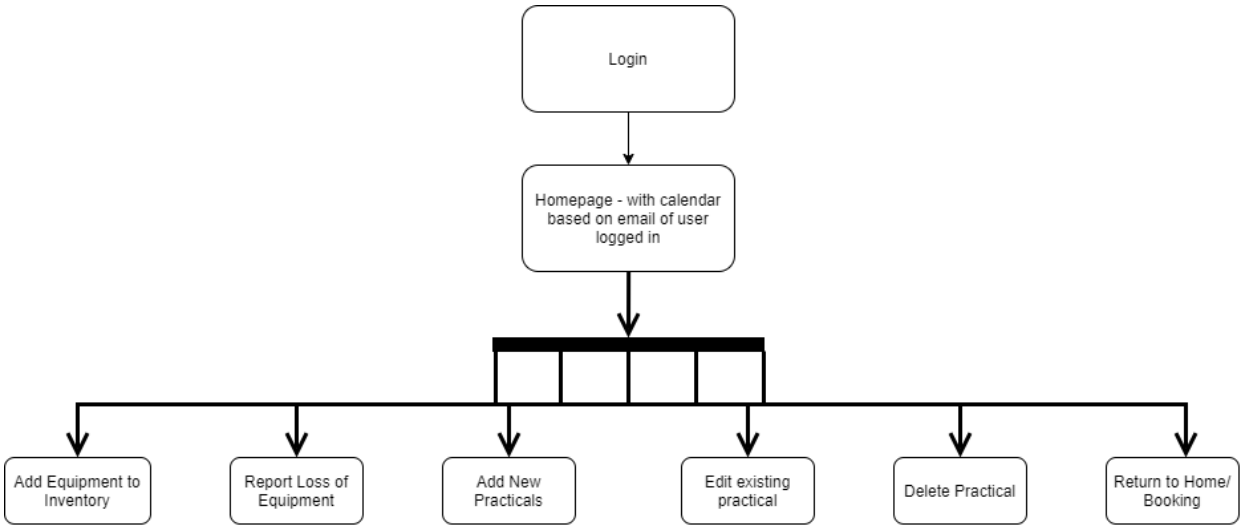
The homepage of the application will lead to 5 other webpages in the application. These will be the page to 'Add a Practical' – Add; 'Edit an Existing Practical' – Edit; 'Delete a Practical' – Delete; 'Report Loss/Breakage' – Loss/Break; 'Add New to Inventory' – New Stock.

Each of the above pages does the following:

1. **Add –** This page gives the user an option to add a new practical in the database. This is based on the stakeholder's requirement of being able to order practicals. Only when you add a practical and its needed equipment to the database is when you will be able to request equipment for that practical. Therefore, a method to add equipment is needed.
2. **Edit –** This page is used to change the equipment and/or their quantities needed in a practical. This will update the practical database and any practical ordered after an edited is made will have the edited number of quantities. This means that we need a method to be able to edit an existing record and modify the database accordingly.

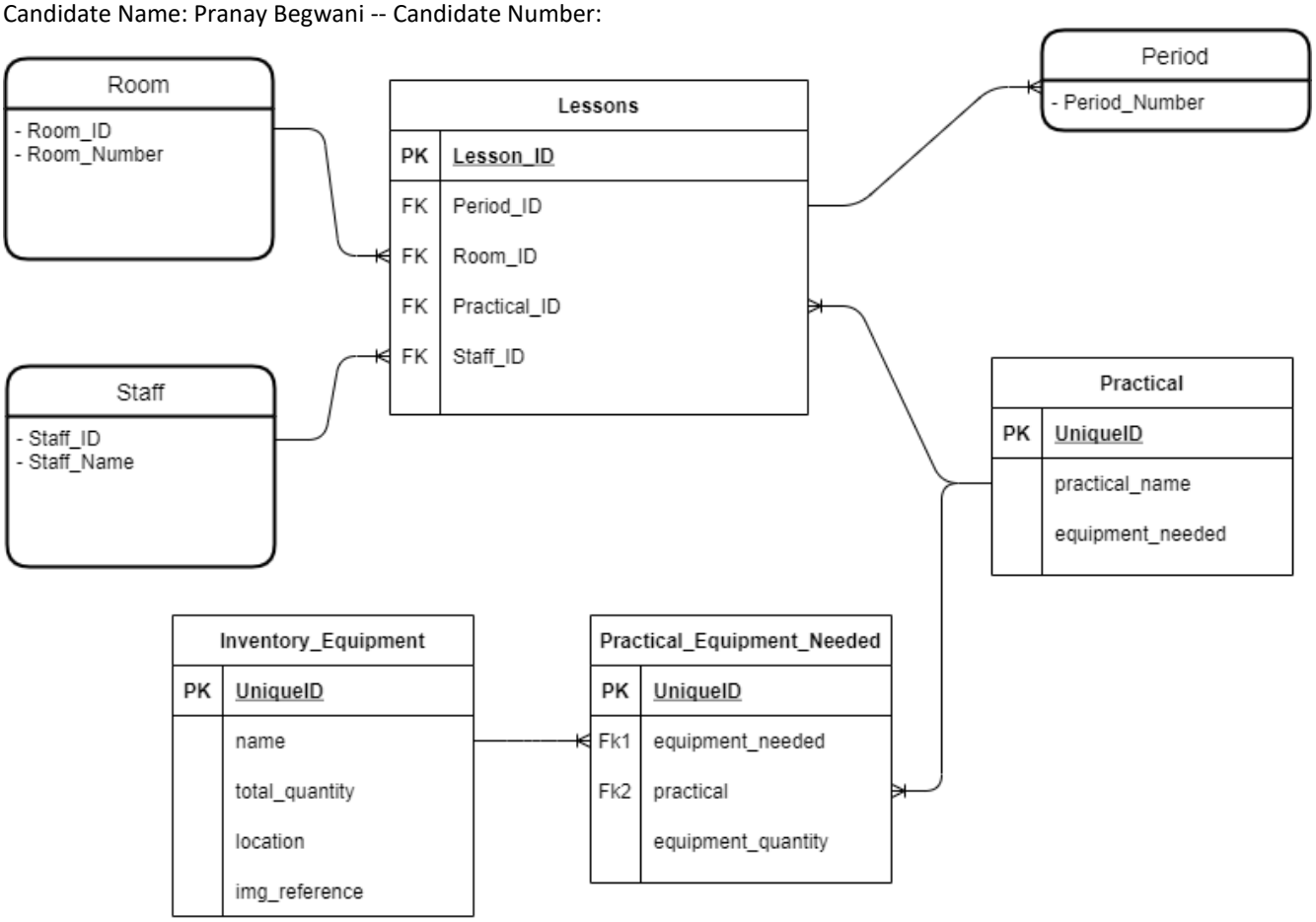
3. **Delete –** This is used to remove a practical from the practical database. A practical deleted will not be available to be requested from the homepage. This implies that a method is needed to delete a record from the database.
4. **Add Stock –** This page will be used to add new items to the inventory of the labs. This is very relevant to my stakeholder's requirement because, if the total equipment needed in the request exceeds what is available in the laboratory, the staff cannot place a request! Doing so requires a method which can be used to add either an entirely new piece of equipment or add to the quantity of an existing piece of equipment.
5. **Loss/Breakage –** This page is used to permanently reduce the amount of an equipment in the inventory, thus reducing the total available number of that particular item. Therefore, I have shown the need of a method to delete in the top-down design diagram above.

'Screenflow' diagram for the whole application is as follows:



Database entity relationship diagram for the whole app:

Each of the above blocks is representing an individual table of my database.



Justifications:

1. **Lessons –** This table is used for booking purposes. Each attribute/row in this table is related to a booking. The time/lesson of booking, the room being booked, the staff booking the room; the practical being performed during the lesson, and the number of students in the lesson.

   Attributes of this table are as follows:
   Lesson ID – This will be the primary key – a unique way to separate each booking
   Period Number – This will show the period number of booking – 1, 2, 3, 4, or 5.
   Room ID – Shows the room booked.
   Staff ID – Shows the member of staff who booked the lesson.
   Practical ID – The practical booked for the lesson.

2. **Room –** This table will store all the rooms that can be booked when booking for a lesson in the lab. The table will have a one-to-many relationship with the 'Lessons' table – one room can have many lessons booked into it.

   Attributes of this table:
   Room ID – Primary key, to identify each room.
   Room Number – The actual room name/number. Eg: B102

3. **Staff –** This table stores list of all the staff in the physics dept. This will be used to show who has made a booking. It has a one-to-many relationship with the lessons table such that each staff may book many lessons.

   Attributes of this table:
   Staff ID – Primary key, to uniquely identify each member of staff.
   Staff Name – To show the staff who booked a lesson.

4. **Period –** This table stores the list of lessons and their times. It has a one-to-many relationship with the lesson table, such that one period may have many lesson booked – but provided that rooms are different.

   Attributes of the table:
   Period Number – 1 to 5, the time booked for a lesson.

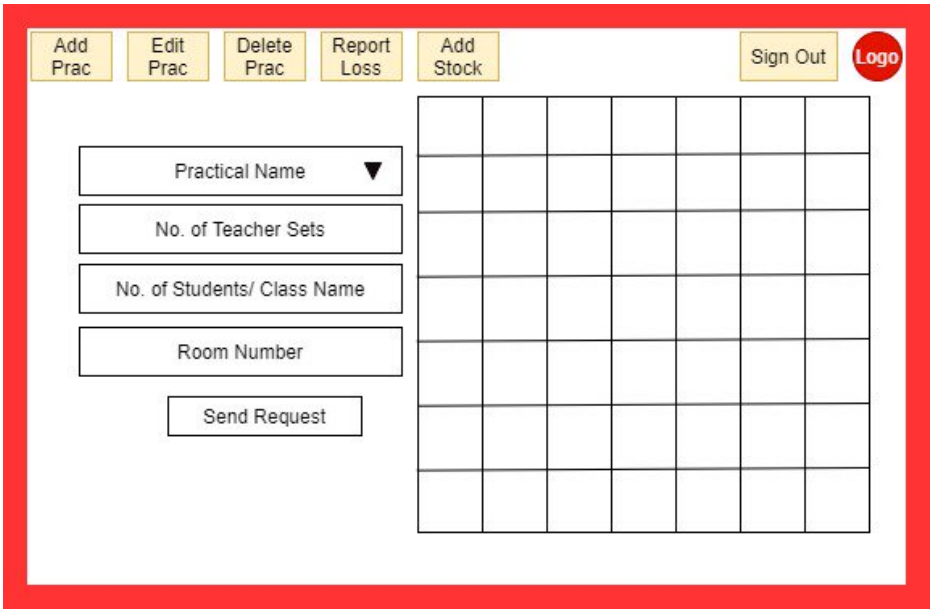
5. **Practical –** This table stores list of the practical. It has a one-to-many relationship with the lessons table such that one practical may be performed in many lessons. It has a one-to-many relationship with the lessons table such that each lesson may have many practical booked.

   Attributes of this table:
   Practical ID – Shows the ID of the practical that is booked, also the primary key.
   Practical Name – Shows the name of the practical booked for that lesson.
   Equipment Needed – Shows the equipment booked for the practical being booked.

MODULE 1: HOMEPAGE

Design:

This is a rough **plan of the homepage** of my application:



This above is the main home page of the application. This page allows the user to send a request to the technician, by selecting a class and lesson from the calendar (on the right) and selecting a practical from the drop-down menu. The page is divided into two sections and will be made using the <TABLE> tag in html. The screen is essentially one row divided into two columns – left and right.

One of the stakeholder's key requirements was a simple to use application with a clear and easily maneuverable layout. I have therefore explicitly named each of the buttons in the top row to the page that they will lead to. Based on changing requirement or development conveniences I may have to add/remove buttons (and hence, pages).

Justification of design features:

The left is the section that the user will use to select a practical, enter no. of students and teachers and send a request to the technician. This request will be sent as an outlook email. This whole process will involve a series of methods which will do as the top-down design diagram says below.

Inputs:

Practical Name – This will be selected from a dropdown menu rendered on the html page. Each option of the dropdown menu will be a record of the table that stores the names of the practical. I will use the 'names' key of the dictionary of each practical to render the name. A loop will run through the whole table to repeat this process for all the options.

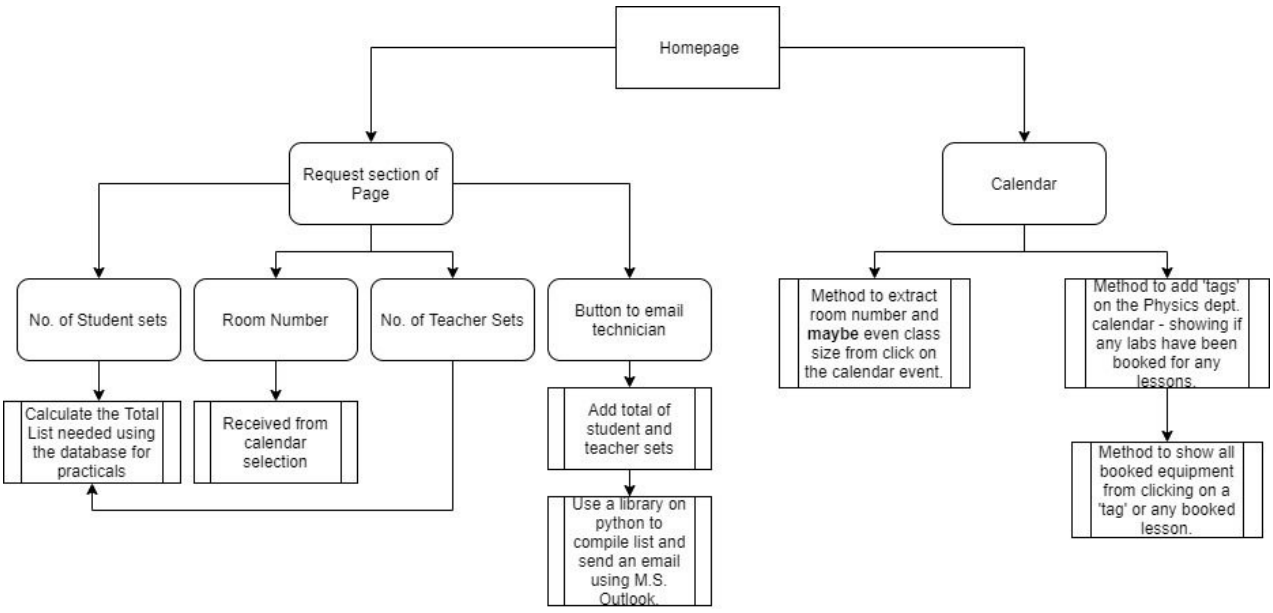
No. Of Teacher Sets – The input will be a number in the textbox. This was a request from my stakeholder and I will implement this using a simple textbox with some basic type and length validation in the backend Django.

No. Of Student Sets – This is also an integer textbox input. Again, this is key to the stakeholder's request and is central to the whole idea of my application. This will allow the application to calculate the total equipment that needs to be ordered.

Room Number – This will be a dropdown menu. The list of options will essentially be the 'Rooms' column of the room table. The user will select the room they want to book.

The right side of the page is a calendar. This calendar will be the physics department's calendar and will contain all their lessons and room numbers for the same. The user can click on the lessons and book the equipment and practical for **x** no. of students accordingly. This calendar will then show a 'TAG' which will show the booking details – this will be helpful to avoid clashes. The methods involved are shown in the top down design diagram below.

Each of the 6 blocks above is a button which do as they say.



The image **above is a hierarchy diagram for the Homepage** of my application. The two main elements of my application's Homepage, i.e. the 'Requesting Area' and the 'Calendar', have been shown below. The curved rectangular blocks simply represent the elements that will be present in the respective section of the page. The blocks with two lines are methods that will be

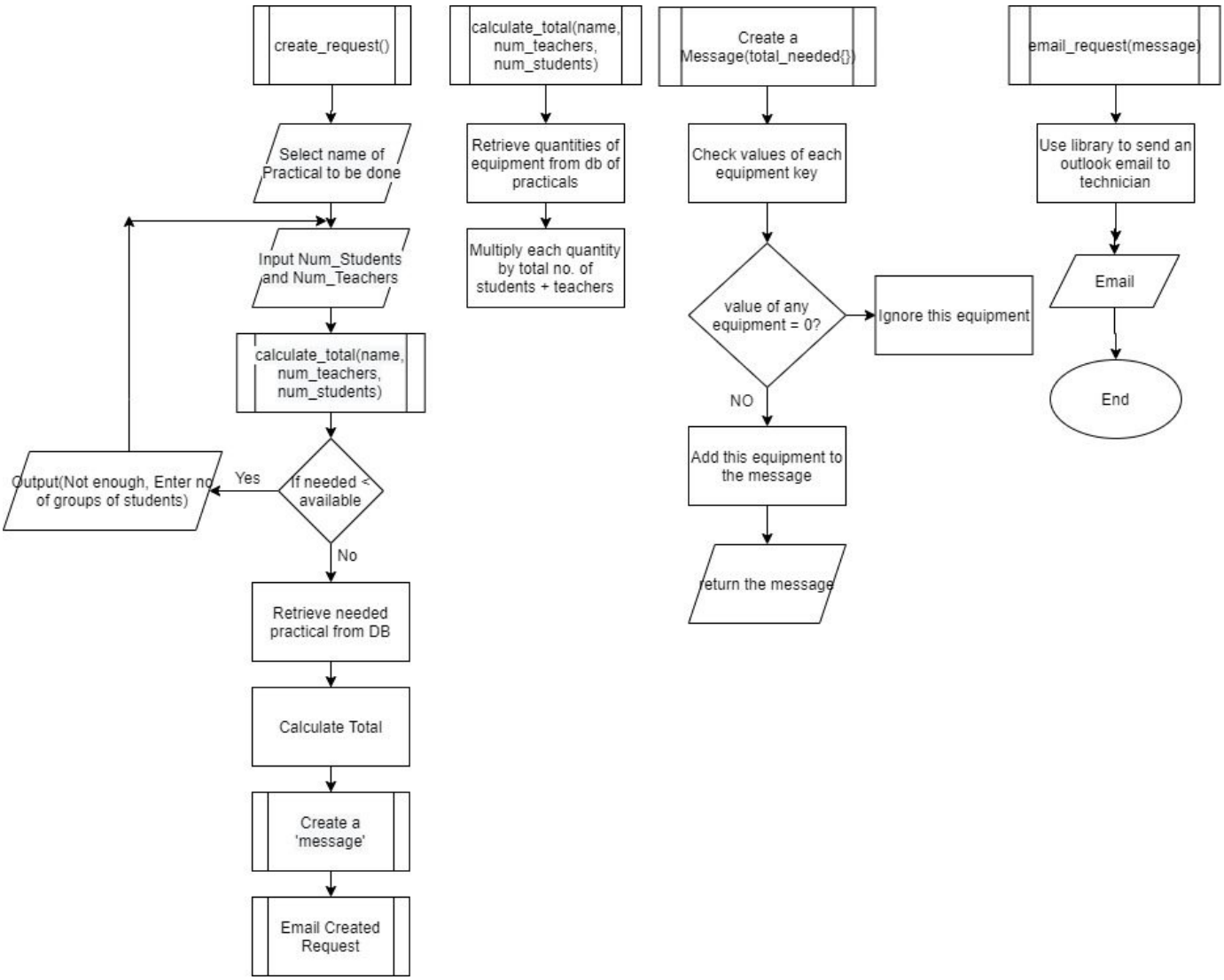
implemented by me to process the information received by the above elements of the Homepage and create the required outputs.

Implementation Considerations:

I am considering implementing an OOP paradigm for the TAGS that I want to create in the calendar of my application. The class will be the tags class and each object, ie the instance of a class, and the attributes will be the booking details. I will use a similar approach for creating the requests. This is because, the framework of the note will be the same, only some of the attributes need to change.

Flowchart:

Below is a flowchart which represents how the requesting feature of the application works:



Pseudocode:

Key variables and data structures defined:

practical_name: variable to store name of a practical
num_students: variable to store no. of students to order sets for
num_teacher: variable for no. of teacher sets needed
practical_inventory{}: dictionary which stores the equipment in ONE set, this is one column of practical table essentially
total_needed{}: dictionary to store total quantities of each equipment needed, essentially (num_students + num_teacher) * practical_inventory{}

```
total_needed{} = {
                 "name": "Hooke's Law",
                 "Rulers": 2,
                 "Springs": 5,
                  etc
                 }
```

Dinctionary contains a key and a value: the key is left coloumn and the value is on the right

Message: Variable which stores the total_needed values as a string combined together to send to technician.
room_num: Variable which stores room to be booked
practical_details{}: Same as practical_inventory{} essentially.

The main function of creating and emailing a request to the technician will be governed by the procedure 'create_request(practical_name, num_students, num_teacher, room_num)' as shown below. To summarize the pseudocode below:

- An input of the practical name is first received as a string from the drop-down menu on the html page – as selected by the user. Each practical will be added using the 'ADD PRACTICAL' page of the application. Every practical added will have their names displayed in this drop-down menu.
- Similarly, the number of students and the number of teachers are also received as inputs on the html page.
- The room number is another string input selected from the drop-down menu.
- The total equipment needed for the order is calculated using another method called 'calculate_total(practical_name, num_students, num_teacher) and the result will be stored in a dictionary called 'total_needed()' as shown above.
- This calculated total will be compared by what is available – if less order will be completed by a series of methods; if more an appropriate message will be displayed.
- Assuming that total_needed < available: We send this dictionary of equipment to a method called 'create_message' which compiles this list of equipment into a string message.

- The string message is passed as an argument to a method called 'email_request(message)' which sends the message to the technician as an email.
- The calendar tag is then created using the method 'add_calendar_tag(total_needed, room_num)' to show booking details.
- Finally, room is marked as 'BOOKED' for an hour using the method 'room_book(room_num)'.

The PSEUDOCODE for the above process is shown below:

```
/* All the arguments and parameters of this method are received from the HTML
template 'homepage' */

PROCEDURE create_request(practical_name,num_students,num_teacher,room_num):
      total_needed{}=calculate_total(practical_name,num_students,num_teacher)
      /* practical_inventory{} is dictionary which represents a record in
      practical table */

      IF (qty values in total_needed > qty values in practical_inventory
      ):
            OUTPUT ("You are asking for more than available, reduce number of
            students")
      ELSE:
            String message = create_message(total_needed)
      ENDIF
      email_request(message)
      add_calendar_tag(total_needed, room_num)
      room_book(room_num)
ENDPROCEDURE

FUNCTION calculate_total(practical_name,num_students,num_teacher)
      total_sets = num_students + num_teacher //var for total sets needed

      /* list_of_practicals[] is an array which represents the whole
      practical table */

      /* Line below gets the dictionary – practical_details{} - for the
      practical that the user wants to book, this dictionary contains details
      (equipment + quantities) of that practical */

      practical_details{} = Database.PracticalTable.get(practical_name)
      // above line gets the details of practical as stored initially

      FOR key in practical_details: /*iterate through the practical's
      details, quantities for each equipment, and calculating total */
            practical_details[key] *= total_sets //key is qty of each equip.
      ENDFOR
      return practical_details{}
```

```
ENDFUNTION

FUNCTION create_message(practical_name, total_needed)
      String message = practical_name //initially, msg. has name of prac only
      FOR key in total_needed: //key contains qty. of each equip.
            IF total_needed[key] != 0: //ignore any equip. not needed
                  message += str(total_needed[key]) + str(total_needed[val])
            ENDIF
      ENDFOR
      return message
ENDFUNCTION

PROCEDURE email_request(message):
      //use external library to send the message via email
ENDPROCEDURE

PROCEDURE add_calendar_tag(total_needed,room_num)
      /* add a tag to calendar to show the details of the practical-
      essentially displaying the arguments/parameters of this function neatly
      */
ENDPROCEDURE

PROCEDURE room_book(room_num):
      /* method to mark the selected room as booked for an hour time slot -
      each day has 5    */
ENDPROCEDURE
```

The email can be sent via Django itself. Django contains a library named, `django.core.mail.send_mail`, this allows to convert html text and pages into text for email. This is very relevant to my application and will be used to send the created message to the technician as an outlook email.

I NEED TO MAKE AN OOP HIERARCHY DIAGRAM FOR THE CALENDAR SIDE OF THE DIAGRAM ABOVE

Important Variables:

In the tests, **lookup table refers to the database tables** – essentially checking if something exists in the database!

Variable Name	Type	Role of Variable
practical_name	String	Stores the name of the practical that is being booked for a lesson
num_students	Int	Stores the number of students who will be doing the practical – as entered by the user who is booking. This is important to calculate the total number of equipment needed.

num_teacher	Int	Stores the number of teacher sets needed to perform the practical during the lesson – as entered by the user.
total_sets	Int	Stores the sum of num_teacher and num_students i.e., the total number of sets needed.
room_num	string	This value stores the room that is being booked for the particular lesson. It is the room as selected by the user, that needs to be marked as booked.
Message	String	This is the 'list' of equipment that is being ordered by the user for a particular lesson. It consists of the practical name, room number, equipment needed for the practical, and TOTAL quantity of each equipment needed (calculated).

Important Data Structures:

Name	Type	Role of DS
total_needed{}	Dictionary	It is used to store the total quantity of each equipment needed to the practical being booked. It is calculated by multiplying the number of students (entered by user) with the equipment needed in a practical (stored beforehand). This is essentially the practical_details dictionary multiplied by number of students
practical_details{}	Dictionary	Stores the details of one practical – queryed from the database using the name of the practical. Example: `{'name': 'Practical', 'test tube':10, 'springs':0, 'burner':2, …}`
practical_inventory[]	Array	This is an array which stores all the practicals that are stores in the practicals table of the database. It is used to call names of practicals and use them to get their details in the practical_details{} dictionary.

Important Methods:

Method Name	Job/Function of Method
create_request()	Makes the text/string which is to be complied in the email which will go to the technician
calculate_total(practical_name, num_students, num_teacher)	Calculate the total amount of equipment that is needed and returns it to be added in the email request
create_message(total_message)	Compiles the equipment into a message for the technician
email_request(message)	Emails the created message to the technician

add_calendar_tag(total_needed, room_num)	Adds the booking as a tag on the calendar – such that it can be viewed by anyone
room_book(room_num)	This method books the room from the room table in db

Testing:

Input Validations for textboxes:

Name	Type of Validation	Test Data	Expected	Actual	Pass/Fail
Practical Name	Presence Check	Nothing Selected Practical	Displays Warning No Change		
No. of Teacher	Presence Check	No value Some value	Prompt to input Nothing		
	Type Check	String/ Char Values Integer Values	Invalid type prompt Nothing		
	Range Check	0 999999	Too Low Too High		
No. of Student	Presence Check	No value Some value	Prompt to input Nothing		
	Type Check	String/ Char Values Integer Values	Invalid type prompt Nothing		
	Range Check	0 999999	Too Low Too High		
Room Number	Presence Check	Nothing Selected Room	Displays Warning No Change		

Other Tests:

Calendar of the Physics Dept Displayed after login	
All boxes take the correct input	
Total equipment calculated correctly	
Email sent to technician with correct equipment	
Send Request button send correct email to technician/email	
Calendar shows booking made as a tag	
Cursor becomes finger pointer and buttons' text becomes red when hovered over – ALL BUTTONS	
Add Practical button leads to respective page	

Delete Practical button leads to respective page	
Add New Stock leads to page to the add to inventory page	
Report loss/ breakage page leads to respective page	
Editing practical page leads to the respective page	

MODULE 2: ADD PRACTICAL PAGE

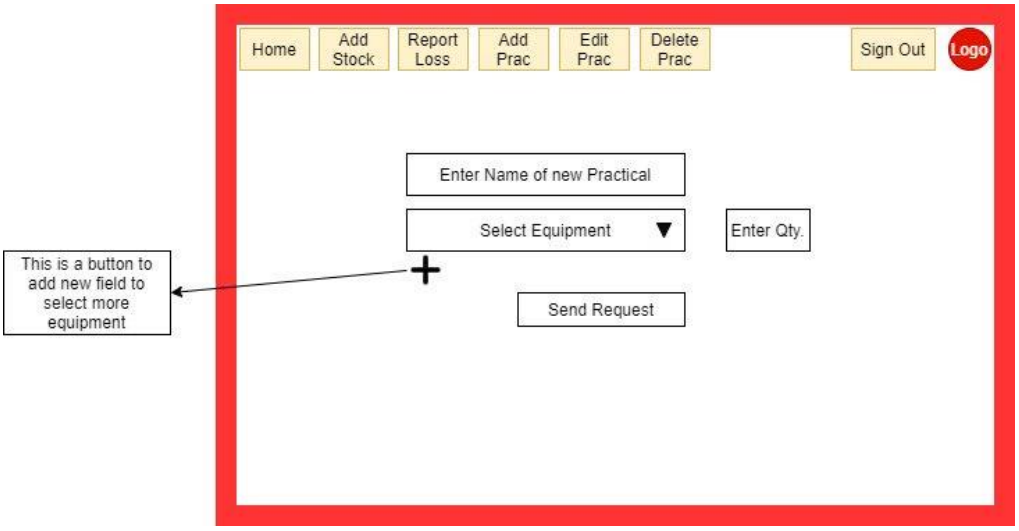
This module will be used to add practicals to the database. Each of these practicals can then be accessed when booking them for students and if editing them.

The inputs received on this page are: Practical Name, Equipment Needed in the practical and the quantity of each equipment. The **'+'** button on the bottom left of the page allows the user to add a new field to add more equipment needed to do that practical. The next input is the 'Send Request' button which confirms the addition of this practical to the database.

Throughout the page, there are a few validations for this page such as, **Presence check, type check and lookup table**. Presence Check will look for: if none of the quantities are blank, none of the equipment names are blank, and the practical name is not blank. Type Check looks for the type of data entered for each of the inputs: Names – Strings; Quantity – Int. Lookup table will be used to verify if a practical already exists and if a piece equipment added is present in the inventory.

Design:

Below is a rough design for the Add Practical Page of the application:



Justification of Design Features:

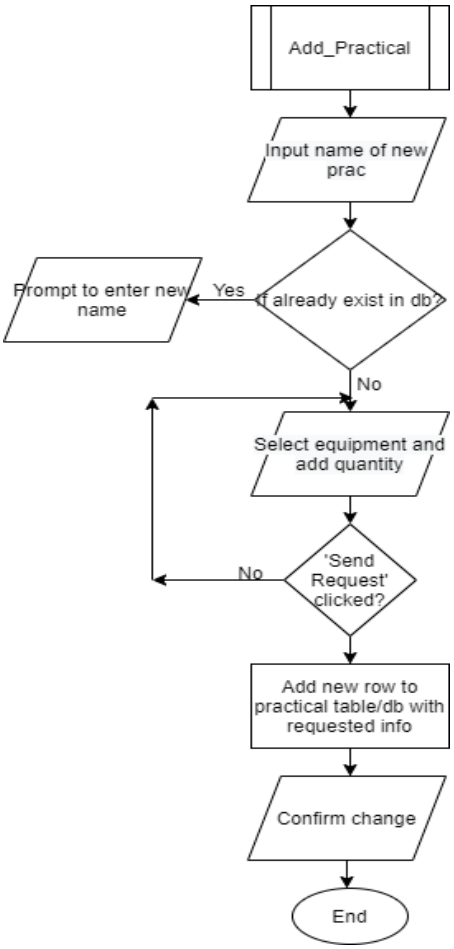
The textbox – This is useful to help name a new practical. This will be particularly helpful when retrieving details of individual practicals either to book them or display them in any situation.

Dropdown to select equipment – Used to select an equipment that exists in the inventory and enter the quantity of the same needed when doing the practical once. This is crucial

because the stakeholder's request was to be able to store equipment and their quantities under the name of the practical – this is only possible with these textboxes.

The '+' button – This button will help to add another equipment list drop down and quantity textbox, allowing the user to add more equipment under the practical. This is also a 'stylistic' addition to be able to add more equipment. Also, because we don't know how many different equipment pieces will be needed to create a practical we are leaving the option to the user to add as many things as they want to.

Flowchart:



Pseudocode:

This page needs me to add a data structure which will be key to my project when adding practical and retrieving them. The main data structure which I am considering to use an array of dictionaries. This will mirror the table in my database which has all the practicals, and the equipment needed to perform each of them, listed together.

```
Practicals =[{name: "Hooke's Law", "Ruler": 2,
……},{…},{…},{…},{…},{…},{…}, …]
```

In the code above, each of {} represents a single practical. The keys of the dictionary represent the individual equipment and the name of the practical. The values of each of the keys will be the name of the experiment and the quantities of the individual equipment respectively.

The **process of adding a new practical** is as follows:

- The user will enter name of the new practical on the html page.
- Check if this already exists – if yes: ask to change name or delete existing; if no: ignore.
- Display box to select needed equipment and enter their quantities.
- If '+' pressed, create new drop-down for equipment and textbox for quantity.
- Run the checks described above.
- Store the information of this practical into a temporary dictionary and display the addition for the user.
- Add to database.

Pseudocode:

```
// practical_name received as input from html page

PROCEDURE add_new_practical(practical_name):
      IF practical_name in database table of practicals:
            output("This practical exists, change name, or delete duplicate
            before adding this one")
      ELSE:
            practical_details{}={
                              'name'= practical_name,
                                  }
      ENDIF
      WHILE ('send request' button != clicked):
            IF ('+' button click received from html page):

                  //render textboxes and drop-downs
                  /* select name of equipment from drop down and enter
                  quantity */
                  equipment_name = textbox input
                  equipment_quantity = textbox input

                  IF (type(equipment_quantity) == int && equipment_quantity
                  != null):
                        // Add the equipment as key to the dictionary
                        practical_details[equipment_name] =
                        equipment_quantity
                  ELSE:
                        OUTPUT ("Enter Valid Quantities")
                  ENDIF
            ENDIF
      ENDWHILE
```

```
        // after send request is clicked
        OUTPUT (practical_details{})
        /* Add this information to the database so that it can be used to book
        later */

ENDPROCEDURE
```

Important Variables:

The key variables and their roles are mentioned in the table below:

Variable Name	Type	Role of Variable
practical_name	String	Stores the name of the new practical that needs to be added.
equipment_name	String	Stores the names of each equipment needed in that practical – temporarily – to be able to pass them to the practical's data structure and then the database.
equipment_quantity	Int	Again, temporarily stores quantity of the respective equipment before passing it to the data structure and then to database.

Important Data Structures:

Only one data structure is needed in this case, this will **store the new practical temporarily** before moving it to the database table.

Name	Type	Role of Data Structure
practical_details{}	Dictionary	Stores every single detail of a newly added practical. The keys of the dictionary are name and equipments.

Important Methods:

Method Name	Job/Function of Method
add_new_practical()	Method to add new practical to the table in the database. This practical can later be ordered using the homepage.

Input Validations:

Name	Type of Validation	Test Data	Expected	Actual	Pass/Fail
Name of Practical	Presence Check	No value Some value	Prompt to input something Valid		
	Type Check	String/Char Values Integer Values	Valid Data Invalid Type Prompt		
	Length Check	Text exceeding 255 characters Correct length text	Too many Valid Data		
Equipment Name (dropdown)	Presence Check	Nothing Selected Equipment	Displays Warning Valid		
Quantity of chosen equipment	Presence Check	No value Some value	Prompt to input Nothing		
	Type Check	String/ Char Values Integer Values	Invalid type prompt Nothing		
	Range Check	0 999999	Too Low Too High		

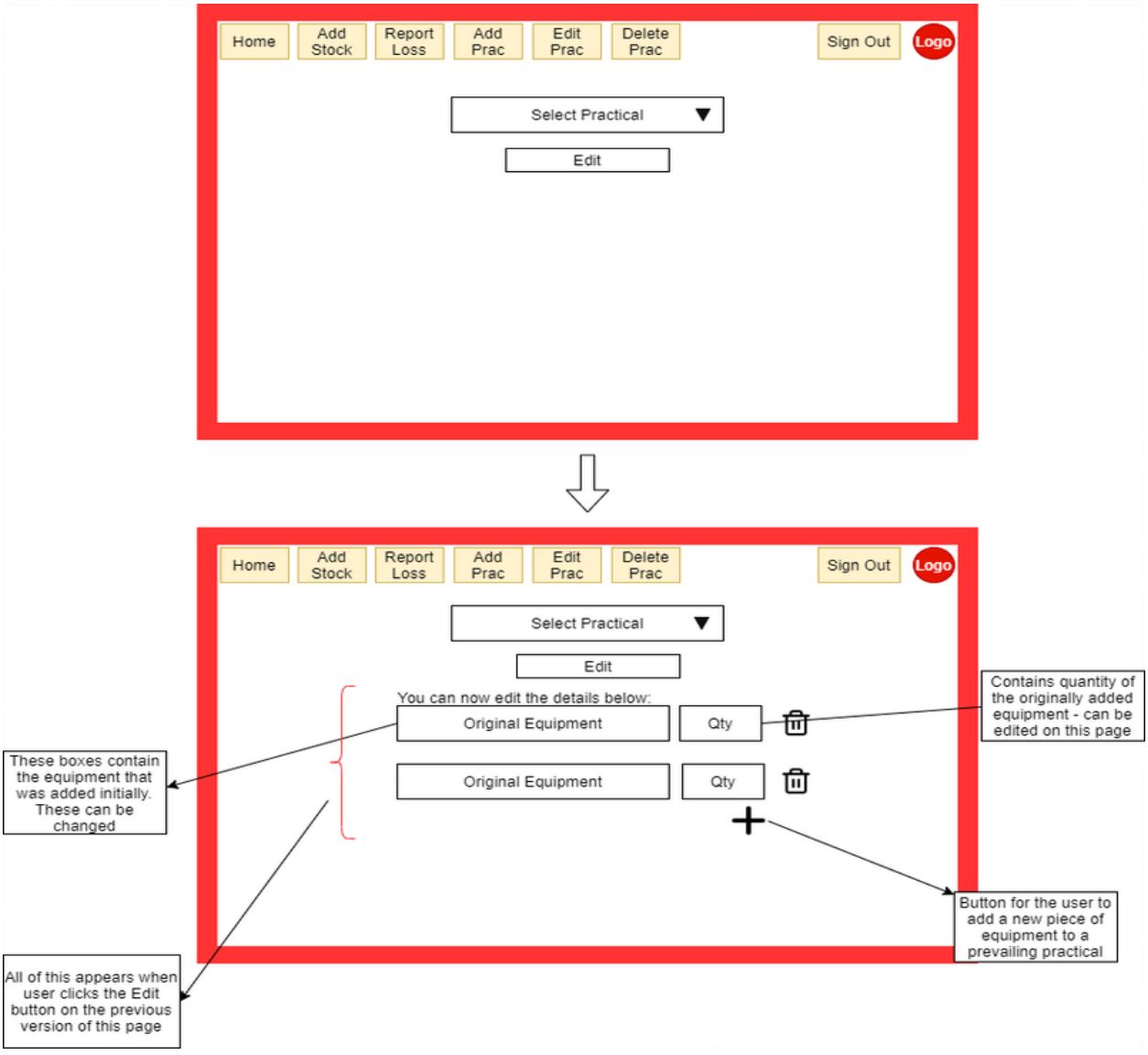
Other Testing:

Textbox for practical name displays prompts if validation is failed	
Checking if new practical is already in the database and prompting user if it is	
Display dropdown of equipment and textbox for quantities of the equipment	
Display a textbox and dropdown pair every time plus sign is clicked	
Dropdown of equipment contains all the equipment added in the inventory	
Buttons work correctly – cursor turns to pointer and text in button becomes red	
All the buttons in the navbar work as intended	
Adding the practical in practical table, linking equipment using the link table in database, and add the quantities in database	

MODULE 3: EDIT PRACTICAL

This page will be used to edit the equipment and/or their quantities for practicals that already exist in the database. The user first selects a practical from the drop-down menu and then all the equipment and their quantities are displayed in editable drop-down boxes and text boxes respectively for the user to edit.

Design:

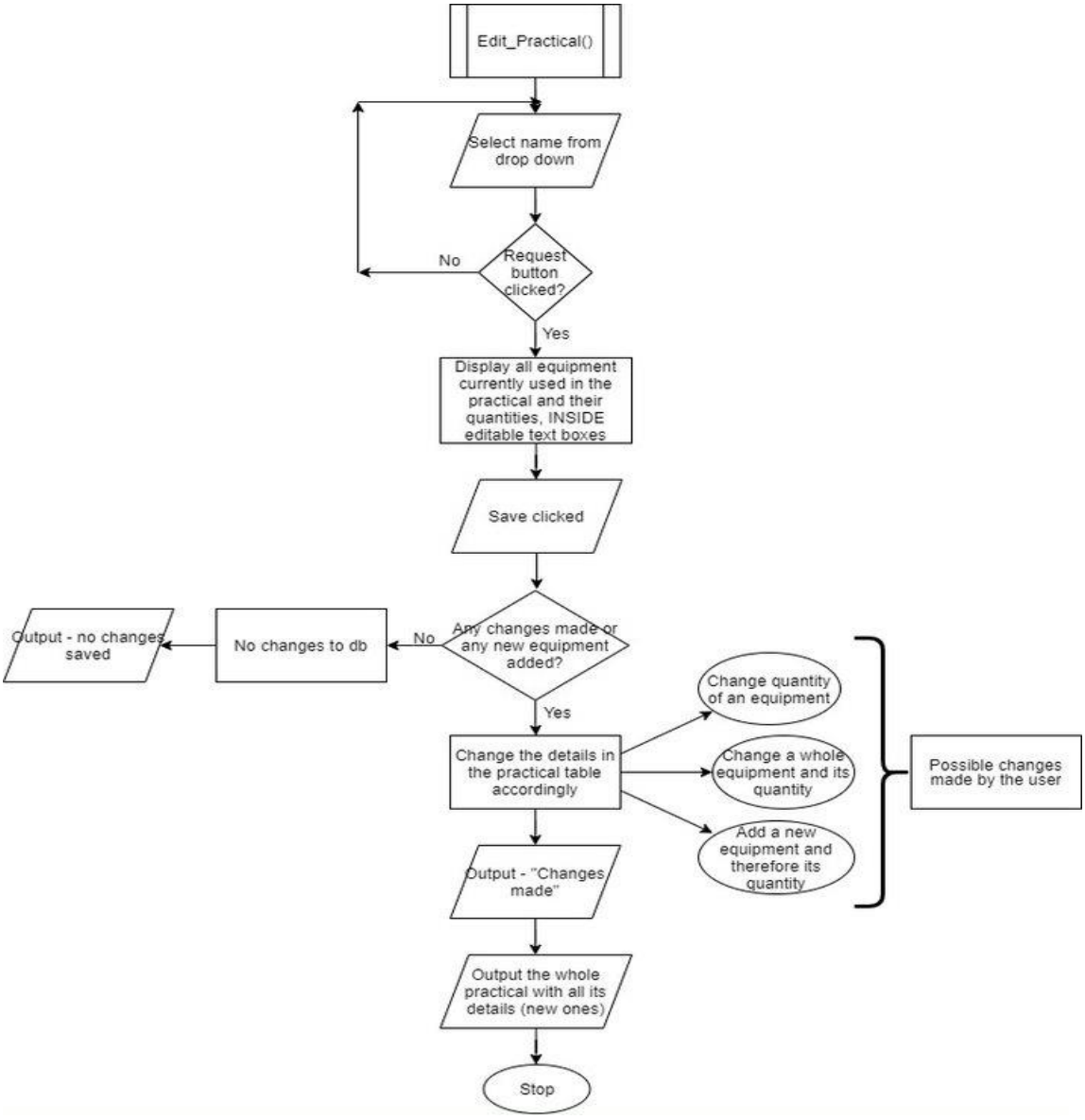


Justification of design features:

- **Dropdown menu –** This dropdown is used to select the practical that needs to be edited. Although not an explicit requirement of the stakeholder, it is obvious that there should be an option to edit details of an existing practical – the equipment or even the name.

- **Equipment Dropdown –** An **X** number of these appears when the user selects a practical from the previous dropdown. The number of these depends on the number of equipments that were initially stored by the user when adding the practical. This will allow the user to change any equipment needed.
- **Quantity Textbox –** This appears along with the equipment dropdown. It allows the user to change the quantity of a piece of equipment needed in a practical.
- **'+' button –** Used to add a new pair of equipment dropdown and quantity textbox. This button allows the user to add new equipment that a practical may need.
- **Delete Button –** This is the 'Trashcan' icon next to a dropdown and textbox pair. This allows the user to delete a piece of equipment that was added when adding the practical.

Flowchart:



Pseudocode:

This page works as follows:

1. The user selects name of the practical they want to edit from the drop-down menu.
2. The dictionary practical_details holds the equipment and details for the practical that the user wants to update.
3. Check if request button is clicked. If clicked, display all the equipment and their quantities in editable textboxes.
4. Validate the inputs for: presence and type check. Presence check: Tests if a quantity is actually entered or an equipment is actually selected; Type Check: Validates if the data type of the quantity entered is int.
5. If all tests passed, prints confirmation, and displays the updated practical.
6. Update practical database.

```
/*practical_details{user_choice} is a dictionary which contains name of the
practical the user wants to edit and the equipment stored in it (with
quantities)*/

PROCEDURE edit_practical(practical_details{user_choice}):
      IF (request button clicked):
            FOR key in practical_details:
                  IF practical_details[key] != 0):
                        /*Display the key and its value in drop down and
                        textbox resepctively*/
                  ENDIF
            ENDFOR
      ENDIF
      IF (equipment_name changed OR equipment_quantity changed OR new
      equipment added) && Save button clicked:

            FOR key in practical_detials:
                  // make needed changes first to dictionary and then db
            ENDFOR

            print("Changes have been made")
            print(Practical_details{user_choice}) /*prints the updated
            practical and its details */
      ENDIF
ENDPROCEDURE
```

Important Variables:

Below are the key variables needed in this module:

Variable Name	Type	Role of Var
equipment_name	String	Stores the current name and then the updated name of an equipment that is/was stored in for a practical.
equipment_quantity	Int	Stores the current and then the updated quantity of an equipment that is/was stored in for a practical.

Important Data Structures:

There is only one important data structure needed for the development of this module.

Name	Type	Role of Data Structure
practical_details{}	Dictionary	This dictionary stores all the details of a practical as they are retrieved from the database. Changes needed are then made to keys in the dictionary and then sent back to database.

Important Methods:

Method Name	Job/Function of Method
add_new_inventory()	Method to add new equipment to the inventory table in the database. This equipment can later be ordered using the homepage to use in a practical

Input Validations:

Name	Type of Validation	Test Data	Expected	Actual	Pass/Fail
Name of Practical (dropdown)	Presence Check	No value Some value	Prompt to input something Valid		
Equipment Name (dropdown)	Presence Check	Nothing Selected Equipment	Displays Warning Valid		
Quantity of chosen equipment	Presence Check	No value Some value	Prompt to input Nothing		
	Type Check	String/ Char Values Integer Values	Invalid type prompt Nothing		
	Range Check	0 999999	Too Low Too High		

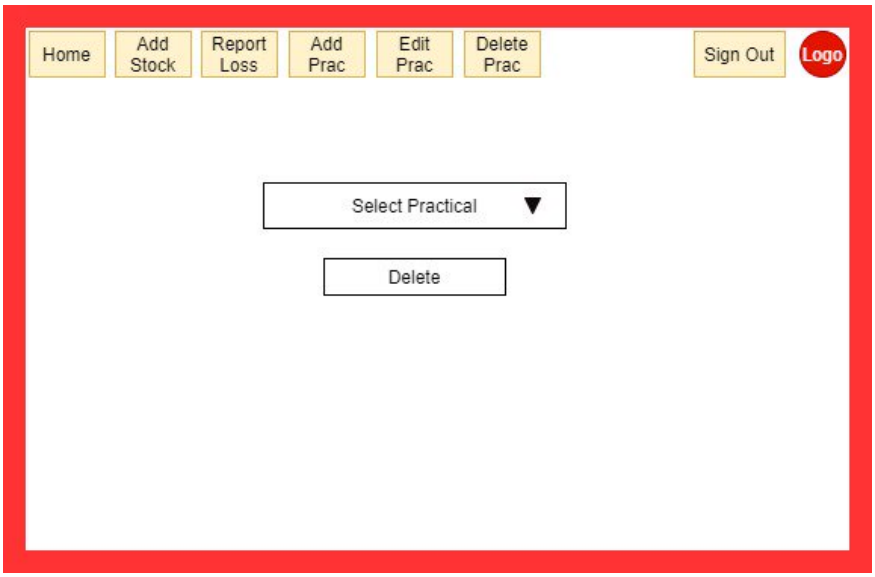
Other Testing:

Dropdown for practical name displays names of all the practical in the database	
Clicking Edit displays all the current equipment and quantities of the chosen practical in dropdowns and textboxes respectively	
Clicking the 'dustbin' deletes the respective equipment from the database	
Display an empty/default textbox and dropdown pair every time plus sign is clicked	
Dropdowns of equipment contains all the equipment added in the inventory	
Buttons work correctly – cursor turns to pointer and text in button becomes red	
All the buttons in the navbar work as intended	
Database reflects the changes made when 'Send Request' clicked by user.	

MODULE 4: DELETE PRACTICAL

This module is used to delete existing practicals. The only job this page does is deleting the practical that is selected by the user. Although not explicitly a stakeholder requirement, this page is crucial so that staff can remove any unwanted practicals from the database.

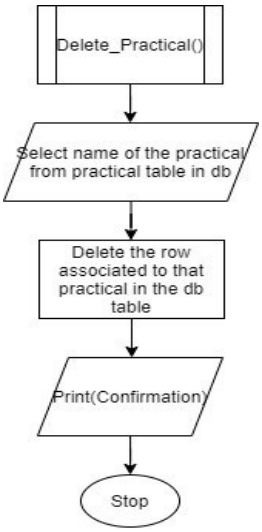
Design:



Justification of Design Features:

Dropdown – This is used to select the practical that the user wishes to delete. The list of all practicals will be generated from the database and user will be prompted to select the practical that they wish to delete.

Flowchart:



Pseudocode:

User selects practical from drop-down menu and deletes it!

```
PROCEDURE delete_practical(practical_name): /* practical_name will be
retrieved from the webpage based on what user selects from the drop
down menu */
      //use mysql commands to delete the record of that practical
      output("Deleted")
ENDPROCEDURE
```

Important Variables:

Variable Name	Type	Role of Variable
practical_name	String	Stores the name of practical that needs to be deleted

No data structures needed on this page.

Important Methods:

Method Name	Job/Function of the Method
delete_practical(practical_name)	Entirely deletes the practical from the Practicals table (table used to store and book practicals) in the database.

Validations:

The only validation needed in this page is a presence check to see if the user has actually selected a practical that they want to delete.

Other Testing:

Display all the practical in the dropdown menu	
Successfully delete the selected practical from the database when delete is clicked by the user	
All other navbar buttons work	

Looking at how simple this page is, I have decided to remove this entirely and add its functionality to the EDIT PRACTICAL page.

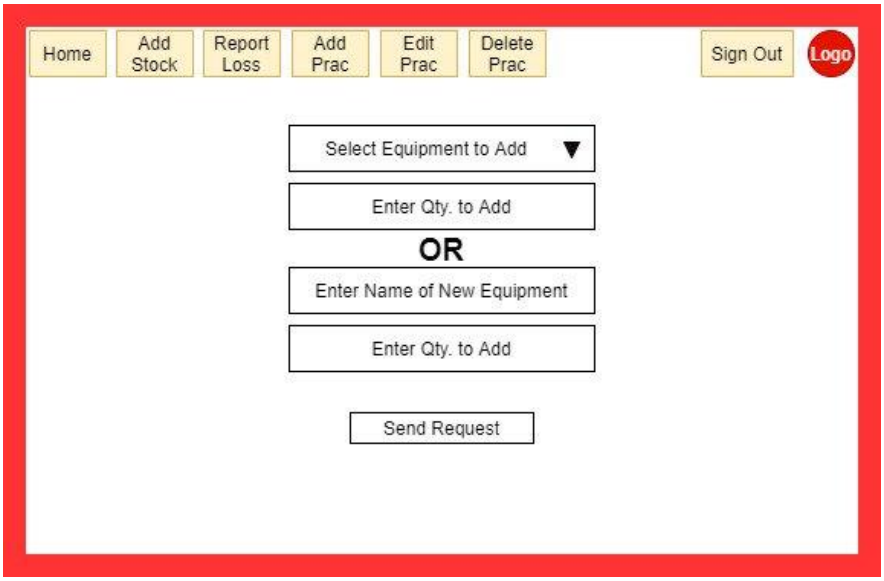
MODULE 5: ADD TO INVENTORY

This module in my application will be used to add item to the inventory. This can be used in situations such as buying new equipment, adding new stock, or setting up the initial inventory. The inputs given this page are simple – the user has the choice to either select an existing equipment and increase its quantity, or they can type an entirely new piece of equipment an enter its quantity.

Based on a later request from the stakeholder:

I have added two more things: A field to enter location/room for the new equipment AND an option to upload the image of the piece of equipment.

Design:



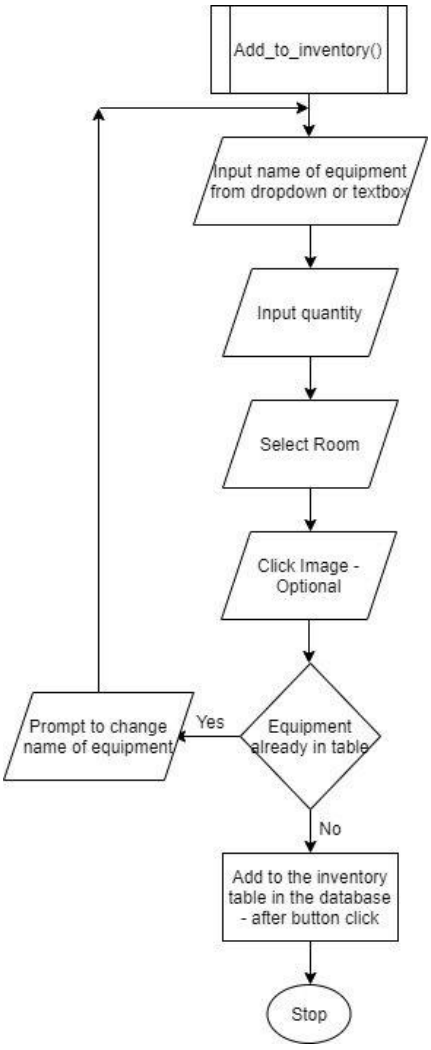
Justifications for design features:

This page is important because the lab needs to keep account of what all they have in the inventory. They cannot lend something they do not have available and they cannot use something that is not even existing in the school! To keep track of how equipment is 'moving' around the school, they need to have an inventory which shows the flow equipment in the school from staff to staff. This page lets the staff add items to the inventory.

1. **Dropdown to select Equipment –** This is used to select name of an equipment which already exists in the database – in case, the staff want to add stock/ buy more of what they have already.
2. **Quantity Textbox –** This first textbox allows the user to enter the quantity of the existing equipment they are adding to the inventory.

3. **Name Textbox –** This is used if the user wants to add an entirely new thing to the inventory – something they do not already have. The box accepts string values.
4. **Quantity Textbox –** Integer box used to enter quantity of the new item that staff needs to add to the inventory.
5. **Room No. –** Although not in the diagram, this dropdown menu will be added. Based on the stakeholder requirements, the inventory must also show where an item is located. Therefore, this dropdown menu displays a list of all the physics rooms in the building and allows the user to select where a new piece of equipment will be kept.
6. **File Field –** Again, based on stakeholder requirement, this field will be added to this page of the application. The stakeholder wanted the users to be able to store an image of where the equipment is located. Therefore, I will add a field which can upload an image/ its reference to the database table, to show where newly added equipment is located and how its kept there.

Flowchart:



Pseudocode:

There are a few types of validations in the case when user types in an entirely new piece of equipment: a **presence check** to see if any equipment is added AND if any quantity is entered; a **type check** because the new equipment name should be a string and the quantities should be int.

There is one more thing which needs to be checked: If the user types in a piece of equipment which already exists they will be prompted to select that from the drop-down menu instead.

```
// equipment_name = Entered in Textbox OR Select from drop down
// equipment_quantity_add = Entered quantity in textbox

PROCEDURE add_new_inventory(equipment_name, equipment_quantity):
      IF (equipment_name in textbox is already in table):
            output("Enter new equipment or select this existing one
            from drop down")
      ELSE IF (equipment_new NOT in table):
            OUTPUT (equipment_name + "added " + "Quantity: " +
            str(equipment_quantity_add))
            /*Add new row for this equipment to the Inventory table of
            database*/
      ELSE:
            // Add entered qty to existing qty of selected item
            Equipment_name.quantity += equipment_quantity_add
      ENDIF
ENDPROCEDURE
```

Important Variables:

Key variable needed for the development of this module are:

Variable Name	Type	Role of Variable
equipment_name	String	Stores the name of the new equipment that is to be added to the database – as entered by the user. This can either be from dropdown (when adding more of an existing equipment) or something entirely new.
equipment_quantity_add	Int	Stores the quantity of the new equipment that needs to be added to the database as entered by the user.

Important Methods:

Method Name	Job/Function of Method
add_new_inventory()	Method to add new equipment to the inventory table in the database. This equipment can later be ordered using the homepage to use in a practical.

Input Validations:

Name	Type of Validation	Test Data	Expected	Actual	Pass/Fail
Name of Equipment	Presence Check	No value Some value	Prompt to input something Valid		
	Type Check	String/Char Values Integer Values	Valid Data Invalid Type Prompt		
	Length Check	Text exceeding 255 characters Correct length text	Too many Valid Data		
Quantity of equipment to add	Presence Check	No value Some value	Prompt to input Valid		
	Type Check	String/ Char Values Integer Values	Invalid type prompt Valid		
	Range Check	0 20 99999	Too Low Valid Too High		

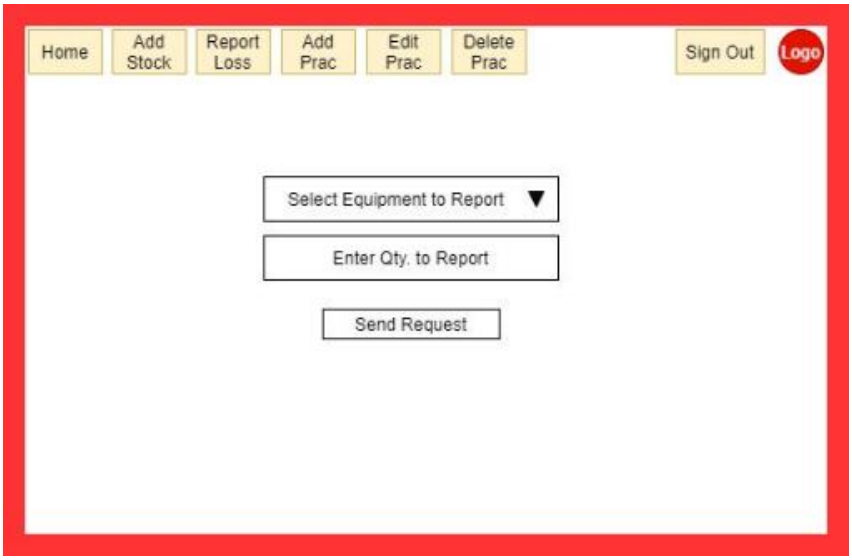
Other Testing:

Dropdown to add to existing equipment displays all the equipment names, currently in the inventory	
Validation for new equipment name is working and textbox takes in new name	
Textboxes for equipment quantity working	
Buttons work correctly – cursor turns to pointer and text in button becomes red	
All the buttons in the navbar work as intended	
Adding the equipment and quantity of the equipment is seen in the database	

MODULE 6: REPORT LOSS/BREAKAGES

This page is similar to the last one. It is used to remove items from the inventory permanently – if lost or broken. The only inputs needed are name of the equipment lost and the quantity of it lost. The page is very important to reflect accurately, at any given point in time, everything that is available in the school inventory when it comes to staff members/ technician making bookings and viewing the inventory.

Design:

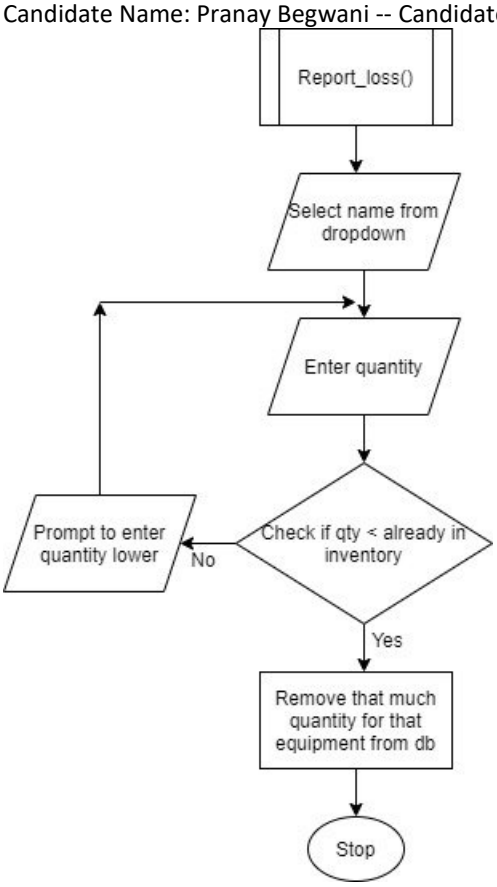


Justification for design features:

This page is important because it allows the users to be able to report losses and breakages of items from the inventory. Realistically, in a school there maybe lots of breakages during experiments. Hence, this page will allow staff to report losses as soon as they occur thus, reducing confusion when practicals are being booked and maintaining a 'real-time' system showing staff what is available at any given time.

1. **Equipment dropdown –** This dropdown will help select the name of a piece of equipment which needs to be removed from the database table.
2. **Quantity Textbox –** Allows user to enter the quantity which needs to be removed for a particular piece of equipment.

Flowchart:



Pseudocode:

Needs a presence check for both fields and a type check for the quantity to make sure it is of int type. A check will be made so as to ensure that the quantity entered for removal is not more than the quantity that is actually available!

```
// equipment_name = Selected from drop down
// quantity_remove = Entered in textbox
PROCEDURE report_loss_equipment(equipment_name, quantity_remove):
      total_quantity = /*accessed using inventory table in db and then
      the record for that equipment_name */
      IF (quantity_remove > total_quantity):
            output("Make sure qty to remove is less than available
            quantity")
            report_loss_equipment()
      ELSE:
            total_quantity = total_quantity - quantity_remove
      ENDIF
      output("Remaining quantity is " + total_quantity)
ENDPROCUEDURE
```

Important Variables:

The key variables needed to develop this module are as follows:

Variable Name	Type	Role of Variable
equipment_remove	String	Stores the name of the equipment that is to be reported for loss/breakages.
quantity_remove	Int	Stores user input of the quantity that needs to be subtracted from the quantity existing
total_quantity	Int	Stores the quantity in the database at any given time

Important Methods:

Method Name	Job/Function of Method
report_loss_equipment()	Method to remove lost or broken equipment from the inventory table in the database.

Input Validations:

Name	Type of Validation	Test Data	Expected	Actual	Pass/Fail
Equipment Name (dropdown)	Presence Check	Nothing Selected Equipment	Displays Warning Valid		
Quantity to remove	Presence Check	No value Some value	Prompt to input Nothing		
	Type Check	String/ Char Values Integer Values	Invalid type prompt Nothing		
	Range Check	0 More than existing	Too Low Too High		

Other Testing:

Dropdown for equipment name displays names of all the equipment in the database	
Textbox for quantity to remove works correctly and follows validations	
Buttons work correctly – cursor turns to pointer and text in button becomes red	
All the buttons in the navbar work as intended	
Database reflects the changes made when ‘Send Request’ clicked by user.	

OTHER PROGRAMMING TECHNIQUES & CONSTRUCTS USED THROUGHOUT THE APPLICATION

Django Template Language:

Django uses what is known as DJANGO TEMPLATING LANGUAGE to help create dynamic and static web pages. Each HTML file in Django is called a template.

This equips me with the ability to utilize what is known as template inheritance. This is very much like Objected Oriented Programming: An HTML page, in my case **base.html**, will contain HTML and CSS for any widget which needs to appear on all HTML pages that inherit from this page.

This can be done by adding the following line to the start of each page which we want to be inheriting from base.html.

For example: Homepage.html contains:

```
{% extends 'base.html' %}
```

This line is added to the top of every page which want to be inheriting from base.html.

Not only this, but Django Templating Language also allows me to use programming constructs such as selection and iterations on my templates – or HTML files!

This will allow me to add FOR loops and WHILE loops in my HTML templates. These will be helpful when it comes to dynamically displaying lists of equipments, practicals, room nos., etc as drop-down menus.

Object Oriented Programming:

Each table in my database will be represented using an individual class. The methods in the classes will be in a separate python file that will perform the CRUD operations on all the attributes (columns) of the class (table). There is no major inheritance features in my application.

Classes include: Inventory, Practical, Rooms, and Lessons – the attributes are shown in the entity relationship diagram at the start of the section.

(3) DEVELOPING THE SOLUTION (25 MARKS)

(I) INTERATIVE DEVELOPMENT PROCESS

(a) Provide annotated evidence of each stage of the iterative development process justifying any decision made.

(b) Provide annotated evidence of prototype solutions justifying any decision made.

(II) TESTING TO INFORM DEVELOPMENT

(a) Provide annotated evidence for testing at each stage justifying the reason for the test.

(b) Provide annotated evidence of any remedial actions taken justifying the decision made.

INTRODUCTION

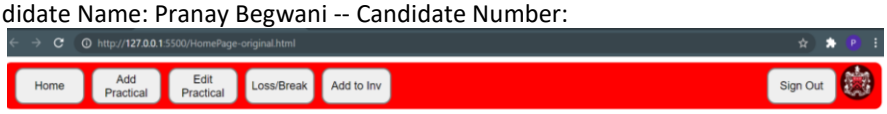
The development of the application will take place in iterative stages where each individual stage will solve one problem at a time. After each development stage, I will take feedback from the user and build upon it and develop a new part/ work on a previous part of the application.

ITERATION 1 – STATIC UI

The major goal of this iteration was to develop a static HTML/ CSS user interface for the web application. Given the fact that the stakeholder wanted a simple UI, I have tried to keep the same very simple and minimalistic. Doing so required abstract thinking – I had to remove features and elements which added a layer of complexity to the application and keep only the bare minimum needed for the application to function.

Stages of development in this section:

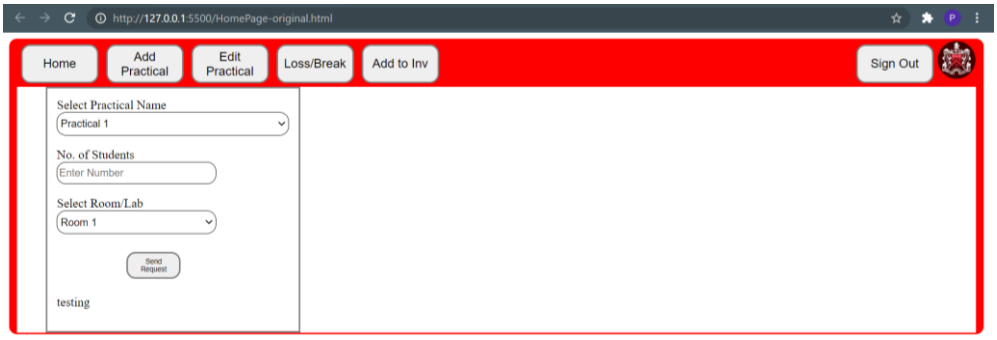
1. Create a homepage:
   My initial approach was to create a homepage for the application. I added a red background and a simple page. The page had a navbar on the top which had 5 buttons – mentioned above in Module 1 of the design section.



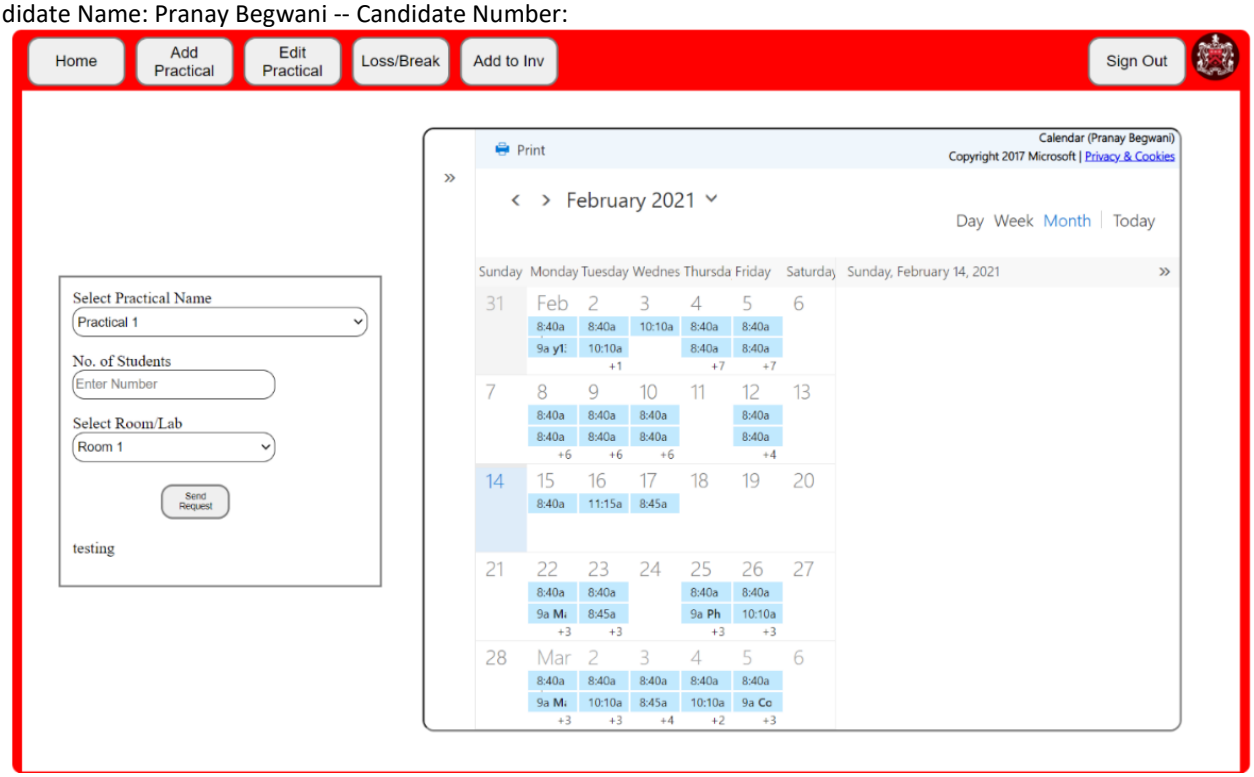
2. Following is the code for the navbar:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>HomePage</title>
        <meta charset="UTF-8">
        <link rel="stylesheet" href="HomePageDesign-original.css">
    </head>

    <body>
        <div class="mainWindow" style="color: black;">

            <div id="TopBar">
                <div id="BarRight">
                    <button class="buttons" id="SignOutButton">Sign Out</button>
                    <a href="https://www.patesgs.org/"><img id="logo" alt="Pate's Logo" src="schoolLogo.png" width="50px" height="50px"></a>
                </div>

                <div id="BarLeft">
                    <a href= "HomePage.html">
                        <button class="buttons" id="Home">Home</button>
                    </a>
                    <a href="AddNewPrac.html">
                        <button class="buttons" id="Add">Add Practical</button>
                    </a>
                    <a href="EditButton.html">
                        <button class="buttons" id="Edit">Edit Practical</button>
                    </a>
                    <a href="ReportButton.html">
                        <button class="buttons" id="Report">Loss/Break</button>
                    </a>
                    <a href= "AddtoInventory.html">
                        <button class="buttons" id="Inventory">Add to Inv</button>
                    </a>
                    </div>
                </div>
```

3. The next stage was to add a section to be able to order equipment.



4. After giving space to order equipment from the homepage, I had to add a calendar to this page. The calendar was a key user requirement and had to be implemented at all costs. This was perhaps the most challenging part of the homepage UI and required some tweaks to the CSS stylesheet of this page. After the CSS tweaks, the page looked something like this:



The calendar I my personal Outlook calendar's shareable .html link. The iframe tag of html enabled me to

5. The HTML code for the remainder of the homepage is below:

```
        <div class = "OrderEquipPage"> <!-- This is a row -->
            <div class = "TheForm"> <!-- Splitting the row above into columns, this one is for taking input-->
                <form action="#" method="POST"> <!-- Add backend link to python file to connect all inputs to the backend -->

                    <label for='Practical Name'> Select Practical Name</label>
                    <br>
                        <select id="practical-selector" name="name" type="text" placeholder="Select Practical">
                            <option value='Practical 1'>Practical 1</option>
                            <option value='Practical 2'>Practical 2</option>
                            <option value='Practical 3'>Practical 3</option>
                            <option value='Practical 4'>Practical 4</option>
                            <option value='Practical 5'>Practical 5</option>
                        </select> <!-- Replace with sqlite link -->

                    <p> No. of Students <br>
                        <input id="student-number-box" name="Select class/ No. of Students" type="text" placeholder="Enter Number"> <!-- Changed to drop down menu -->
                    </p>

                    <label for='Room Number'> Select Room/Lab</label>
                    <br>
                        <select id="room-number-box" name="name" type="text" placeholder="Select Room/Lab">
                            <option value='Room 1'>Room 1</option>
                            <option value='Room 2'>Room 2</option>
                            <option value='Room 3'>Room 3</option>
                            <option value='Room 4'>Room 4</option>
                            <option value='Room 5'>Room 5</option>
                        </select> <!-- Replaced with sqlite link when connected to django -->
                    <br>
                    <br>
                    <button class= "buttons" id="send-request">Send Request</button>

                </form>
                <p>testing</p>
            </div>

            <div id = "TheCalendar" unselectable="on"> <!-- Splitting the row above into columns, this one is for putting the calendar -->
                <iframe id="calendar" src="https://tinyurl.com/yxf3v8pm" height="700" width="770"></iframe>
            </div>
        </div>
    </div>
</body>
</html>
```

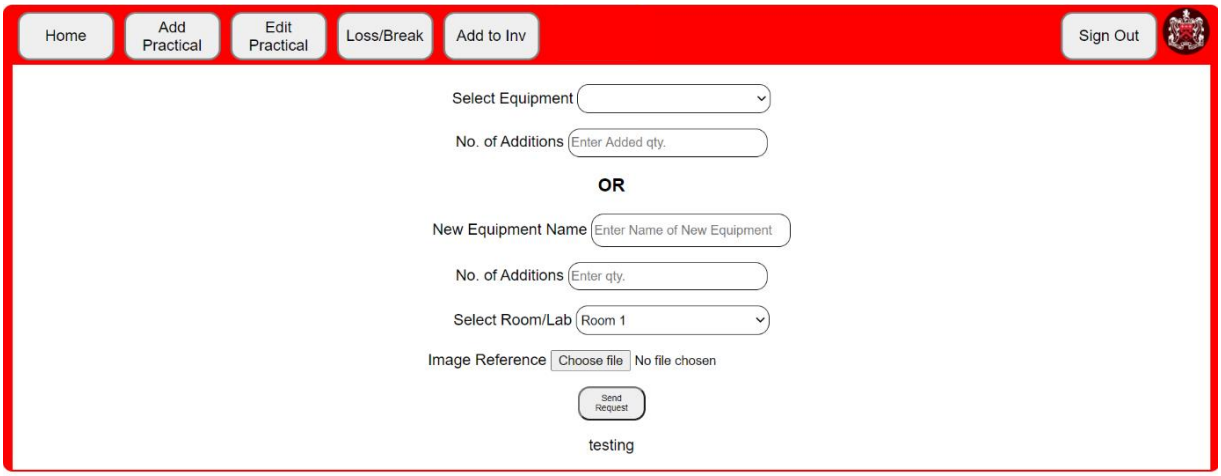
6. The CSS stylesheet of the homepage is:

```css
.mainWindow{
    border-style: solid;
    border-color: red;
    border-radius: 10px;
    border-top-width: auto;
    border-bottom-width: auto;
    border-left-width: 10px;
    border-right-width: 10px;
    width: 98%;
    height: 98%;
}

#TopBar{
    background-color: red;
    overflow: auto;
}

.buttons{
    color: black;
    border-style: solid;
    border-color: grey;
    border-radius: 12px;
    font-size: 16px;
    cursor: pointer;
    margin: 5px;
    width: 100px;
    height: 50px;
    padding: 5px;
    text-align: center;
}

.buttons:hover{
    color: red;
    border-radius: 12px;
}

#BarRight{
    display: flex;
    float: right;
}
```

```css

#BarLeft{
    display: flex;
    float: left;
}

.OrderEquipPage:after {
    display: flex;
    content: "";
    flex-direction: row;
    clear: both;
}

.TheForm{
    padding: 1%;
    margin: auto;
    margin-right: auto;
    margin-left: 3%;
    display: inline-block;
    align-self: center;
    vertical-align: middle;
    border: 2px solid grey;
}

#practical-selector{
    width: 300px;
    border-radius: 12px;
    background-color: transparent;
    height: 30px;
    border-color: black;
    border-style: solid;
}

#student-number-box{
    width: 200px;
    border-radius: 12px;
    background-color: transparent;
    height: 25px;
    border-width: 1px;
    border-color: black;
    border-style: solid;
}
```

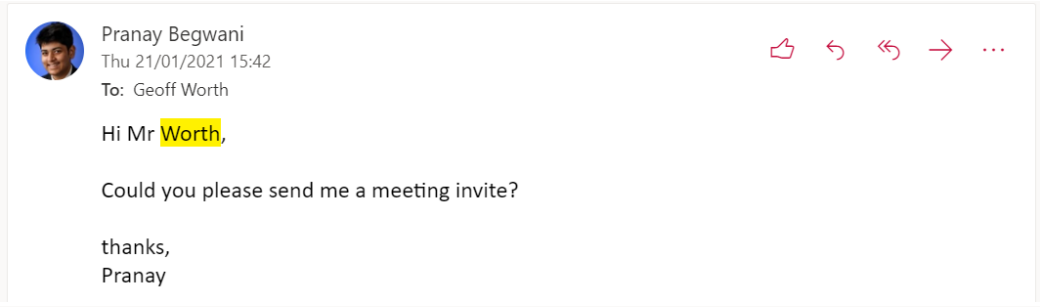
```css
#room-number-box{
    width: 206px;
    border-radius: 12px;
    background-color: transparent;
    height: 30px;
    border-color: black;
    border-style: solid;
}

#TheCalendar{
    padding: 3%;
    margin: auto;
    display: inline-block;
    align-self: middle;
    vertical-align: middle;
}

#calendar{
    border-radius: 12px;
    border-color: grey;
    width: 60vw;
    height: 100vh;
    position: relative;
}

#send-request{
    border-radius: 12px;
    width: 70px;
    height: 35px;
    font-size: 9px;
    text-align: center;
    margin-left: 30%;
}
```

7. The next stage was to add the navbar onto each of the individual pages of the application. The code for the navbar was simply copied and pasted in this because this is a static html file.
8. After this I focused on the pages which **add and remove from the inventory**. The styling of this page was later modified and the textboxes were made same in size and given rounded edge.



9. The pages for adding/removing practical were left intentionally to begin with because of how complicated they were. These pages need some backend development before I could implement their user interfaces.
10. After this, I organized a TEAMS meeting with my stakeholder and demonstrated them the working of the user interface elements.

Conversation Screenshots:

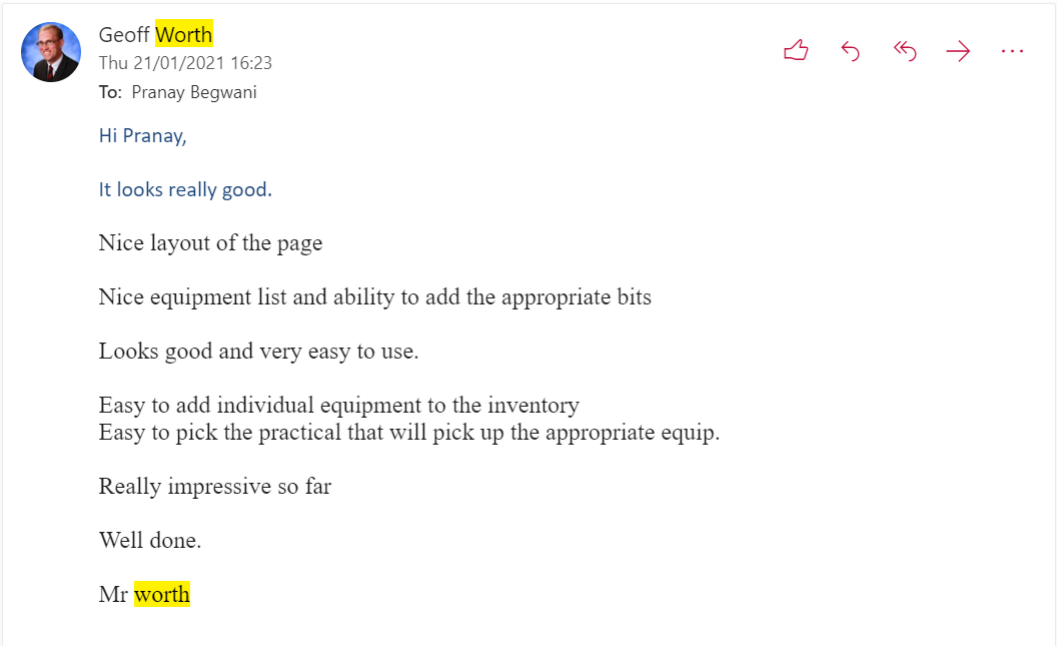


Pranay Begwani
Thu 21/01/2021 15:42
To: Geoff Worth

Hi Mr Worth,

Could you please send me a meeting invite?

thanks,
Pranay



Geoff Worth
Thu 21/01/2021 16:23
To: Pranay Begwani

Hi Pranay,

It looks really good.

Nice layout of the page

Nice equipment list and ability to add the appropriate bits

Looks good and very easy to use.

Easy to add individual equipment to the inventory
Easy to pick the practical that will pick up the appropriate equip.

Really impressive so far

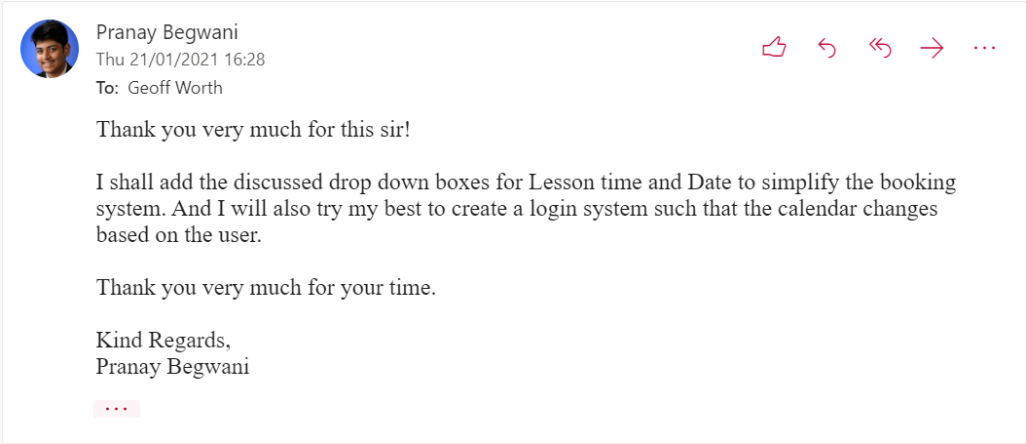
Well done.

Mr worth

Based on our conversation over the TEAMS meeting, there were a few points/ changes to the application that I noted:

- Making the calendar dynamic i.e. the calendar changes based on who logs in to the system.
- Add textboxes for Lesson Number and Date on the homepage of the application, so that bookings are easier to make.

- Style uniformity over the different pages of the app such that the different pages of the app look and feel similar.



Pranay Begwani
Thu 21/01/2021 16:28
To: Geoff Worth

Thank you very much for this sir!

I shall add the discussed drop down boxes for Lesson time and Date to simplify the booking system. And I will also try my best to create a login system such that the calendar changes based on the user.

Thank you very much for your time.

Kind Regards,
Pranay Begwani

11. Their feedback was as follows. They recommended certain changes. Login change will be implemented when I create the login and date/time and lesson number fields were added as follows.

ITERATION 1 REVIEW:

What has been done?

The basic framework of the application's user interface has been completed. The homepage's first design was completed. I imported my personal Microsoft Calendar's shareable link to meet the stakeholder's request of having a calendar – this link can later be modified to meet the stakeholder requirement completely. The page needed to Add Items to Inventory and Remove Items from Inventory were completed. I added a navbar to all the pages of the application and created blank templates for the remaining pages. There was some CSS styling added to the homepage and to the pages concerned with the inventory and this styling will form the rest of the basis of the application. At this stage, the program can navigate between all the pages of the application and the homepage was basically ready.

How was it tested?

The individual HTML pages were tested as they were being built. I installed an extension on Visual Studio Code known as Live Server which allowed me to view changes on the HTML pages as soon as changes were made in the HTML code/ CSS script. This made the testing very easy.

I made sure all buttons were functional by navigating to the different pages using the navbar.

I then set-up a video call with my stakeholder to ensure that they had also tested the application's user interface and given me their feedback on the same. The results of our conversation have been included in the above section of this iteration. In the next stage of the application I will work on some of the stakeholder feedback. Although the application's user interface looked good stylistically, the stakeholder asked for some changes in the inputs: addition of new textboxes/dropdowns and a dynamic calendar which changes based on the user who has logged in.

Testing methods undertaken:

Calendar of the Physics Dept Displayed	
All boxes take the correct input	Not yet done
Total equipment calculated correctly	Not yet done
Email sent to technician with correct equipment	Not yet done
Send Request button send correct email to technician/email	Not yet done
Cursor becomes finger pointer and buttons' text becomes red when hovered over – ALL BUTTONS	
Add Practical button leads to respective page	
Delete Practical button leads to respective page	
Add New Stock leads to page to the add to inventory page	
Report loss/ breakage page leads to respective page	

Editing practical page leads to the respective page	

Success Criteria and Stakeholder demands met:

2.	Physics Department/Any required calendar visible on the homepage	Successfully displaying a required calendar on the homepage of the application.
3.	Functional User Interface elements and simply layout of the application	Navbar of the application works properly, user can navigate between all pages of the application easily. The layout of all the UI elements like textboxes, dropdowns, buttons is simple and can be navigated easily.

As shown in the section before, the stakeholder was clearly satisfied with the criteria number 3, the user interface was simple and easy to navigate. The calendar link can also be changed very easily to display whichever calendar the user wants to.

Changes in the design section:

So far, I have followed my design section to make the user interface framework of the application. I have just not made some of the modules of the application because they required me to start the backend development of the application.

Summary of the whole Project as a prototype at this stage:

Basic and static UI of the application has been made. I can now move on to convert these to dynamic Django templates and connect them to the backend of the application to build on from there.

ITERATION 2 – DJANGO BACKEND + DYNAMIC WEBPAGES

The primary focus of the second iteration was to convert the static HTML files into Python and Django compatible 'templates', work on some of the user feedback, and program the backend system for adding things to the inventory. Over this iteration I used elements of Django Templating Language – a 'sub-system'/part of Django which allows to implement logic in HTML pages, templates in the case of Django. Doing so allowed many things:

1. Removing repetitive template code like that for the navbar using what is known as template inheritance. All the elements which need to be in all pages (navbar mainly) were placed in a file called 'base.html'. All other files which needed these elements AND also needed their own elements were 'told' to inherit from the base.html file – or `{{extends 'base.html'}}`, in this case.
2. When converting the static webpages into Django compatible templates, I had to place the navbar in the base.html file. This is the file that all other html templates inherit from. The `{% load static %}` command in the code below is needed to render css styling for these pages. All of the 'blocks' in `{% %}` allow the user to add content which is unique to the page that otherwise inherits from base.html.
3. Below is the code for the `base.html` file.

```
{% load static %} <!-- essential to be able to add css to page -->
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>{% block title %}HomePage{% endblock %}</title>
        <!-- The block means that the content between these tags can be different for each page that inherits from base -->
        <meta charset="UTF-8">
        <link rel="stylesheet" href="{% static 'css/HomePageDesign.css' %}">
        {% block extrahead %}{% endblock %} <!-- Extrablock is needed so that I can add CSS for individual pages that inherit-->
    </head>

    <body id="bg">
        <div class="mainWindow" style="color:□black;" >

            <div id="TopBar">
                <div id="BarRight">
                    <button class="buttons" id="SignOutButton">Sign Out</button>
                    <a href="https://www.patesgs.org/"><img id="logo" alt="Pate's Logo"
                        src="{% static 'images/schoolLogo.png' %}" width="50px" height="50px"></a>
                </div>

                <div id="BarLeft"> <!-- All the individual buttons for the navbar -->
                    <a href= "/">
                        <button class="buttons" id="Home">Home</button>
                    </a>
                    <a href="/AddPractical">
                        <button class="buttons" id="Add">Add Practical</button>
                    </a>
                    <a href="/EditPractical">
                        <button class="buttons" id="Edit">Edit Practical</button>
                    </a>
                    <a href="/LossReport">
                        <button class="buttons" id="Report">Loss/Break</button>
                    </a>
                    <a href= "/AddInventory">
                        <button class="buttons" id="Inventory">Add to Inv</button>
                    </a>
                    <a href= "/ViewInventory">
                        <button class="buttons" id="View_Inventory">View Inventory</button>
                    </a>
                    </div>
                </div>

                {% block content %}
                    <!-- All the content between these tags will be unique to the page that inherits from base -->
                {% endblock %}

            </div>
        </body>
</html>
```

4. After this, I was required to link all the pages using the `urls.py` files in the application and the project. This file guides the flow of webpages and determines which URL leads to which webpage. This was multi-step process: (i) I had to create a file with all the URLs in it; (ii) I had to program view functions which determined which function in the backend is carried out when a particular URL address is called.
5. The `urls.py` file for the **Inventory app** looked like this:

```python
from django.urls import path

from . import views # . is from 'this app' import views

urlpatterns = [
    path("", views.homepage, name="home"), #name must be same as the function name in the views.py, the path('') in the urls.py of mysite looks for the same path
    path("LossReport/", views.report_loss, name="loss_report_page"),
    path("AddInventory/", views.add_to_inventory, name="add_to_inventory"),
    path("AddPractical/", views.add_new_practical, name="add_new_practical"),
     path("EditPractical/", views.edit_practical, name="edit_practical"),
]
```

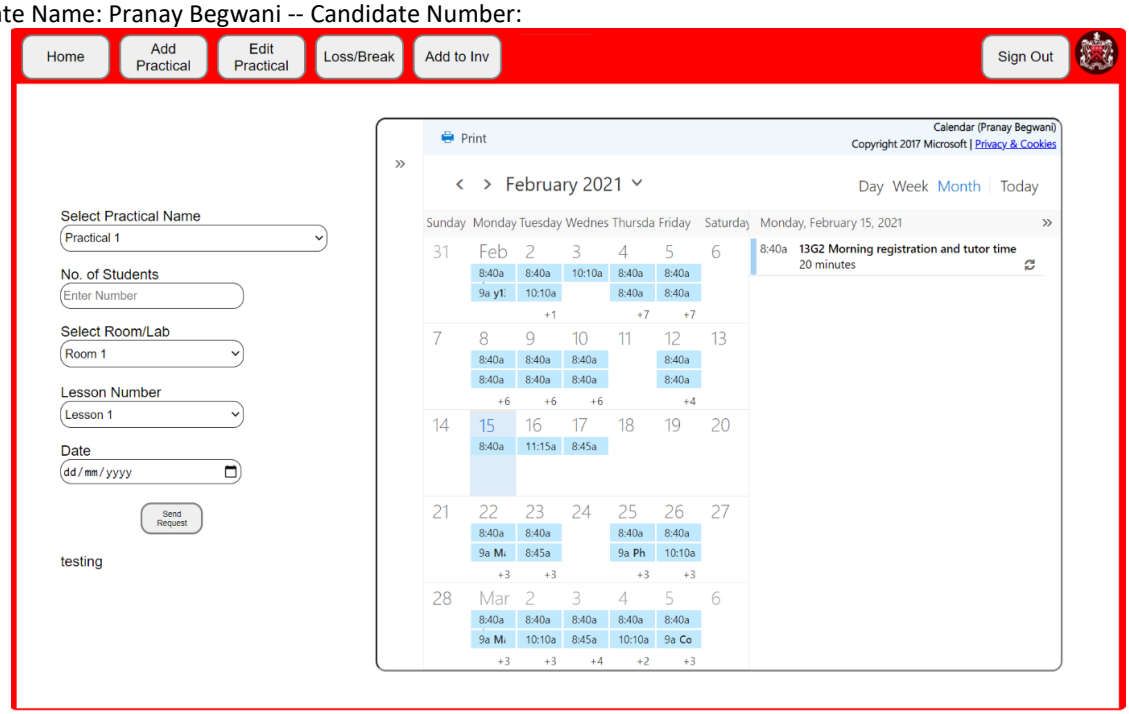
   The `urls.py` file for the **Project**, as a whole, looked like this:

```python
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('home/', include('main.urls')),
]
```

6. The `views.py` file to render each of these templates looked like this:

```python
from django.shortcuts import render #import a libray from the fjango framework to render the webpages
from django.http import HttpResponse    #importing library used to send and receive data to and from webpages
# Create your views here.

def homepage(response):
    return render(response, 'main/HomePage.html', {}) # render(type of render, 'name of html template', {context})
    # context is the data passed from the backend server to the template front end

def report_loss(response):
    return render(response, 'main/LossOrBreakReport.html', {})

def add_to_inventory(response):
    return render(response, 'main/AddToInventory.html', {})

def add_new_practical(response):
    return render(response, 'main/AddNewPractical.html', {})

def edit_practical(response):
    return render(response, 'main/EditPractical.html', {})
```

7. This was followed by me adding the fields/dropdowns for the user to enter the lesson number and date of the booking they were making. This was **one of the major suggestions made by the stakeholder from the previous iteration** and so had to be made. The homepage now looks like this:



8. After this, I made the stylistic changes as requested by the stakeholder. Doing so enhanced the looks of the website and made the style consistent all the way through the website. The layout of the textboxes and font above is what I have used for the remaining application. I can amend this later should the stakeholder need me to.
9. The next and perhaps the most important part of this iteration was to start the development of the back-end of the inventory in my application. The inventory is where all the individual quantity is stored in the database. To start, I created a table called '**Inventory**' in the database of this application. For the database, I used **MySQL** through an XAMPP server. By default, Django uses SQLite3 and therefore, I had to make a few amendments to the `settings.py` file in the application so that it linked to the correct database table and server. Below are the changes made to the settings.py file.
10. Initially the settings.py file looked like this, with the sqlit3 database connection:

```
74  # Database
75  # https://docs.djangoproject.com/en/3.1/ref/settings/#databases
76
77  DATABASES = {
78      'default': {
79          'ENGINE': 'django.db.backends.sqlite3',
80          'NAME': BASE_DIR / 'db.sqlite3',
81              }
82  }
```

11. To make the application compatible with MySQL, I had to make the following changes to the above code:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',   #link to the backend database engine
        'NAME': 'physics laboratory',           #name of the database
        'HOST': 'localhost',                    #name of the host- localhost while development
        'PORT': '3306',                         #port number for connection to the server
        'USER': 'root',                         #username for connection to the database server
        'PASSWORD':'',                          #password for connection to the database server
        'OPTIONS': { # additional settings for the connection to the database server
            'init_command': "SET sql_mode='STRICT_TRANS_TABLES'"
        },
    }
}
```

The table in the database was name **Inventory** and the table looks like this:

	#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra
☐	1	id 🔑	int(11)			No	*None*		AUTO_INCREMENT
☐	2	name	varchar(255)	utf8mb4_general_ci		Yes	*NULL*		
☐	3	total_quantity	int(11)			Yes	0		
☐	4	Location	text	utf8mb4_general_ci		Yes	*NULL*		
☐	5	Img_Reference	text Media (MIME) type: image/jpeg	utf8mb4_general_ci		Yes	*NULL*		

12. The details of the individual columns of this table are as follows:
    - The **id is a primary key –** an identifier used to uniquely identify a record in a table. It is always unique for every new record and is automatically incremented to a value higher than the older one.
    - Name – this is used to store the name of a newly added equipment
    - Total_quantity – this stores the total quantity of an equipment available in the department
    - Location – refers to where a particular equipment is stored in the school, assumes a particular equipment is stored in one place only
    - Img_reference – stores the string url/reference for an image uploaded by the user.

    These above columns have the following attributes and features:

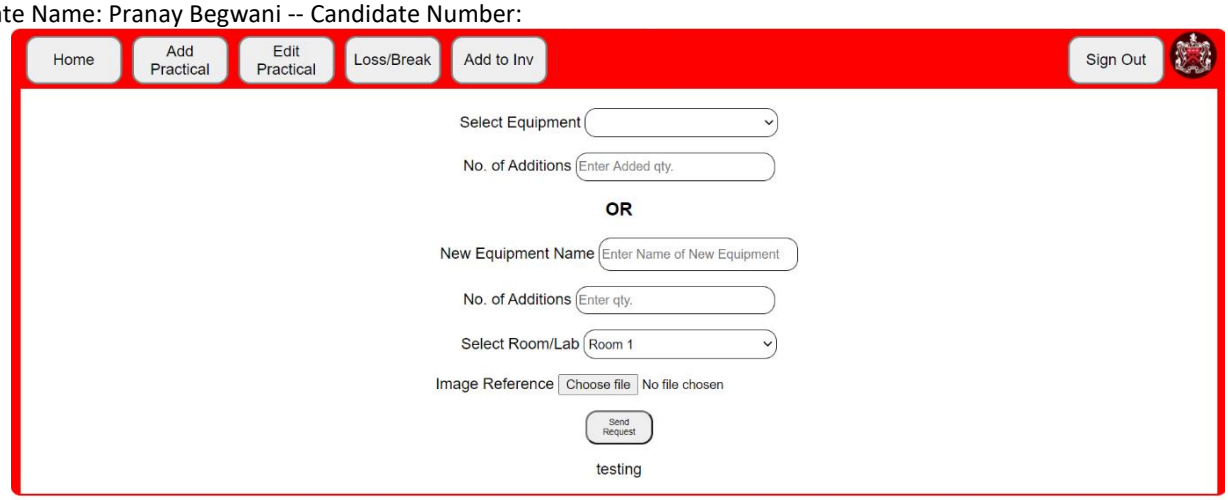
    - TYPE – It is used to describe the type of data stored in a field.
    - NULL – This is a constraint to tell whether or not a block in the table can be null.
    - DEFAULT – The value given to a field if it is left blank.
13. This was followed by creating a `models.py` file. This file utilized elements of object-oriented programming where each class – a blueprint for a set of objects – represents an individual table and each of the attributes of this class are the individual columns of the Inventory table. The attributes of this class had to be modified several times to be in conjunction with the database table on the server. The initial iteration of this file looked like this:

```python
from django.db import models


class Inventory_Equipment(models.Model):
    id = models.AutoField(primary_key=True)
    name = models.CharField(max_length=255, null=False, default="Name this Equipment")
    total_quantity = models.IntegerField(null=False)
    location = models.CharField(max_length=20, default='Physics Office')
    img_reference = models.ImageField(null=True)

    class Meta:
        db_table="inventory"
```

14. After testing out various parameters for the individual 'fields' of the table – or attributes of the class in this case – I came up with this version of the same file.
    I changed the layout of this file so that code was more legible.

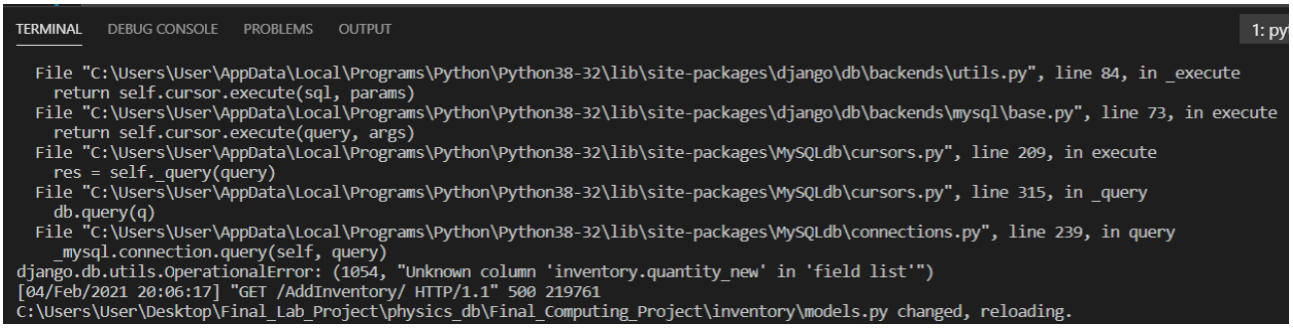
```python
class Inventory_Equipment(models.Model):
    #id = models.IntegerField(primary_key=True, null=False, blank=True)
    name = models.CharField(
        max_length=255,     #maximum length of input
        blank=True,         #the field can be blank
        null=True,          #the field can be null
        default="Name this Equipment")  # the defualt value if it is left blank
    total_quantity = models.IntegerField(
        null=True,
        blank=True)
    location = models.CharField(
        max_length=20,
        default='Physics Office',
        blank=True)
    img_reference = models.ImageField(
        null=True,
        blank=True,
        upload_to='images_uploaded/',   #the location of where to store uploaded images
                                        #currently the location is local device
        default='images_uploaded/default.jpg')  #the default image stored if no image is uploaded by user

    class Meta:
        db_table="inventory"    #name of the table in the database to which this model links to

    def __str__(self):
        return self.name    #when queryed, the 'name' is returned

    #add validation functions - validation is a step i will do after barebones of the program are made
```

15. The `views.py` file holds all the logic for all the pages rendered across my application. This file is crucial for exchanging data to and from the templates of the app.
16. The next task of this iteration was to add things to the inventory table. The user would enter name of the equipment, quantity of equipment to add, the room where the equipment will be located, and an image of the equipment. Based on the `models.py` file, the image will be optional and the user may or may not upload one. As shown above, this form will be divided into two parts: a part to add quantity to an existing piece of equipment and a part to add an entirely new piece of equipment. The html/css style code of this page is above and the page looks like this:



17. There were a **few challenges** faced during the development of this part of the application:

- **Challenge 1:** This was the part to divide the form into two sections.
  - **i.** To fix this I tried two things – firstly I added new attributes to the `models.py` file. This did not work because the `models.py` file is a representation of the database and because the database did not have the fields for adding to existing equipment I got an error.



  - **ii.** To fix this I had to delete the field entirely and migrate the changes again. I then tried to add the user entered value for adding stock of existing equipment manually.

iii. This was done by adding two optional fields to the `forms.py` file, in addition to all the fields that came from the `models.py` file. The forms.py file looked like this:

```python
from django import forms    # importing the forms module from django
from .models import Inventory_Equipment, Practical, Practical_Equipment_Needed
# importing all the models created in models.py       Find related code in Final_Computing_Project

class Add_Inventory_Form(forms.ModelForm):
    new_quantity = forms.IntegerField(# field to enter the qty needed to be added to existing equipment
        required = False,
        widget=forms.HiddenInput(),
        initial=0)
    existing_name = forms.CharField( # field to select the equipment to add qty to
        max_length=255,
        required = False)
    class Meta:
        model = Inventory_Equipment # name of the model that this form represents
        fields = "__all__" # means that all attributes of the model named above will have a field
```

iv. Doing so meant that the html template could render all fields of the models.py file and these two additional fields as well, thus fixing the issue.

v. To then complete this form, I had to write a function in the views.py file that would first check if the quantity in the top form was blank or not. If blank, it would add an entirely new record to the database else, it would add the entered quantity to the selected equipment's current quantity. Adding also required knowledge querying data, a terminal screenshot showing how to query, followed by the code for this function is below:

Terminal screenshot:

```
TypeError: QuerySet indices must be integers or slices, not str.
>>> f=I.objects.filter(name = 'test').values('total_quantity')
>>> print(f)
<QuerySet [{'total_quantity': 5}]>
>>> print(f[0])
{'total_quantity': 5}
>>> print(f[0]['total_quantity'])
5
>>>
```

Function code:

```
def add_new_to_inventory(request):
    equipment_name = Inventory_Equipment.objects.all() #gets all the equipment stored in the inventory table of the db
    if request.method == "POST": # checks if data is being SENT/POSTED to the template

        form = Add_Inventory_Form(request.POST, request.FILES) #holds all the information from our form.
        #When submit is clicked this gets a dictionary of all attributes and inputs, creates a new form with all values you entered.

        if form.is_valid(): #built-in menthod to verify the constraints in forms.py file

            if (form.cleaned_data.get("new_quantity") is not None): # checks if the field to ADD to existing qty is blank or not
                equipment_name_selected = form.cleaned_data.get("existing_name")
                print(equipment_name_selected)  #for debugging purposes
                exisitng_quantity = Inventory_Equipment.objects.filter(name = equipment_name_selected).values('total_quantity')[0]['total_quantity']
                print(exisitng_quantity)    #for debugging purposes
                new_quantity = exisitng_quantity + form.cleaned_data.get("new_quantity")    #adds the user entered qty to exisitng_qty
                Inventory_Equipment.objects.filter(name = equipment_name_selected).values('total_quantity').update(total_quantity = new_quantity)
                print (new_quantity)    #for debugging purposes
            #add else here and check
            else:
                form.save() # if the qty to add is blank, saves the botton part of the form, creating a new record in database
            return redirect('/ViewInventory')   #displays the new record in a table - this was added later
            # link to a page which shows the full table inventory
    else: #basically if method is GET
        form = Add_Inventory_Form()
    # below line contains the context being sent to the template AddToInventory.html
    return render(request, 'main/AddToInventory.html', {'form': form, 'equipment_names': equipment_name})
```

- **Challenge 2:** Images were not being uploaded as required.
  i. The fix for this issue was rather simple, it just required me to add the parameter `request.FILES` to the line which renders the form.

ITERATION 2 REVIEW:

What has been done?

Through this stage of the development, I have focused on three major things:

3. Converting static HTML/CSS pages into dynamic Django templates
4. Working on stakeholder feedback from the previous iteration:
   a. Adding new fields – COMPLETED.
   b. Stylistic uniformity throughout the applications – COMPLETED.
   c. Calendar changes based on user logged in – will complete when login is made.
5. Fully develop the functionality needed to add equipment to the inventory.

I was successful in converting all static pages to be compatible with Django and I did implement most of what the stakeholder wanted me to. I have also fully completed the functionality to allow the user to add equipment to the inventory of the app.

How was it tested?

To test the features, I essentially just checked if the 'dynamic' version of the website's templates worked as they did previously. So, essentially ensure that the buttons were functioning well and the navbar worked fine.

To then test the feature of adding items to the inventory – I had to check the Django admin page of the Model/table to see if the changes in the form had been posted to the backend of the application and stored in the database. Just repeating this test with a variety of inputs was the major testing undertaken. This included:

- Not uploading images
- Writing string and characters for the quantity
- Not selecting an equipment in the part of the form to add to existing equipment's quantity.
- Checking if the correct default location was inputted in the database i.e. "Physics Office", if the user did not select anything in the location dropdown.

Testing methods undertaken:

Dropdown to add to existing equipment displays all the equipment names, currently in the inventory	
Validation for new equipment name is working and textbox takes in new name	Not yet added validation
Textboxes for equipment quantity working	
All the buttons in the navbar work as intended	

Adding the equipment and quantity of the equipment is seen in the database	

Success Criteria and Stakeholder demands met:

Apart from completing the previously completed success criteria again – this time to suit the back end of the application – I completed the below task. The application could now add new equipment to the inventory.

7.	Add new stock	Select an equipment or type in a new equipment, add the quantity, and be able to view the change in the stock/inventory database.

Changes in the design section:

There were no major changes in the design section for this page. This iteration was focused on building MODULE 5 of the design section and the code does as the algorithm in design section is described to do. The only change to the user interface was the addition of fields that took the location of where an equipment would be located and a field for storing the image of an equipment – this was an important stakeholder requirement.

Summary of the whole Project as a prototype at this stage:

At this stage, the project has a fully functional – but static – user interface such that the user can navigate between the different modules (some not completed yet though!). The project also has a Django back end framework ready which can be easily expanded later as I progress on in the later stages to add more functionality. Furthermore, the application now has a fully functional database – with **only the one inventory table** currently – and is therefore, capable of storing data on a backend server. The application can also store equipment in the inventory now.

In the next iteration, I will proceed on to work on some stakeholder feedback and add the options to report losses and breakages and edit details of equipment in the inventory.

ITERATION 3 – CRUD INVENTORY ITEMS

The primary focus of this iteration was to build on the functionality of the inventory made in the previous iteration. The previous section adds new items to the inventory database, thus creating new records in the database table named 'Inventory'. This iteration's role is to the Read, Update, and Delete records that may have been created by the user.

1. The first goal was to be able to report losses and breakages. This meant that the total quantity of equipment would have to be reduced by the quantity the user asked for.
2. The steps to achieve this are as followed:
    - As shown previously this is the template for removing equipment.

- The code for the above template was connected to Django backend so that the dropdown menu could be dynamic and could display only things that are added in the inventory database. The main code for the body of the template is:

```
<body>
    {% block content %}
        <div class = "FormForReporting">
            <form action="/LossReport/" method="POST" autocomplete="off">
                {% csrf_token %}
                <label for='equipment_name'> Select Equipment</label>
                <!-- below line GETS the list of equipment stored in the database from the server/backend-->
                {{ forms.equipment_name }}
                    <select id="equipment-name" name="equipment_name" type="text" placeholder="Select Equipment">
                        <!-- Default heading/option that is disabled to guide the user -->
                        <option value='Select' disabled="true" selected>Select</option>
                        <!-- For loop to display each individual equipment as a selectable list option in the drop-down -->
                        {% for equipment_name in equipment_names %}
                            <option value="{{ equipment_name.name }}">{{ equipment_name.name }}</option>
                        {% endfor %}
                    </select>

                <p> No. of Losses {{ forms.quantity_to_remove }}
                    <input id="qty-loss" name="quantity_to_remove" type="text" placeholder="Enter Lost qty.">
                </p>

                <button name="save" type="submit" class= "#" id="Send-Request">Send Request</button>

            </form>
            <p>testing</p>
        </div>
    {% endblock %}
</body>
```

- To convert the textboxes into fields of a backend form, I had to copy the `Add_Inventory_Form(forms.ModelForm)` function in the `forms.py` and create a form called `Remove_Inventory_Form(forms.ModelForm)`. The code for this form is as follows:
- This was followed by me adding the code to actually remove the equipment. To do so, I created a function in the `views.py` file, called `def report_loss(request)`, because this file acts as the direct link between the front-end. The code was basically same as the code to add the equipment, but quantity was subtracted as opposed to adding. The code for the same is below:

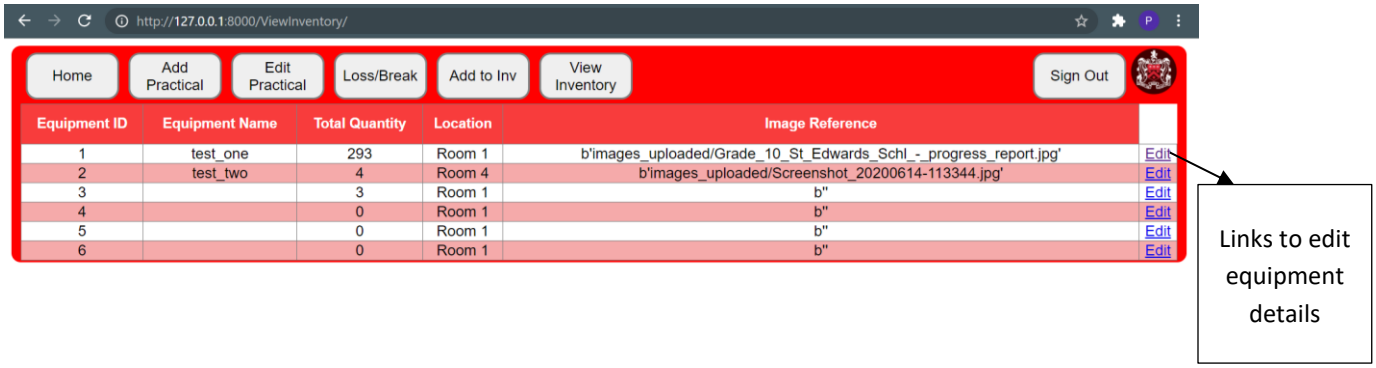
```python
#function to delete a certain quantity of the selected equipment_name
def report_loss(request):
    equipment_name = Inventory_Equipment.objects.all()  # querys all the data from the inventory table into a queryset - used to render the dropdown
    if request.method == "POST": #checks if data is being SENT to template
        form = Remove_Inventory_Form(request.POST) #holds all the information from our form.
        # When submit is clicked this gets a dictionary of all attributes and inputs, creates a new form with all values you entered.
        if form.is_valid(): #check if data entered in the form meets the constraints for forms.py file
            if (form.cleaned_data.get("quantity_to_remove") is not None): # use is not null instead
                equipment_name_selected = form.cleaned_data.get("equipment_name")
                # below line querys the quantity existing in the database
                exisitng_quantity = Inventory_Equipment.objects.filter(name = equipment_name_selected).values('total_quantity')[0]['total_quantity']
                new_quantity = exisitng_quantity - form.cleaned_data.get("quantity_to_remove")  #subtract the quantity
                Inventory_Equipment.objects.filter(name = equipment_name_selected).values('total_quantity').update(total_quantity = new_quantity)
            return redirect('/ViewInventory')
            # link to a page which shows the full table inventory
    else: #basically if method is GET
        form = Remove_Inventory_Form()
    return render(request, 'main/LossOrBreakReport.html', {'form': form, 'equipment_names': equipment_name})
```

3. The next part of this iteration was to create a page to be able to edit the details of an existing piece of equipment in the inventory. These details include name of equipment,

```python
#form to report loss of equipment from inventory
class Remove_Inventory_Form(forms.ModelForm):   #name of the form to remove equipment
    quantity_to_remove = forms.IntegerField(    #integer field for the quantity that is to be removed
        required = False,
        widget=forms.HiddenInput(),
        initial=0)
    equipment_name = forms.CharField(   #name of the equipment to be removed from - this is rendered as a dropdown
        max_length=255,
        required = False)
    class Meta:
        model = Inventory_Equipment # name of the model that this form is connected to
        fields = "__all__"  # means that all attributes of the model named above will have a field
```

its location, and the image stored under its name. To be able to edit the details of an equipment, during this iteration, I felt that it would be beneficial to have a page that allowed the user to view what they had in the inventory. **This was also a recommendation from the stakeholder at the end of the previous iteration and therefore, I thought that it made sense to add the option to edit equipment on this page.** To do so, I added a new tab on the navbar – and therefore, a new page to the application – which displayed the whole inventory as a table.

The page looks like this:



The HTML code for the displaying the table is as follows:

```
26                  <tbody>
27                      <!-- inventory_obj is the context passed from the template to refer to the table -->
28                      <!-- the lines below loops through the records of the table displaying needed information -->
29                      {% for equipment in inventory_obj %}
30                          <tr>
31                              <td class="table-blocks">{{ equipment.id }}</td>
32                              <td class="table-blocks">{{ equipment.name }}</td>
33                              <td class="table-blocks">{{ equipment.total_quantity }}</td>
34                              <td class="table-blocks">{{ equipment.location }}</td>
35                              <td class="table-blocks"><a href= "{{ MEDIA_URL }}{{ equipment.img_reference }}">IMAGE</a></td>
36                              <td class="table-blocks">
37                                  <a href="/EditInventory/{{ equipment.id }}" id="Edit-Page-Link">Edit</a>
38                              </td>
39                          </tr>
40                      {% endfor%}
41                  </tbody>
```

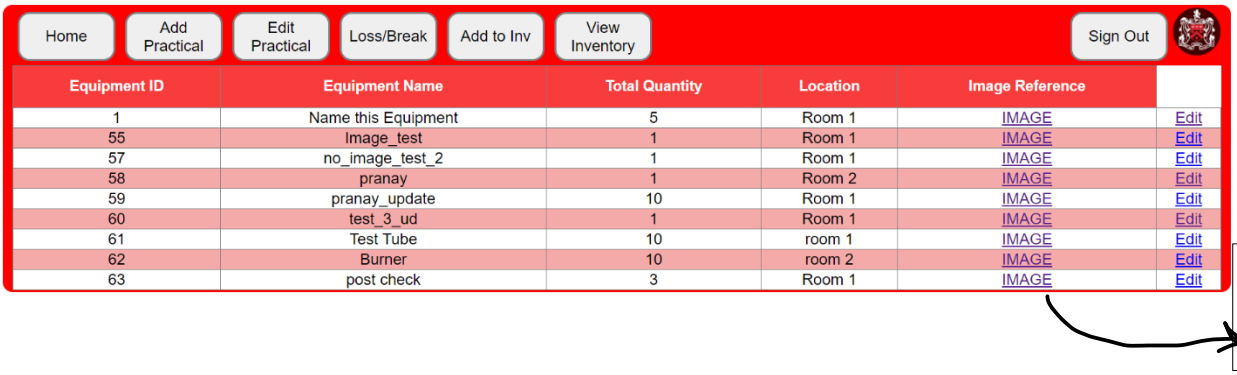
The styling is not so important, alternate rows were colored in a lighter shade of red for appeal purposes only.

The `views.py` function – named `def view_inventory(response)` - to render the information in the table is below:

```
57  # function to get all records from the inventory table and store them in a context dictionary
58  def view_inventory(response):
59      inventory_obj = Inventory_Equipment.objects.all()   #querys every single record into this identifier
60      return render(response,"main/ViewInventory.html", {'inventory_obj': inventory_obj}) #template rendered & inventory_obj passed as context to template
```

4. I faced a few **challenges** in achieving proper implementation for the table:
   - **Challenge 1: Viewing an uploaded Image:**
     i. The main challenge was being able to view the image in the form of a link.
     ii. The error in this case was the type of data used to store the reference of the image in the database table. **Initially** I was using the LONGBLOB datatype in a database to store this link. Doing so stored the image reference in the binary format as b”” as shown in the inventory table above.
     iii. After research, I converted this into a simple STRING type data. This meant that the reference could be accessed simply using the link as a URL.

iv. After making this change I had to change the URL rendering also and the inventory table page looked something like this:



Equipment ID	Equipment Name	Total Quantity	Location	Image Reference	
1	Name this Equipment	5	Room 1	IMAGE	Edit
55	Image_test	1	Room 1	IMAGE	Edit
57	no_image_test_2	1	Room 1	IMAGE	Edit
58	pranay	1	Room 2	IMAGE	Edit
59	pranay_update	10	Room 1	IMAGE	Edit
60	test_3_ud	1	Room 1	IMAGE	Edit
61	Test Tube	10	room 1	IMAGE	Edit
62	Burner	10	room 2	IMAGE	Edit
63	post check	3	Room 1	IMAGE	Edit

Image links fixed

- **Challenge 2: Image uploading:**
  i. This challenge was similar to the previous one. In this case, the image uploading functionality of the application would break entirely if the user did not upload an image. This was one of the most tedious errors so far!
  ii. There were two issues: the link/URL that displayed the image was broken and the program was not 'told' what to do if there was no image uploaded.
  iii. To fix the above, I added a '`default`' parameter in the `Invetory_Equipment` model of `models.py` file and chose a default image from the internet AND I changed the link/URL that the `ViewInventory.html` template led to when the 'IMAGE' link was clicked.

The change in the `models.py` file:

Default image code

```
20  class Inventory_Equipment(models.Model):
21      #id = models.IntegerField(primary_key=True, null=False, blank=True)
22      name = models.CharField(
23          max_length=255,      #maximum length of input
24          blank=True,          #the field can be blank
25          null=True,           #the field can be null
26          default="Name this Equipment")  # the defualt value if it is left blank
27      total_quantity = models.IntegerField(
28          null=True,
29          blank=True)
30      location = models.CharField(
31          max_length=20,
32          default='Physics Office',
33          blank=True)
34      img_reference = models.ImageField(
35          null=True,
36          blank=True,
37          upload_to='images_uploaded/',   #the location of where to store uploaded images
38                                          #currently the location is local device
39          default='images_uploaded/default.jpg')  #the default image stored if no image is uploaded by user
```

Image 'default.jpg':

5. After the image details and the view inventory page were completed, I moved on to develop the edit inventory page. This page was a bit more challeging. Each page had different details and to be able to edit them, those details had to be displayed in textboxes. To achieve the editing task I did the following:
   - Created two new functions in the views.py files: to display the information in editable textboxes and one to actually edit the details and update the database record respectively. The new functions in the `views.py` file to edit the inventory looked like this:

```python
# in both the functions below, id is the ID for the item in inventory to be edited

# function to get the record that needs to be edited and pass it to the template
def edit_inventory(request, id):
    inventory_item = Inventory_Equipment.objects.get(id = id) # get the item with the given id from the inventory table
    return render(request,'main/EditInventoryDetails.html', {'inventory_item': inventory_item})

# function to edit the exisiting item
def update(request, id):
    inventory_item = Inventory_Equipment.objects.get(id = id)  # get the item with the given id from the inventory table
    if (request.method == 'POST'):  # if data is being sent to the server/POSTED to server
        # below line renders the form, Add_Inventory_Form with data already filled in - the data is the instance
        form = Add_Inventory_Form(request.POST or None, request.FILES or None, instance = inventory_item)
        if form.is_valid(): #if the form meets validation criteria
            form.save() #save updated details to the database
        return redirect("/ViewInventory")   # once completed lead the inventory table page
    else: #basically if method is GET
        form = Remove_Inventory_Form()  #empty form if GETTING the page from database
    return render(request, 'main/EditInventoryDetails.html', {'inventory_obj': inventory_item })
```

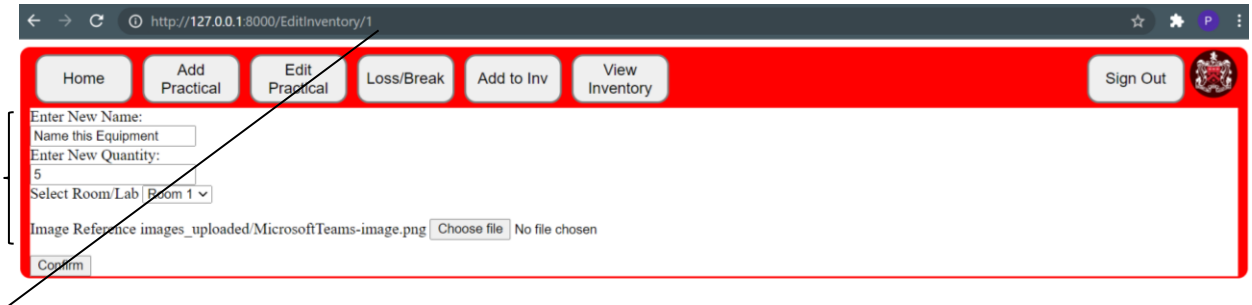
   - I also created a new html template for editing an inventory item's details – still need to add style though. The code for the template's BODY is below. In order to accelerate the development pace for the core elements of the application, I decided to add to this page later on.

```html
    <body>
        {% block content %}
            <form action="/Update/{{ inventory_item.id }}" method="post" class="form-group" enctype="multipart/form-data">
                {% csrf_token %}
                <label for="name">Enter New Name: </label><br> <!-- Input textbox to enter new name of item -->
                <input type="text" name="name" value= "{{ inventory_item.name }}"><br>

                <label for="total_quantity">Enter New Quantity: </label><br> <!-- Input textbox to enter new quantity of selected item-->
                <input type="text" name="total_quantity" value= "{{ inventory_item.total_quantity }}"><br>

                <label for='Room Number'> Select Room/Lab</label>   <!-- Input dropdown to select new location for selected equipment-->
                <!-- I will change the below OPTIONS when I make rooms table in the database - this will then be dynamically set -->
                    <select id="room-number" name="location" type="text" placeholder="Select Room/Lab" value="{{ inventory_item.location }}">
                            <option value='Select' disabled="true">Select</option>
                            <option value='Room 1'>Room 1</option>
                            <option value='Room 2'>Room 2</option>
                            <option value='Room 3'>Room 3</option>
                            <option value='Room 4'>Room 4</option>
                            <option value='Room 5'>Room 5</option>
                    </select>

                    <p> Image Reference <!-- Image field to select new image for the item selected -->
                        <input name="img_reference" type="file" placeholder="Enter Image for location" value="{{inventory_item.img_reference}}">
                    </p>

                <button name="save" type="submit" class="btn btn-success">Confirm</button>
            </form>
        {% endblock %}
```

- Updated the `urls.py` for this URL to be able to change based on the ID needed to be accessed.

```
urlpatterns = [
    path("", views.homepage, name="home"),
    #name must be same as the function name in the views.py.
    #The path('') in the urls.py of mysite looks for the same path in '' and then renders the page which follows that
    path("LossReport/", views.report_loss, name="loss_report_page"),
    path("AddInventory/", views.add_new_to_inventory, name="add_new_to_inventory"),
    path("AddPractical/<int:id>", views.add_new_practical, name="add_new_practical"),
    path("NameNewPractical/", views.name_new_practical, name="name_new_practical"),
    path("EditPractical/", views.edit_practical, name="edit_practical"),
    path("ViewInventory/", views.view_inventory, name="view_inventory"),

    # new urls added to be able to edit the inventory equipment - based on their ID
    path("EditInventory/<int:id>", views.edit_inventory, name="edit_inventory"),
    path("Update/<int:id>", views.update, name="update"),
]
```

- The edit inventory page – for EACH individual inventory item – finally looked like this:



Textboxes filled with details of the item with the selected ID

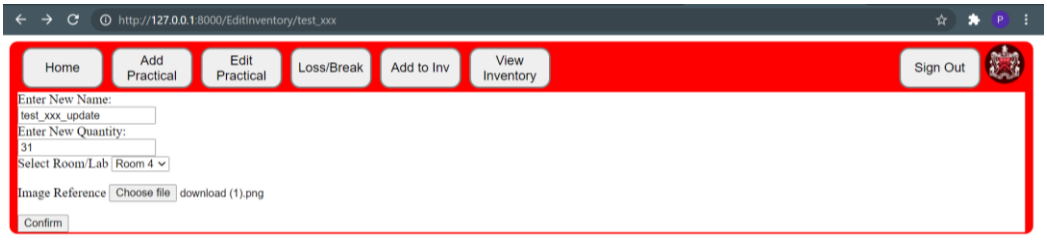
Details of item with this ID will be pre-filled

6. **Challenges faced:**

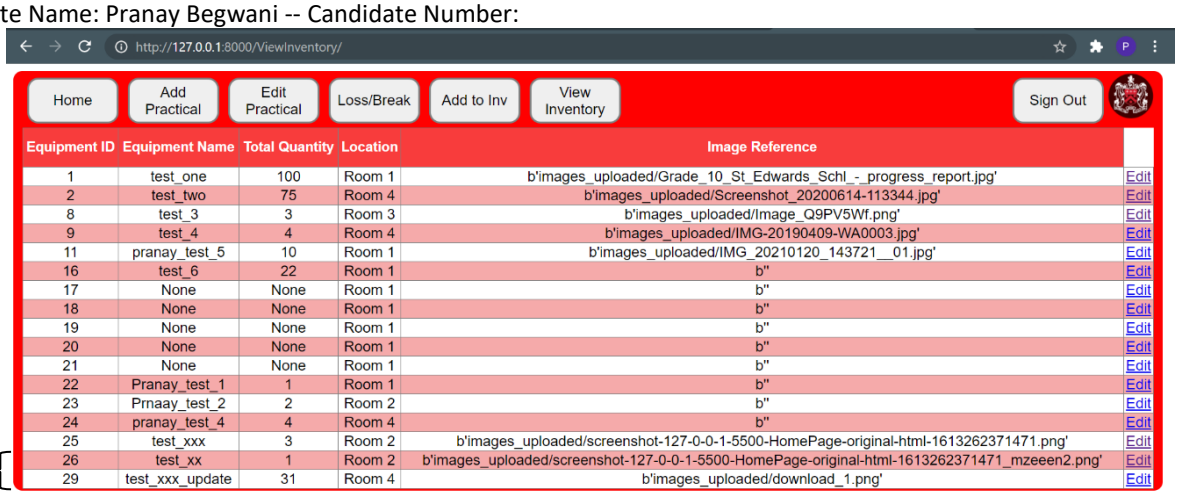
- **Challenge 1: A new record was being created everytime I edited an exisitng one:**

i. The reason I could think of when trying to fix this was the primary key of this table ID. Initially I had defined an ID for the table `Inventory_Equipment` in the MySQL server and in the `models.py` file. This meant that the ID – which was also the primary key was not unique and thus new records were created everytime. To fix this I removed ID attribute from `Inventory_Equipment` model in `models.py`. The error, class of this table, and the fixed program now looks like this.

Editing page for item 'test_xxx'



The inventory table after editing test_xxx:



Equipment ID	Equipment Name	Total Quantity	Location	Image Reference	
1	test_one	100	Room 1	b'images_uploaded/Grade_10_St_Edwards_Schl_-_progress_report.jpg'	Edit
2	test_two	75	Room 4	b'images_uploaded/Screenshot_20200614-113344.jpg'	Edit
8	test_3	3	Room 3	b'images_uploaded/Image_Q9PV5Wf.png'	Edit
9	test_4	4	Room 4	b'images_uploaded/IMG-20190409-WA0003.jpg'	Edit
11	pranay_test_5	10	Room 1	b'images_uploaded/IMG_20210120_143721__01.jpg'	Edit
16	test_6	22	Room 1	b''	Edit
17	None	None	Room 1	b''	Edit
18	None	None	Room 1	b''	Edit
19	None	None	Room 1	b''	Edit
20	None	None	Room 1	b''	Edit
21	None	None	Room 1	b''	Edit
22	Pranay_test_1	1	Room 1	b''	Edit
23	Prnaay_test_2	2	Room 2	b''	Edit
24	pranay_test_4	4	Room 4	b''	Edit
25	test_xxx	3	Room 2	b'images_uploaded/screenshot-127-0-0-1-5500-HomePage-original-html-1613262371471.png'	Edit
26	test_xx	1	Room 2	b'images_uploaded/screenshot-127-0-0-1-5500-HomePage-original-html-1613262371471_mzeeen2.png'	Edit
29	test_xxx_update	31	Room 4	b'images_uploaded/download_1.png'	Edit

A new record is added on editing

The **new and edited** `models.py` file is below:

```
# class to represent the INVENTORY table in the database       Find related code in Final_Computing_Projec
class Inventory_Equipment(models.Model):
    #id = models.IntegerField(primary_key=True, null=False, blank=True)
    # The above field now not in the code so that the records for each equipment are unique
    name = models.CharField(
        max_length=255,     #maximum length of input
        blank=True,         #the field can be blank
        null=True,          #the field can be null
        default="Name this Equipment")  # the defualt value if it is left blank
    total_quantity = models.IntegerField(
        null=True,
        blank=True)
    location = models.CharField(
        max_length=20,
        default='Physics Office',
        blank=True)
    img_reference = models.ImageField(
        null=True,
        blank=True,
        upload_to='images_uploaded/',   #the location of where to store uploaded images
                                        #currently the location is local device
        default='images_uploaded/default.jpg')  #the default image stored if no image is uploaded by user

    class Meta:
        db_table="inventory"    #name of the table in the database to which this model links to

    def __str__(self):
        return self.name    #when queryed, the 'name' is returned

    #add validation functions - validation is a step i will do after barebones of the program are made
```

7. The final job was to show whatever I had so far to the stakeholder, Mr Worth. I set up a Team's call with him to demonstrate the functionality of the application and his feedback is as follows:

Geoff Worth
Thu 11/03/2021 08:27
To: Pranay Begwani

Hi Pranay,

Great work on the computing project.

The inventory is really excellent. Clear, easy to use and will become a real resource that staff can use. It allows staff to enter all the equipment we own and looks really neat too.

Thanks,

Mr Worth

Looking at the above feedback, he appeared to be pretty pleased with how the application was developed thus far.

- Based on the discussions in the call he wanted me to add two things to application:
  i. Basic Validation for fields.
  ii. Ability to entirely delete an equipment's record from the inventory.
  iii. Style Uniformity of the application.
- In response to the above requests:
  i. I am planning on doing a separate iteration where I add validation to every field in the application.
  ii. I will add that!
  iii. I am planning to do a separate iteration where I enhance the application's stylistic elements.

ITERATION 3 REVIEW:

What has been done?

The main focus for this stage of development was to develop the functionality to report losses and breakages of equipment and edit the details of existing pieces of equipment.

To implement these, I used Django to render a form which allowed the user to perform the needed operation. This process also entailed me making three more dynamic HTML templates for the webpage of the application:

- View Inventory Page
- Edit Inventory Page
- Report Loss/Breakage Page

The user can now report losses of any equipment and the change is reflected in the application. And the application also allows the user to edit details – name, quantity, location, image reference – of any equipment selected to edit on the View Inventory Page table.

How was it tested?

To test the features, I essentially just checked if the 'dynamic' version of the website's templates worked as they did previously, ensured that the buttons were functioning well and the navbar worked fine.

To then test the feature of reporting losses and breakages of selected inventory items – I had to check the Django admin page of the Model/table to see if the changes in the form had been posted to the backend of the application and stored in the database. Just repeating this test with a variety of inputs was the major testing undertaken.

Furthermore, I also tested the editing feature of the application by undertaking repeated testing of the features. There were a few features that were not working as expected, but after research I was able to fix them. The major testing undertaken was:

- Not uploading images when editing items
- Writing string and characters for the edited quantity

Testing methods undertaken:

Dropdown for equipment name displays names of all the equipment in the database	
Textbox for quantity to remove works correctly	
Validation features work as expected	Validation not yet added

Buttons work correctly – cursor turns to pointer and text in button becomes red	
All the buttons in the navbar work as intended	
Database reflects the changes made when 'Send Request' clicked by user.	

Other tests added for new pages:

View inventory page shows details of all equipment added	
Link to view an image works perfectly	
Edit inventory item page displays current information in editable fields	
Edit Page fields have appropriate validations	Validation not yet added
New details saved in the database aptly	

Success Criteria and Stakeholder demands met:

Apart from checking that the previously completed success criteria worked, I completed the below tasks. The application could now remove a certain quantity of equipment from the inventory and edit the details of a selected item.

8.	Report a loss or breakage	This involves successfully selecting a piece of equipment, entering quantity of the broken equipment, clicking the button which then reduces the quantity of the equipment from the school stock/inventory, and this change should be visible in the database that the viewer views.
9.	Viewing the details of a selected equipment in the inventory	This involves viewing quantity, availability and bookings of an item selected by the user from the inventory.
10.	Edit details of selecting an item from inventory	Successfully editing the details of a selected item in the inventory. This includes the name, qty, location, and image reference.

Changes in the design section:

There were some changes from the design section for this page. This iteration was focused on building MODULE 6 from the design section and the code does as the algorithm in design section is described to do.

The modification in this stage from the design section were the addition of a page to view the inventory. This page shows the details of all the equipment. This page also gives the ability to edit details of equipment selected. A further change and a rather major and essential one was

the addition of the functionality to be able to edit details of existing equipment. This was not something that struck me during design but seemed essential as development progressed.

Summary of the whole Project as a prototype at this stage:

At this stage, the project, the application now has a fully functional database – with **only the one inventory table even right now** – and is therefore, capable of storing data on a backend server. The application can store equipment, remove quantity of equipment, and edit the details of equipment selected.

In the next iteration, I will proceed on to work on some stakeholder feedback and add the functionality to add new practical to the database. This will allow the user to store equipment lists under the name of a practical.

ITERATION 4 – NEW PRACTICAL ADD & EDIT

Now that the majority of the inventory was complete, I started the development of the option to add new practicals. The role of this part of the development is to allow the user to store lists of different types of equipment under the name of a practical. To do so, they must first create a new practical. After this they can selected varying quantities of each type of equipment (already added in the inventory) to stores details of practicals.

1. To start this stage I built the backend database tables to be able to store the practical and link the equipment to it. This required a many-to-many relationship because many practicals can use many items of equipment. To achieve a normalized and consistent many to many relationships, I needed a link table that connected the two tables together. The entity relationship diagram for this case looked like this:

The amendment made to the `models.py` file for creating the tables to store all practical and their respective equipment looked like this:

```
50
51  # Table to store list of all the practical - just their names
52  class Practical(models.Model):
53      practical_name = models.CharField(
54          max_length=100,
55          blank=True)
56      equipment_needed = models.ManyToManyField(  # this is used to link to the link table to normalize the many to many relationship
57          'Inventory_Equipment', # name of the table to which this has a many to many relationship with
58          blank=True,
59          through='Practical_Equipment_Needed')   # name of the link table to normalize the many to many relationship
60
61      class Meta:
62          db_table="practical"
63
64      def __str__(self):
65          return self.practical_name
66
67  # Table to store the equipment needed in every practical and their quantities
68  class Practical_Equipment_Needed(models.Model):
69      equipment_needed = models.ForeignKey(   # a foreign key of the inventory table to store the equipment
70          'Inventory_Equipment', # table of which this is a foreign key
71          on_delete=models.CASCADE)   # needed to reflect any deletions all across the database
72      practical = models.ForeignKey(  #foreign key of the Practical table
73          'Practical',
74          on_delete=models.CASCADE)
75      equipment_quantity = models.IntegerField(   # integer field needed to store the quantity of an equipment
76          default=0)  # the quantity if no number is entered by the user
77
78      class Meta:
79          db_table="practical_equipment_needed"
80
81      def __str__(self):
82          return self.practical.practical_name
```

2. The database tables for the above newly defined models looked like this:

Practical table:

Table structure | Relation view

#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
1	id	int(11)			No	*None*		AUTO_INCREMENT	Change Drop More
2	practical_name	varchar(100)	utf8mb4_general_ci		No	*None*			Change Drop More

Link table to normalize the relationship:

Table structure | Relation view

#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
1	id	int(11)			No	None		AUTO_INCREMENT	Change Drop More
2	equipment_quantity	int(11)			No	None			Change Drop More
3	equipment_needed_id	int(11)			No	None			Change Drop More
4	practical_id	int(11)			No	None			Change Drop More

The Names above essentially refer to column headings of the table and the other attributes are as they say.

- The id – this is a primary key used to uniquely identify each record of this table. Each record of this table consists of a practical id, equipment id, and equipment quantity. These refer to what quantity of each equipment is needed for a particular practical.
- Equipment_quantity – this column stores the quantity of each equipment required to do a practical
- Equipment_needed_id – this is a foreign key of the Inventory table used to store the id of an equipment needed (from the Inventory table) needed in the practical, as added by the user
- Practical_id – another foreign key, this time of the Practical table. It is used to store the id to refer to the practical which needs an equipment.

None of these above columns can be null, as shown in the table above. The primary key i.e. ID has an extra attribute called AUTO_INCREMENT. This is useful in keeping the ID unique for every piece of equipment added to a practical.

3. The next stage was to create a form in the `forms.py` file which allowed the user to create a practical and store equipment under its name. This was perhaps the most challenging part of the application: to figure out how to create a form which fed data to a many-to-many relationship table in the database. Normally, it is easier because a **'through'** table is not defined explicitly but this was necessary for me to be able to store quantities of equipment. Therefore, making the appropriate form was difficult. Rendering a form was particularly difficult because, in this instance, it also involved rendering a varying number of fields depending on how many time user clicked the **'+'** button.

   To try and achieve this I tried to use what is known as an inline formset in Django. The whole idea of which being to render multiple copies or instances of the same form. To make this I wrote this code in the `forms.py` file:

```
47  Add_Practical_Formset = inlineformset_factory(  #creating the formset to input details of practical
48      Practical,  # the parent model for the form
49      Practical.equipment_needed.through, # 'through' reference/ link model for the form
50      fields=['equipment_needed','equipment_quantity'],   #the fields needed in  the form
51  )
```

Although the forms were rendering fine, they were not saving any data to the database table, therefore, something clearly was not working. I then tried to generate a single model form and loop it again and again to be able to repeatedly store data. But this was too complex to achieve in the given timeframe and was extremely time consuming.

Finally, I resorted to use **Model Formsets,** which are also a library of Django used to render multiple instances of a form. The key difference between inline formsets and model formsets is the fact that model formsets directly save data into the specified model. This was implemented in the `forms.py` file like this:

```
# form to input a single equipment and its quantity for a practical
class New_Practical_Detail_Form(forms.ModelForm):   # declaring the form
    class Meta:
        model = Practical_Equipment_Needed  # the model that this form is connected to
        fields = ('equipment_needed', 'equipment_quantity', )   # fields needed in the form

# formset which creates multiple instances of the form above
Add_Practical_Formset = modelformset_factory(   # declaring the formset
    Practical_Equipment_Needed, #the models that this formset is connected to
    New_Practical_Detail_Form   # the form (above) whose copies this formset creates
)
```

4. The next part was to create the template which allowed the user to create a new practical.
    - To do so, I first opened the html page that I had left blank for this and then developed over there. Because I was unsure about how to render the form, I let Django do all the work and generate a simple formset without any styling at all. The HTML code for the body of the page, `AddNewPractical.html` looked like this:

```
<body>
    {% block content %}
        <div class = "FormForAdding">
            <!-- Declaring the form to add details -->
            <form action="#" method="POST" autocomplete="on" enctype="multipart/form-data">
                {% csrf_token %}

                {{ formset.as_p }} <!-- Django template language to allow django render simple formset-->

                <button name="save" type="submit" class= "btn btn-success" id="Send-Request">Send Request</button>

            </form>
            <p>testing</p>
        </div>
    {% endblock %}
</body>
```

    I did not add any styling to this form and therefore, the body code is so slim. To progress as much as possible on the core of the application, I will add styling at the end.

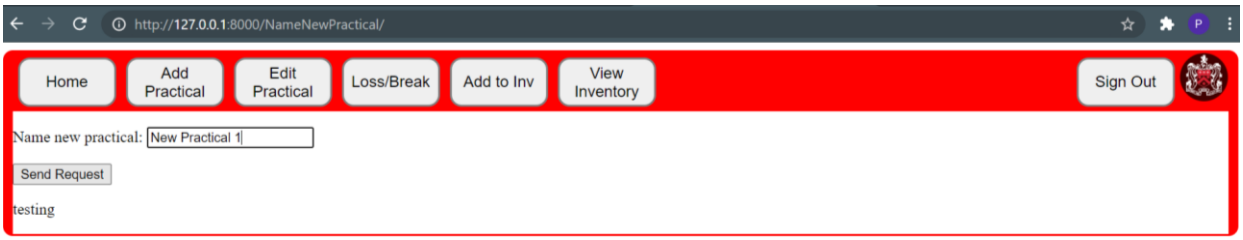
5. The next part of this iteration was to create a `views.py` file for this section of the application. The new function to add a new practical to this file was crucial to connect with the front end of the form. It is this function which renders the formset on the HTML

template, take input from the form and processes it and stores the user entered data in the database.

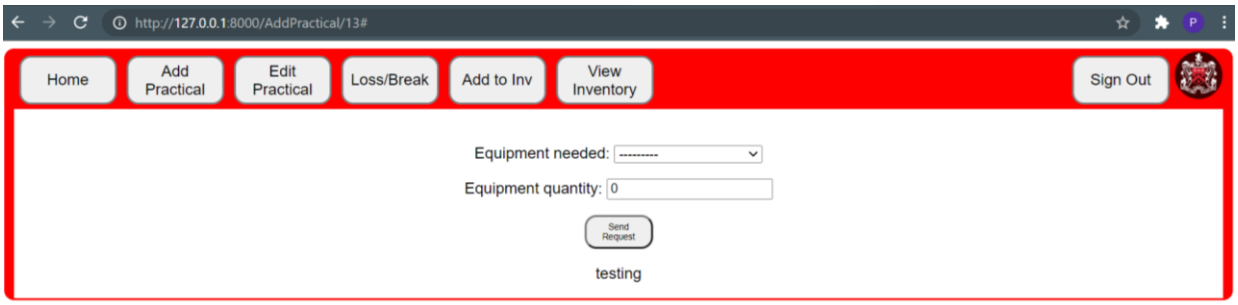
- **Challenges faced** in developing functions for the `views.py` file:
  i. The hardest part of this stage of the application was to render totally different python forms on the same page. In this case, these forms were the form to input name of a new practical and the formset that the user used to enter details of the newly added practical.
  ii. The simplest fix for this problem, was the addition of a new HTML template that took input of the name of the new practical. This new page – without any style – looked like this in my website:

This page appears when 'Add Practical' is clicked.

The name of this page is 'NameNewPractical' and the moment user the clicks 'Send Request' after typing name of a new practical, this leads to the add practical page for that new practical.



Clicking send request in the page above leads to the page below where user can select equipment from the dropdown and enter their quantity.



This page is still lacking styling features and I will add them towards the end. The functions for these two pages in the `views.py` file are below:

```python
# function to create new empty practical       Find related code in Final_Computing_Project
def name_new_practical(request):
    practical = Practical.objects.all()

    if (request.method == 'POST'):  #if connection method is POST
        form = New_Practical_Form(request.POST) #creates new form to send data to server
        if form.is_valid():
            new_name = form.cleaned_data.get('name_new_practical')  #gets name of the new practical in the variable
        new_practical = Practical.objects.create(practical_name = new_name)    # creates new practicalin the database
        new_id = Practical.objects.filter(practical_name= new_name).values('id')[0]['id']   #gets id of the newly added practical

        print (str(new_id))
        practical = Practical.objects.get(pk = new_id)  # querys the new practical using its ID to pass to template
        return redirect('/AddPractical/%d'%new_id)  #once all tasks complete, leads to the AddPractical page to enter details of new practical
    else:
        form = New_Practical_Form() # if the method is GET, displays the empty form
    return render(request, 'main/NewPractical.html', {'form': form, 'practical': practical})

# function to add details of newly added practical to database
def add_new_practical(request, id): # ID of the newly added practical is a parameter so that equipment is added for that practical only

    if (Practical_Equipment_Needed.objects.filter(practical_id = id).count()) == 0: # checks if the practical already exists in the database
        new_practical_details = Practical_Equipment_Needed()    # if it doesn't exist, creates empty object for details
    else:
        print ('Exists')    # prints exists if practical already exists
        # redirect to edit practical page

    if (request.method == 'POST'):  # if method is POST
        formset = Add_Practical_Formset(request.POST)   # creates new empty formset to add details for the selected practical
        if formset.is_valid():
            instances = formset.save(commit=False)  # saves data but doesn't send it to the database
            for instance in instances:  # querys through every individual instance of the formset
                instance.practical_id  = id
                instance.save() #saves individual instance to the Practical_Equipment_Needed database table
            return redirect('/AddPractical/%d'%id) #return to the same page after save to be able to add more equipment

    formset = Add_Practical_Formset(queryset = Practical_Equipment_Needed.objects.filter(practical__id = id))   # empty formset

    return render(request, 'main/AddNewPractical.html', {'formset': formset})
```

6. The next role in this iteration was for me to create URLs for these pages. Although I created them before the `views.py` page, it was crucial to explain how the views.py file worked before this. Below is a snippet of the `urls.py` file for the inventory app:

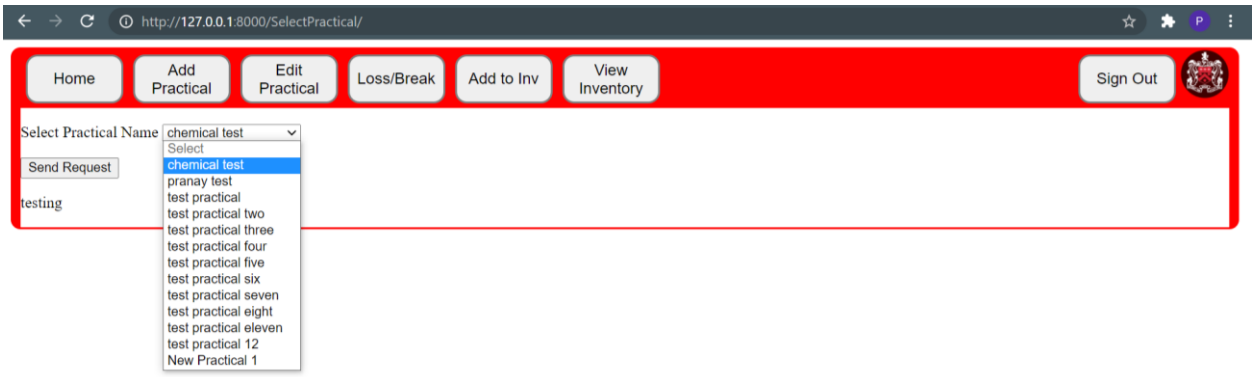
```python
    # below is link to the page that helps adding new equipment to the inventory
    path("AddPractical/<int:id>", views.add_new_practical, name="add_new_practical"),
    # below is link to the page that allows user to create a new practical. This page leads to the above url
    path("NameNewPractical/", views.name_new_practical, name="name_new_practical"),
```

8. After researching further, to create the functionality to develop the part of the application that allowed user to edit practicals, I decided to develop this part in the same iteration, given the similarity in procedure.
    - Firstly, I had to create two templates to accommodate two different forms:
        i. Form to select the practical whose details the user wanted to edit.
        ii. Formset that actually allowed the user to edit the practical.

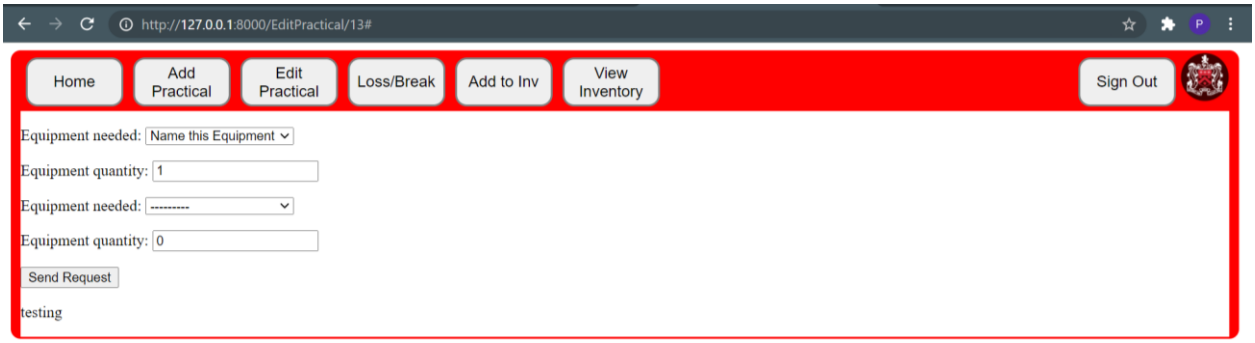
    The above forms only required me to add one new form to the forms.py file: a form to allow the user to select the name of the practical that they want to edit. I called this form `Select_Practical_Form()`. The formset that enabled the user to edit the practical was the same as the formset that was used to add details – `Add_Practical_Formset()`. Below is a snippet of the forms.py file which shows the form needed to select practical:

```
#Form to select the practical that the user wants to edit
class Select_Practical_Form(forms.Form):
    name_practical = forms.CharField(max_length=255)
```

- Next part was to create the two templates using HTML – again, I kept the templates confined to the form only and did not add any styling at this stage. The page where the user selected a practical then led to the page with the existing details of the selected practical to edit those details. I named these templates `SelectPracticalToEdit.html` and `EditPractical.html` respectively. Below are screenshots of the two templates:
  Page to select the name of a practical (`SelectPracticalToEdit.html`):



  Page that appears after selecting a practical (`EditPractical.html`):
  In this case, I have selected 'New Practical 1' – the formset displays all the equipment and their quantities I added in this practical (when adding it) inside editable dropdown and textboxes respectively.



- After this, I created the URLs for these pages in the `urls.py` file. The new additions to the urls.py file are below:

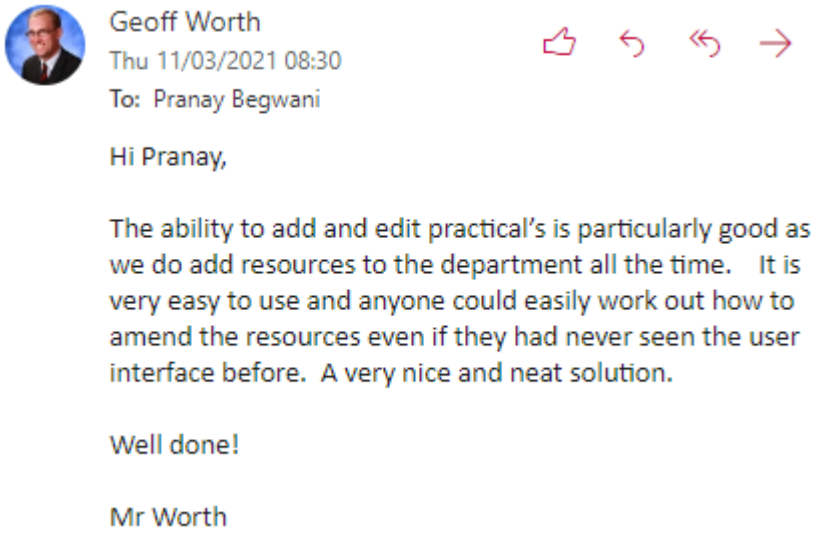
```
    # the url which lead to the page needed to select a practical to edit
    path("SelectPractical/", views.select_practical_to_edit, name="select_practical_to_edit"),
    # the url which leads to the page to edit details of the practical with id - id - in the Practical table
    path("EditPractical/<int:id>", views.edit_practical, name="edit_practical"),
```

- The final part of this iteration was to write the functions for both jobs – select a practical and edit its details – in the `views.py` file. The two functions are below:

```python
# function to enable user to choose the practical whose details they want to edit
def select_practical_to_edit(request):
    practical_names = Practical.objects.all()   # storing objects of all practical to render names in the dropdown on webpage

    if (request.method == 'POST'):  # if POSTING data to the backend
        form = Select_Practical_Form(request.POST)  # rendering the Select_Practical_Form needed to take input of practical name
        if form.is_valid(): # checking if form meets validation constraints
            selected_practical_name = form.cleaned_data.get('name_practical')   # storing the name selected by user in a variable

        id_selected  =  Practical.objects.filter(practical_name= selected_practical_name).values('id')[0]['id'] # getting id of the selected practical
        return redirect('/EditPractical/%d'%id_selected)    # redirecting user to the page needed to edit the details of the selected practical

    else:   # if GETTING data/information
        form = Select_Practical_Form()  # displays an empty form

    return render(request,'main/SelectPracticalToEdit.html', {'form': form, 'practical_names': practical_names})    #renders webpage and sends data to template

# function to edit details of practical selected using select_practical_to_add
def edit_practical(request, id):
    practical_names = Practical.objects.all()   # storing all practical objects in a queryset

    if (request.method == 'POST'):
        formset = Add_Practical_Formset(request.POST)   # renders the formset to allow user to edit practical selected
        if formset.is_valid():
            instances = formset.save(commit=False)  # commit is false so that the user can make multiple additions at the same time
            for instance in instances:  # cycle through all instances and save them individually to database
                instance.practical_id = id  # making sure instance is saved for the practical with th correct id
                instance.save()
        return redirect('/EditPractical/%d'%id) # leads to the same page to allow user to add/edit more details

     # displaying formset with existing details of practical
    formset = Add_Practical_Formset(queryset = Practical_Equipment_Needed.objects.filter(practical__id = id))

    return render(request, 'main/EditPractical.html', {'formset': formset, 'practical_names': practical_names})
```

- Given the similarity between editing and adding practicals there were no major challenges that I faced. The only key difference between the two parts was the addition of a ‘queryset’ parameter in the formset of the editing function. This basically ‘pre-fills’ a form with data queryed from the database table.

9. The final job was to show whatever I had so far to the stakeholder, Mr Worth. I set up a Team’s call with him to demonstrate the functionality of the application and his feedback is as follows:



Geoff Worth
Thu 11/03/2021 08:30
To: Pranay Begwani

Hi Pranay,

The ability to add and edit practical’s is particularly good as we do add resources to the department all the time. It is very easy to use and anyone could easily work out how to amend the resources even if they had never seen the user interface before. A very nice and neat solution.

Well done!

Mr Worth

Looking at the above feedback, he appeared to be pretty pleased with how the application was developed thus far.

There were no other request from the stakeholder and they seemed satisfied with the solution’s development so far.

ITERATION 4 REVIEW:

What has been done?

The primary focus for this stage of development was to develop the functionality that allows the user to add a new practical and edit an existing practical. There were a few stages to both tasks. The first being to figure out a suitable format for the backend database. The next being to find an appropriate Django form construct needed to make the form which allowed the user to enter a varying number of equipment and their quantities for each practical.

After this it was a matter of leading the user into entering the name for a practical and lead them to then enter the details of that practical: the equipment needed and quantity of the equipment. This was made simple using Django formset which allowed me to create multiple instances/copies of the same form and save it again and again. The process of editing a practical's details was very similar and utilized formsets again – this time rendering the existing details into fields/empty fields.

This process also entailed me making 4 more dynamic HTML templates for the webpage of the application:

- Naming a new practical
- Enter details for new practical
- Select Practical to edit
- Edit selected practical

The user can now enter a new practical and store equipment (existing in the inventory) under its name. They can also, edit details of this newly added practical.

How was it tested?

To test the features, I essentially just checked if the 'dynamic' version of the website's templates worked as they did previously, ensured that the buttons and navbar were functioning as before.

To then test the feature of adding practicals and editing them – I had to check the Django admin page of the 'Practical_Equipment_Needed' model to see if the changes in the form had been posted to the backend of the application and stored in the database. Just repeating this test with a variety of inputs was the major testing undertaken.

Below is summary of testing undertaken for both functionalities:

Take input of name of practical	

Textbox for practical name displays prompts if validation is failed	No validation added yet
Checking if new practical is already in the database and prompting user if it is	No validation added yet
Lead to page to enter details of practical after button click	
Display dropdown of equipment and textbox for quantities of the equipment	
Display a textbox and dropdown pair every time plus sign is clicked	
Dropdown for practical name displays names of all the practical in the database	
Clicking Edit displays all the current equipment and quantities of the chosen practical in dropdowns and textboxes respectively on a new page	
Clicking the 'dustbin' deletes the respective equipment from the database	
Display an empty/default textbox and dropdown pair every time plus sign is clicked	
Dropdowns of equipment contains all the equipment added in the inventory	
Successfully delete the selected practical from the database when delete is clicked by the user	
All the buttons in the navbar work as intended	
Database reflects the changes made when 'Send Request' clicked by user.	

Success Criteria and Stakeholder demands met:

Apart from checking that the previously completed success criteria worked, I completed the below tasks. The application could now add a new practical to the database, allow the user to store equipment for that practical, and edit the details (equipment and/or their quantities) for practical.

4.	Ability successfully to add a practical experiment	Successfully adding a practical to the database. This will involve selecting a piece of equipment/creating a new equipment, selecting quantity of this equipment, confirming the addition, and being able to view this change in the database.
5.	Ability to amend an existing practical	This involves, successfully entering the page to make the change, select the name of the practical, make the needed change – this could be the name, equipment, or the quantity – save it and view it in the database table.

Changes in the design section:

There were some changes from the design section for this page. This iteration was focused on building MODULES 2, 3 and 4 mainly from the design section and the code does as the algorithm in design section is described to do.

The modification in this stage from the design section were the addition of separate pages to name a new practical (when adding) and select one (when editing). This was simply because of the ease of development, the benefits of building multiple forms on the same HTML template were far outweighed by the time saved by simply adding new pages to the website. These pages served as intermediaries to lead the user into storing equipment information and editing it respectively.

Summary of the whole Project as a prototype at this stage:

At this stage, the project, the application now has a fully functional database. This iteration allowed me to considerably expand the database, which now stores equipment in the inventory, practical, and a table which stores the number of equipment needed in each practical.

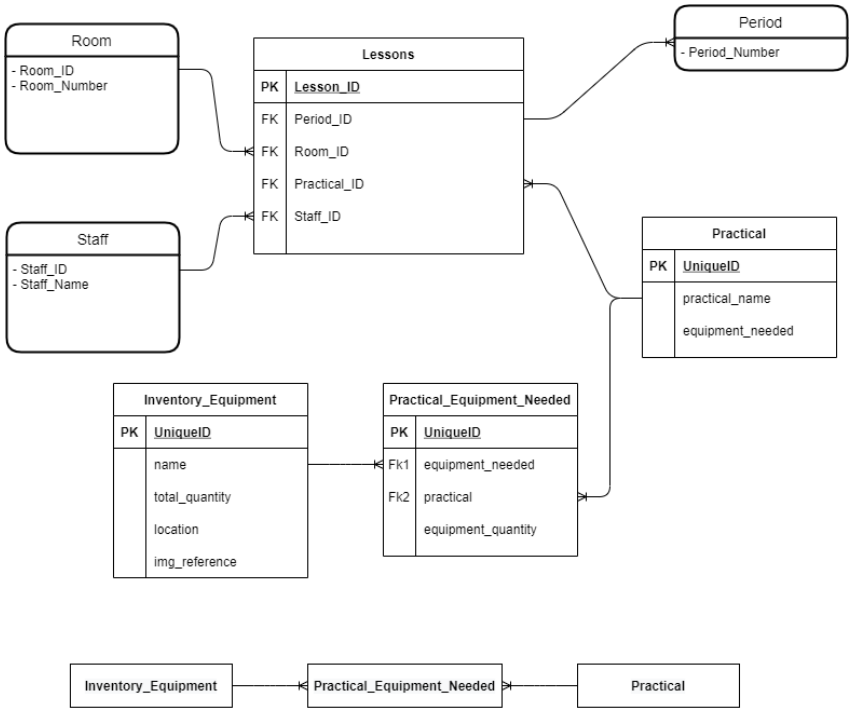
In the next iteration, I will proceed on to work on some stakeholder feedback and allow the user to book a particular practical for a certain number of students. This booking will be showed in something like a log.

ITERATION 5 – BOOKING PROCESS START

The main aim of this iteration was to start the development of the booking system of the application. This process had several layers involved in the development stage:

- Making the homepage form functional.
- Creating a form to take booking details.
- Create dictionary to store booking details as entered by the user.
- Create more tables in the backend of the database such that the database stored information like room number, teacher names, lesson time, etc.
- Check for booking clashes.
- Calculate total equipment needed in a practical – using equipment needed x no. of students.
- Create a message with the list of all equipment.
- Send booking details to technician in an email if the booking is successful.

1. The first job in this iteration was to form the backend network of tables in the database. This database entity relationship diagram for the database looked as planned in the design section:



The code to create the above database in the `models.py` file is as follows:

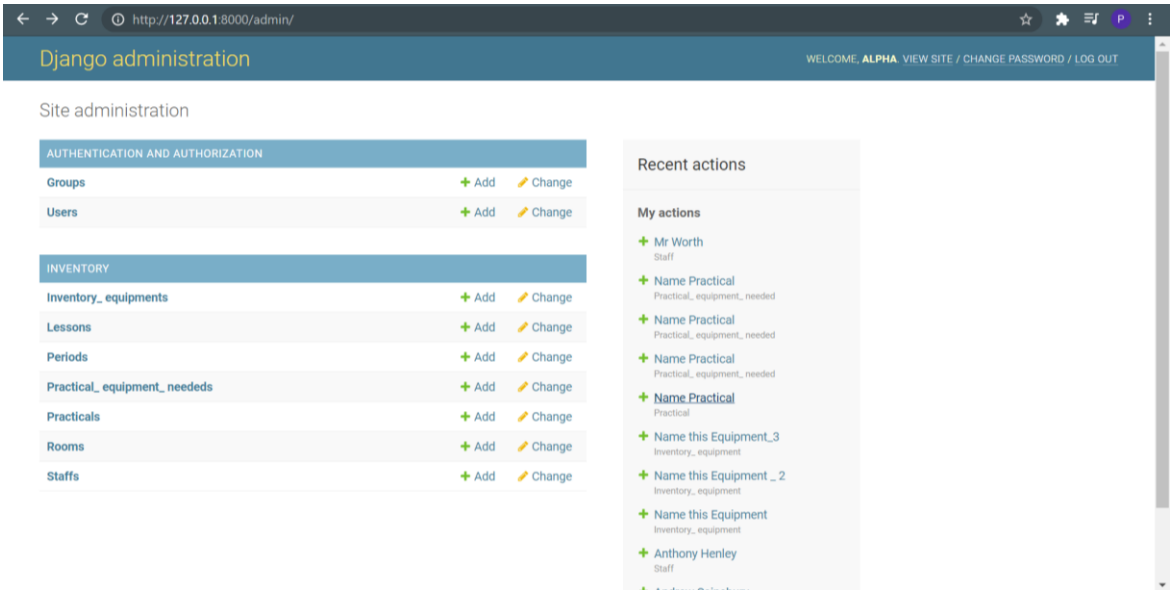
```python
# Model for table with names of all staff members
class Staff(models.Model):
    staff_name = models.CharField(max_length=255)# something like a dropdown of the staff names from the database

    class Meta:
        db_table="Staff"

    def __str__(self):
        return self.staff_name

# Model for the table of all the rooms in the school science dept.
class Room(models.Model):
    room_name = models.CharField(max_length=30)#something like a dropdown of all rooms from the database...

    class Meta:
        db_table="Room"

    def __str__(self):
        return self.room_name

# Model for the lesson time (1 - 5)
class Period(models.Model):
    PERIOD_NUMBER_CHOICES = [
            ('Lesson 1','Lesson 1'),
            ('Lesson 2','Lesson 2'),
            ('Lesson 3','Lesson 3'),
            ('Lesson 4','Lesson 4'),
            ('Lesson 5','Lesson 5'),
        ]

# choice field because I need this to be a dropdown which holds the above 5 ooptions - it is a constant
    period_number = models.CharField(
                                choices=PERIOD_NUMBER_CHOICES,
                                max_length = 10)

    class Meta:
        db_table = 'period_number'

    def __str__(self):
        return self.period_number
```

```python
# Model which stores the booking details if a booking is made
class Lesson(models.Model):
    staff = models.ForeignKey(Staff, on_delete=models.CASCADE, default= 1, null=True, blank=True)
    # name of staff who is booking a lesson - foreign key of the staff model
    period_time = models.ForeignKey(Period, on_delete=models.CASCADE)
    # period number of the booking - foreign key of the Period model
    date = models.DateField()
    # date of the booking - needed to check for clashes in bookings
    practical_booking = models.ForeignKey(Practical, on_delete=models.CASCADE)
    # Foreign Key for Practical model - needed to store the name of the practical that the user wants to book
    room = models.ForeignKey(Room, on_delete=models.CASCADE)
    # room number that the member of staff wants to book - foreign key of the Room model
    number_students = models.PositiveIntegerField(default = 0)
    # number of students that will be doing the practical

    class Meta:
        db_table = 'Lesson_Bookings'
```

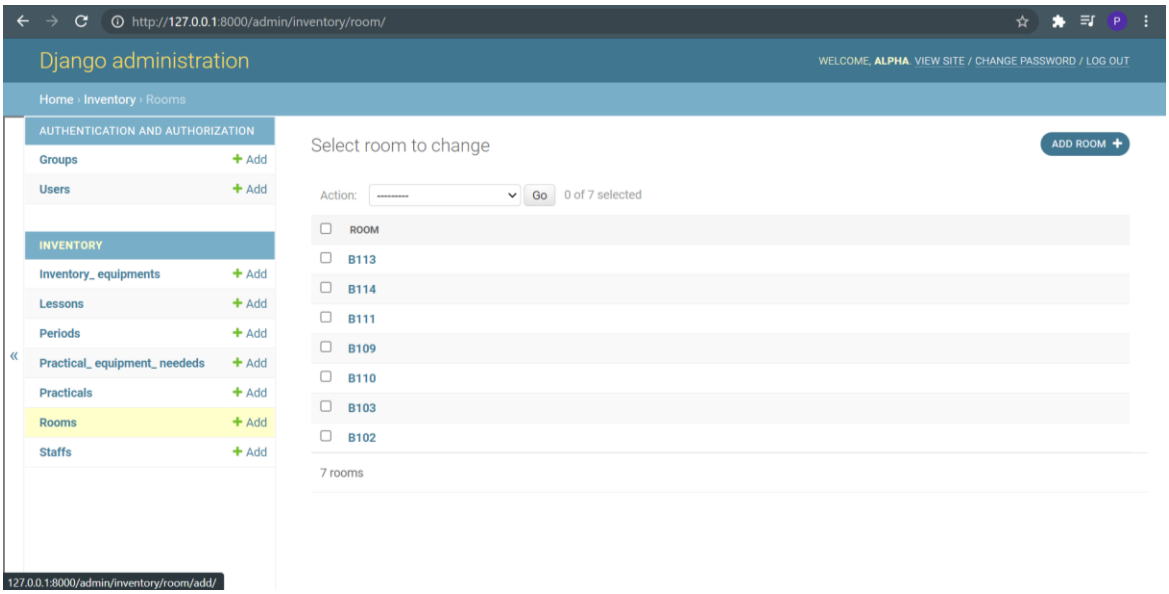
2. The user will have access to the Django admin of the website. This meant that I did not think it was worthwhile to add separate pages in the website which allowed the user to add more staff and rooms etc. Details like **Lesson Numbers**, Room Numbers, Staff

members were likely to be constant most of the time because the school adding new Science rooms, new teachers all the time is unlikely. The school timetable changing the number of lessons is also unlikely and hence I encoded the number of lessons in the code to save time. The Django admin looks as shown below and the forms to add new teachers and room numbers are shown below as well:

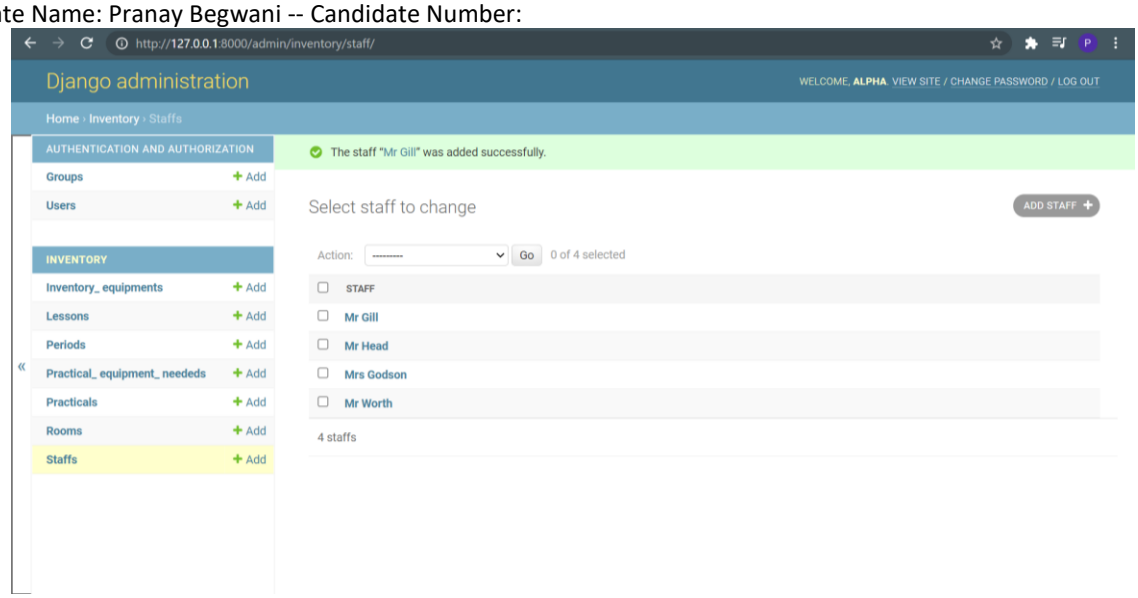
Admin page with access to all database tables:



Page to add new rooms to the database:



Page to add new staff:



All of the additions to the admin were reflected in the SQL database table and these were then also shown clearly in the dropdown menus on the website booking page.

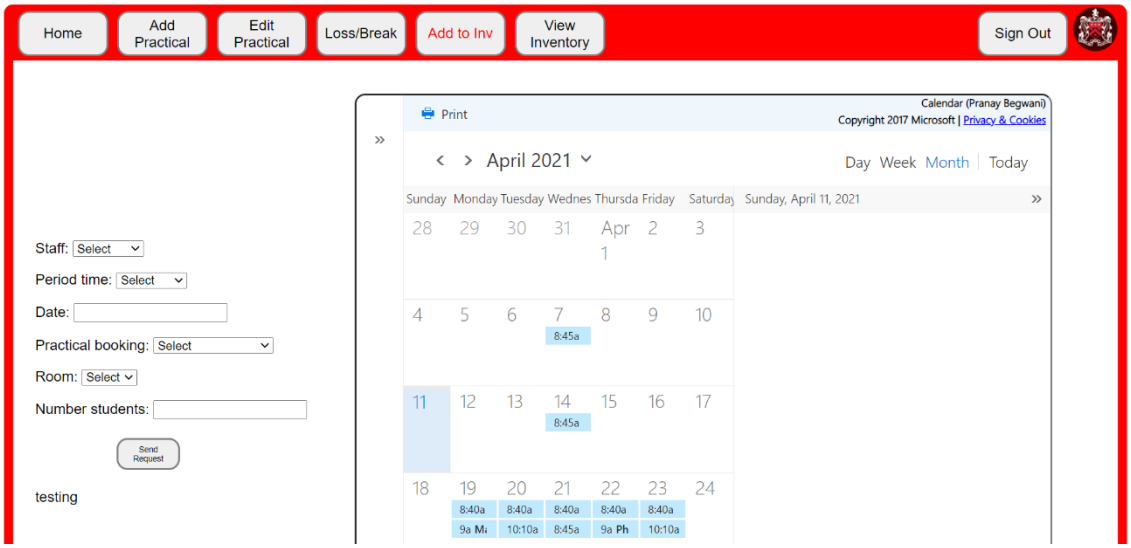
3. Following the creation of the models, I created a `forms.py` file needed to make a booking. The inputs of this form would mirror the `Lesson` model in the `models.py` file. This is the form which would be rendered on the HTML template of the Homepage and would be used by the user to make bookings of practicals. The inputs needed by this form would be the name of the staff booking the practical, lesson number being booked, date of when the lesson was to be conducted, practical being performed during the lesson, room number that is to be booked, and the number of students the room is to be booked for. The `forms.py` module for this form is as follows:

```
# Form to book a lesson in a particular room for some number of students
class Book_Lesson_Form(forms.Form):
    staff = forms.ModelChoiceField(queryset=Staff.objects.all(), empty_label='Select')
    # dropdown to select staff name
    period_time = forms.ModelChoiceField(queryset=Period.objects.all(), empty_label='Select')
    # dropdown to select choice of lesson
    date = forms.DateField()
    # date field that display a calendar allowing the user to select a specific date
    practical_booking = forms.ModelChoiceField(queryset=Practical.objects.all(), empty_label='Select')
    # dropdown to select what practical to boo for
    room = forms.ModelChoiceField(queryset=Room.objects.all(), empty_label='Select')
    # dropdown to select room number to book
    number_students = forms.IntegerField()
```

4. The next part of this stage was to render the above form in the html template of the homepage. I did not style this section initially and therefore, the rendering process is relatively simple. The form tag for `homepage.html` file looks like this:

```
<body>
    {% block content %}
        <div class = "OrderEquipPage"> <!-- This is a row -->

            <div class = "TheForm"> <!-- Splitting the row above into columns, this one is for taking input-->
                <form action="#" method="POST" enctype="multipart/form-data">
                    {% csrf_token %}

                    {{ form.as_p }}
                    <!-- Renders the form which takes the booking-->

                    <button name="save" type="submit" class= "buttons" id="send-request">Send Request</button>

                    {{ form.errors }}
                    <!-- Used to display all the error messages in the form like clashes in bookings, etc.-->
                </form>

                {% if messages %}
                    <ul class="messages">
                    {% for message in messages %}
                        <li  {% if message.tags %} class=" {{ message.tags }} " {% endif %}> {{ message }} </li>
                    {% endfor %}
                    </ul>
                {% endif %}

                    <p>testing</p>
                </div>

                <div id = "TheCalendar" unselectable="on"> <!-- Splitting the row above into columns, this one is for putting the calendar -->
                    <iframe id="calendar" src="https://tinyurl.com/yxf3v8pm" height="700" width="750"></iframe>
                </div>
            </div>
        {% endblock %}
    </body>
```

Based on the code above, the homepage of the website now looks like this:



Adding style to the form will be part of a later iteration in the development section.

5. To display the messages, I used the Django messages library. This library would test for certain queries to make sure things like whether a booking exists, a teacher is booked, room is booked, equipment is booked, etc., and then display a message on the website's template.
6. The most important task of this iteration was to follow the design of module 1 in the design section to enable the user to make a booking and send an email to the technician. This process involved several steps:

a. The first difficulty was to store all the needed information of a booking and then send it off to the `Lessons` table in the backend. To do so, I created a dictionary called `booking_details = {}`. The keys in the dictionary were `'staff'`, `'period_time'`, `'date'`, `'practical_booking'`, `'room'`, and `'number_students'`. The values for each of these pairs was the respective value for the key. These values were input by the user using the form made in the previous step of this iteration. The dictionary is below:

```
booking_details = {
    'staff': '',
    'period_time': '',
    'date': '',
    'practical_booking': '',
    'room': '',
    'number_students': '',}
```

b. Next job was to add the values for each of these pairs using what the user input in the form on the html homepage of the website. The code snippet for the same is below:

```
staff_name = form.cleaned_data.get('staff')
#   get the data from form
staff_obj = Staff.objects.filter(staff_name=staff_name)[0]
#   get the object of that data from the respective table
booking_details['staff'] = staff_obj
#   add the object from above to the dictionary

period_number = form.cleaned_data.get('period_time')
period_obj = Period.objects.filter(period_number= period_number)[0]
booking_details['period_time'] = period_obj

booking_date = form.cleaned_data.get('date')
booking_details['date'] = booking_date

practical_to_book = form.cleaned_data.get('practical_booking')
practical_booking_obj = Practical.objects.filter(practical_name=practical_to_book)[0]
booking_details['practical_booking'] = practical_booking_obj

room_to_book = form.cleaned_data.get('room')
room_obj = Room.objects.filter(room_name = room_to_book)[0]
booking_details['room'] = room_obj

booking_details['number_students'] = form.cleaned_data.get('number_students')
```

c. After this, I created a function that would use the data in this form to calculate the total equipment that the user is booking. This involves retrieving the practical and their equipment and their quantities from the many-to-many link table `Practical_Equipment_Needed` and multiply them by the number of students that the user entered in the form. The function for the above task is as follows:

```python
# function to calculate the total equipment needed to perform the practical
def calculate_total_equipment(booking_details):
    number_students = booking_details['number_students']    #var to store number of students
    practical_to_book = booking_details['practical_booking']    #var to store the practical to be done
    practical_equipment = 0
    equipment_names_list = []   #list to store names of all equipment in the practical that user wants to do
    equipment_quantities_list = []  #list to store quantities of all the equipment in the above list
    practical_id = int(Practical.objects.filter(practical_name = practical_to_book).values('id')[0]['id']) #id of practical
    equipments = Practical_Equipment_Needed.objects.filter(practical_id=practical_id).values('equipment_needed_id')
    #list of dictionaries that stores ID of each equipment

    # for loop needed to retrive name of equipment and their quanities
    for i in range (0, len(equipments)):

        # creating lists which store the name and quanities of each equipment needed in a practical
        equipment_name = Inventory_Equipment.objects.get(id = equipments[i]['equipment_needed_id']).name
        equipment_names_list.append(equipment_name)

        equipment_quantities = Practical_Equipment_Needed.objects.get(equipment_needed_id=equipments[i]['equipment_needed_id'], practical_id=practical_id).equipment_quantity
        equipment_quantities_list.append(equipment_quantities)

    print (equipment_names_list)        #check print
    print (equipment_quantities_list)   #check print

    total_equipment_needed = [i * number_students for i in equipment_quantities_list]   #list to store the total quantity of each equipment
    print (total_equipment_needed)      #check print

    return total_equipment_needed, equipment_names_list
```

d. After this, I created a function which returned the message string that was to be sent to the technician. This took three parameters, the lists of equipment, list with total quantities, and booking details dictionary. The code for the above is as follows:

```python
# Function to create a message that needs to go to the technician
def create_message(equipment_names, equipment_quantities, room_number):
    message_text = 'Room Number '+ str(room_number)+'\n'    #initial string with string of the message
    for i in range (0, len(equipment_names)):   # cycling through the equipmetn lists to add them to the message
        name_quantity_pair = ''
        name_quantity_pair = str(equipment_names[i]) + ' ' + str(equipment_quantities[i]) + '\n'
        message_text += name_quantity_pair

    print (message_text)
    return message_text

    '''
        The message looks something like this:

        Room Number B101
        10  Testtubes
        15  Rulers
        20  Clamps
        30  Bosses

        ##  I still need to add the time and date of the lesson in this message!
    '''
```

e. The final part of this stage was to send an email. Initially, I used an app called mailtrap to send my emails to fake accounts for testing. After this I made sure that the application was working and then added my personal email to the application's backend to send out emails. Doing so required me to make a procedure and add some code to the `settings.py` file.

Views.py function:

```
def email_message(message):
        send_mail(
            'Practical Request',
            message,
            '14pbegwani@patesgs.org',
            ['pranaybegwani@gmail.com'],
            fail_silently=False,
        )
```

Settings.py additions:

```
EMAIL_USE_TLS = True
EMAIL_HOST = 'smtp-mail.outlook.com'
EMAIL_HOST_USER = '14pbegwani@patesgs.org'
EMAIL_HOST_PASSWORD = [redacted]
EMAIL_PORT = 25
```

7. Lastly, I had to add this booking to the Lessons table. This was relatively simple and required a query with all parameters being elements of the `booking_details{}` dictionary.

```
Lesson.objects.create(
    staff=booking_details['staff'],
    period_time=booking_details['period_time'],
    date=booking_details['date'],
    practical_booking=booking_details['practical_booking'],
    room=booking_details['room'],
    number_students=booking_details['number_students'])
```

ITERATION 5 REVIEW:

What has been done?

Throughout this iteration of the development stage, my focus was to add functionality to the homepage form. This involved creating a booking for practical. The input needed by this form included:

- Name of staff booking
- Room to book
- Practical to book
- Lesson number to book for
- Date of the booking
- Number of students in the lesson

These details of the booking would then be added to a table called 'Lesson' which would store all bookings made. These would be needed to avoid clashes later.

Furthermore, I added the functionality to calculate the total equipment needed based on the number of students and the practical details that should have been added previously. In addition to this, based on the plan of the project, I added the functionality to send an email to the technician (or any other email), with the list of all equipment needed, the date, and the room number.

How was it tested?

To test the features, I essentially just checked if the 'dynamic' version of the website's templates worked as they did previously, ensured that the buttons and navbar were functioning as before.

To test the email functionality, I used an application called MAILTRAP which is essentially an email outbox and stores all the emails being sent without actually sending them to the recipient. Then, I undertook some alpha testing to see if correct lists of equipment were being emailed out in all cases.

In addition to these, I tested whether bookings were being added to the Lesson's table. I also tested the Django admin to add relatively static details to the database such as, the rooms in the school, staff in the school, number of lessons, etc. I tried a variety of length of inputs in the textboxes. I made sure if everything else previously working was still working!

Below is summary of testing undertaken for both functionalities:

Calendar of the Physics Dept Displayed after login	Not yet added

All boxes take the correct input	
Total equipment calculated correctly	
Email sent to technician with correct equipment	
Send Request button send correct email to technician/email	
Calendar shows booking made as a tag	
Cursor becomes finger pointer and buttons' text becomes red when hovered over – ALL BUTTONS	
Add Practical button leads to respective page	
Delete Practical button leads to respective page	
Add New Stock leads to page to the add to inventory page	
Report loss/ breakage page leads to respective page	
Editing practical page leads to the respective page	

Success Criteria and Stakeholder demands met:

11.	Successfully calculate the total equipment needed to be ordered	This involves, selecting the name of the practical, adding number of teacher and student sets, calculating the total equipment needed.
12.	Successfully make a list that needs to be emailed to the technician	This involves making a list of all the equipment that was calculated and placing them like a message.
13.	Successfully emailing the created list to the technician	This involves successfully sending an outlook email of the list created to the technician.

Changes in the design section:

No changes from as planned so far. The application is pretty much following Module 1 of the design section.

Summary of the whole Project as a prototype at this stage:

The project is at a good stage currently. I need to work on adding input validations, fixing stylistic elements, and creating a system to avoid booking clashes now. Other than those, the application is pretty much complete as the core of the application of being able to select practical and order equipment works pretty well and based on feedback, stakeholder is satisfied!

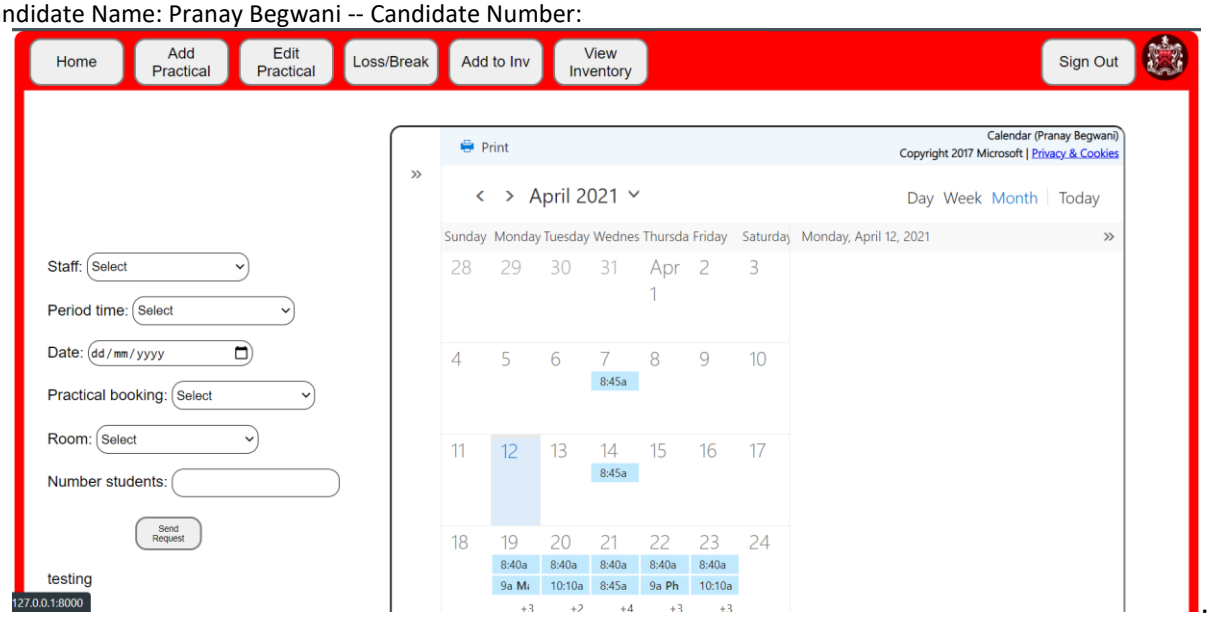
ITERATION 6 – DESIGN + STAKEHOLDER FEEDBACK

The primary goal of this iteration was to add to the stylistic aesthetic element of the website. The stakeholder was satisfied with the design of the elements in the first two iterations of the website therefore, I decided to strive for uniformity and gave all the pages of the website, a similar design. This meant that I could reuse a large part of the CSS code of the website, making my process easier!

1. The most difficult part of this iteration was to ensure that the designs of the different pages of this website are consistent and followed a similar pattern. Although pretty simple, because I essentially had to reuse CSS code from other elements of the website, I had to figure out a way to link Django fields to CSS because of how the forms were rendered, as a whole rather than as individual fields, using simple `{{ form.as_p }}` tags in the templates.
2. To fix this, I had to undertake some research to assign an ID and/or a CLASS to each field of the form. Once I found it, it was relatively simple to implement. This required me to make modifications to the `forms.py` file of the project where forms used are actually defined to start with. The homepage form now looked like this:

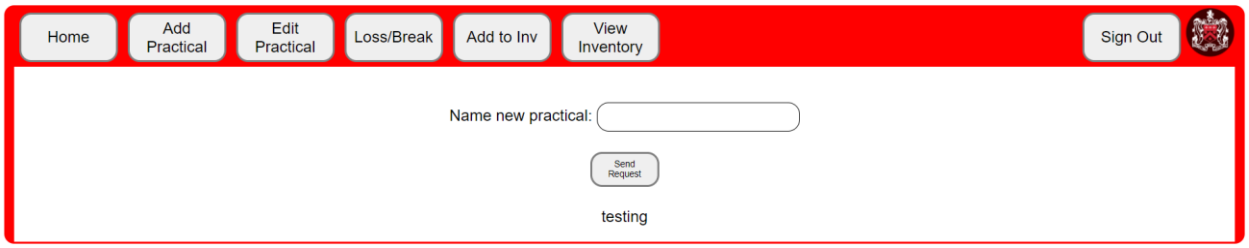
```
# Form to book a lesson in a particular room for some number of students
class Book_Lesson_Form(forms.Form):
    staff = forms.ModelChoiceField(
        queryset=Staff.objects.all(),
        empty_label='Select',
        widget=forms.Select(attrs={'class' : 'small-drop-downs'}))
    # dropdown to select staff name
    period_time = forms.ModelChoiceField(
        queryset=Period.objects.all(),
        empty_label='Select',
        widget=forms.Select(attrs={'class' : 'small-drop-downs'}))
    # dropdown to select choice of lesson
    date = forms.DateField(widget=forms.widgets.DateInput(
        attrs={'type': 'date', 'id': 'date-widget'}))
    # date field that display a calendar allowing the user to select a specific date
    practical_booking = forms.ModelChoiceField(
        queryset=Practical.objects.all(),
        empty_label='Select',
        widget=forms.Select(attrs={'id' : 'practical-list'}))
    # dropdown to select what practical to boo for
    room = forms.ModelChoiceField(
        queryset=Room.objects.all(),
        empty_label='Select',
        widget=forms.Select(attrs={'class' : 'small-drop-downs'}))
    # dropdown to select room number to book
    number_students = forms.IntegerField(
        widget=forms.TextInput(attrs={'class' : 'text-boxes'}))
    # integer field to enter the number of students in the lesson being booked
```

3. After this, I simply used the static CSS file that I had in Iteration 1 to apply style for the given '`class`' in the code above. The form on the homepage now looked like this:

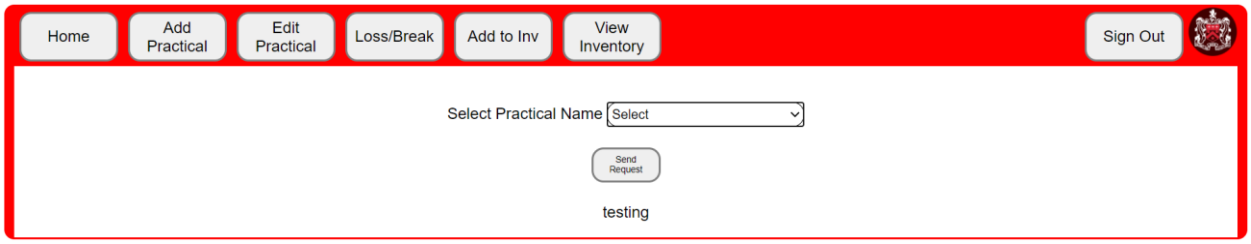


4. Using similar code in the forms.py file and CSS files, I was able to get the other pages to look similarly styled.

   Add Practical:

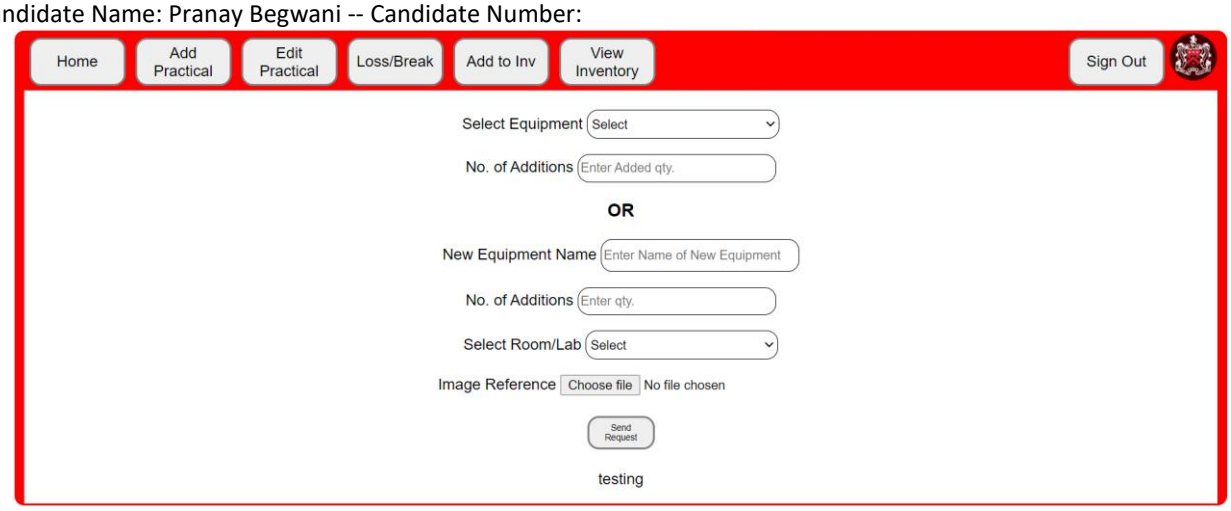


   Edit Practical:



   Loss/Break:

   Add to Inv:



5. After this I added another page to the application. This wasn't planned initially but **looking at stakeholder feedback** from the previous iteration, I felt that it would be useful. I added a page that allowed the users of this website to view booking history. This page would be in a tabular format and would allow the users to view all details of a lesson booked.
   The process of doing this was very similar to the process of adding the page that helped the user to view the inventory.
   The steps of doing the same were as follows:
   - Add a button on the navbar.
     `Base.html`: part of the code for the navbar:

```
<div id="BarLeft"> <!-- All the individual buttons for the navbar -->
    <a href= "/">
        <button class="buttons" id="Home">Home</button>
    </a>
    <!-- <a href="/AddPractical"> -->
    <a href="/NameNewPractical">

        <button class="buttons" id="Add">Add Practical</button>
    </a>
    <a href="/SelectPractical">
        <button class="buttons" id="Edit">Edit Practical</button>
    </a>
    <a href="/LossReport">
        <button class="buttons" id="Report">Loss/Break</button>
    </a>
    <a href= "/AddInventory">
        <button class="buttons" id="Inventory">Add to Inv</button>
    </a>
    <a href= "/ViewInventory">
        <button class="buttons" id="View_Inventory">View Inventory</button>
    </a>
    <a href= "/BookingHistory">
        <button class="buttons" id="Booking_History">Booking History</button>
    </a>
    </div>
```

   - Add a url for this button in the `urls.py` file:

```
path("BookingHistory/", views.booking_history, name="booking_history"),
```

   - Add a function in the views.py file to retrieve data from the Lesson table in the database:

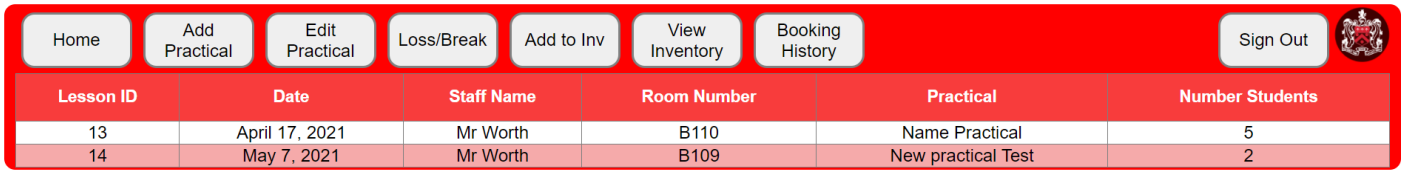
```
# function to get all the bookings made
def booking_history(response):
    lesson_obj = Lesson.objects.all()   #querys every single record into this identifier
    return render(response,"main/BookingHistory.html", {'lesson_obj': lesson_obj}) #template rendered & lesson_obj passed as context to template
```

- Create a new template that would display the table.

  The code for the main part of this template is as follows:

```
{% block content %}
    <table>
        <thead>
            <tr>
                <th class="tabled-headers">Lesson ID</th>
                <th class="tabled-headers">Date</th>
                <th class="tabled-headers">Staff Name</th>
                <th class="tabled-headers">Room Number</th>
                <th class="tabled-headers">Practical</th>
                <th class="tabled-headers">Number Students</th>
            </tr>
        </thead>

        <tbody>
            <!-- inventory_obj is the context passed from the template to refer to the table -->
            <!-- the lines below loops through the records of the table displaying needed information -->
            {% for lesson in lesson_obj %}
                <tr>
                    <td class="table-blocks">{{ lesson.id }}</td>
                    <td class="table-blocks">{{ lesson.date }}</td>
                    <td class="table-blocks">{{ lesson.staff }}</td>
                    <td class="table-blocks">{{ lesson.room }}</td>
                    <td class="table-blocks">{{ lesson.practical_booking }}</td>
                    <td class="table-blocks">{{ lesson.number_students }}</td>
                </tr>
            {% endfor%}
        </tbody>
    </table>
{% endblock %}
```

6. After following the above series of steps, a new button was added to the navbar of the application – called 'Booking History'. This button led to the above HTML template which displayed details of the practical being performed in a lesson. The new page of the application was styled similarly to the View Inventory page and it looks like this:



Lesson ID	Date	Staff Name	Room Number	Practical	Number Students
13	April 17, 2021	Mr Worth	B110	Name Practical	5
14	May 7, 2021	Mr Worth	B109	New practical Test	2

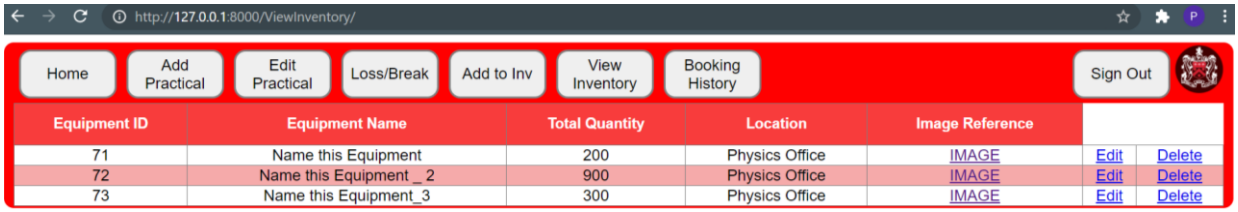
As shown above, a new button is added to the navbar and the table is showing all important details of a lesson booked.

7. The final role of this iteration was to work on previous stakeholder feedback and add functionality to delete practicals and inventory items. To implement the functionality to delete a practical:

   First added a column in the View Inventory table which included the option to delete a record/inventory item. The code for to add a column is below:

```
{% for equipment in inventory_obj %}
    <tr>
        <td class="table-blocks">{{ equipment.id }}</td>
        <td class="table-blocks">{{ equipment.name }}</td>
        <td class="table-blocks">{{ equipment.total_quantity }}</td>
        <td class="table-blocks">{{ equipment.location }}</td>
        <td class="table-blocks"><a href= "{{ MEDIA_URL }}{{ equipment.img_reference }}">IMAGE</a></td>
        <td class="table-blocks">
            <a href="/EditInventory/{{ equipment.id }}" id="Edit-Page-Link">Edit</a>
        </td>
        <td class="table-blocks">
            <a href="/Delete/{{ equipment.id }}" id="Delete-Page-Link">Delete</a>
        </td>
    </tr>
{% endfor%}
```

The inventory page after this addition looked like this:



As shown above, the last column gives an option to delete a particular equipment entirely.

The code for the views.py file to delete the record is shown below:

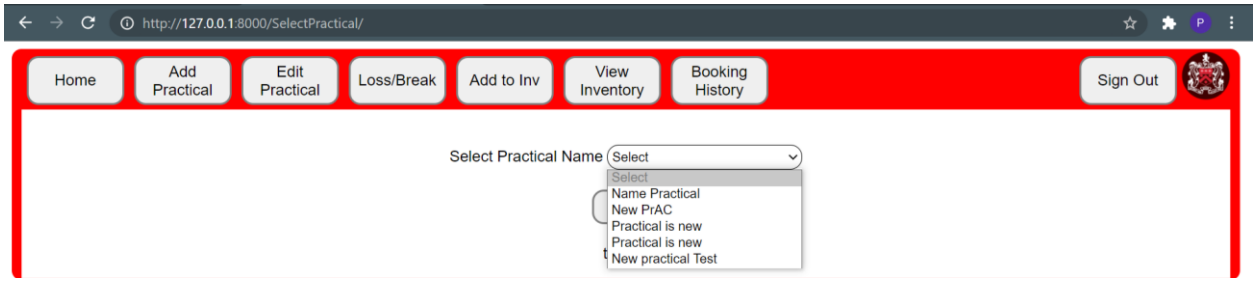
```
# function to delete exisiing inventory item from the table
def delete_inventory_item(request, id):
    equipment_delete = Inventory_Equipment.objects.get(id = id) # stores the quereyset of the item that needs to be deleted
    equipment_delete.delete()   # deletes the item
    return redirect('/ViewInventory')   #redirects to the inventory page

    return render(request,'main/DeleteEquipment.html', {})
```

8. The code to allow the user to delete equipment from a practical was relatively simple and required only one addition to the formset code. The parameter to be added was `can_delete = true`. The final formset is as follows:

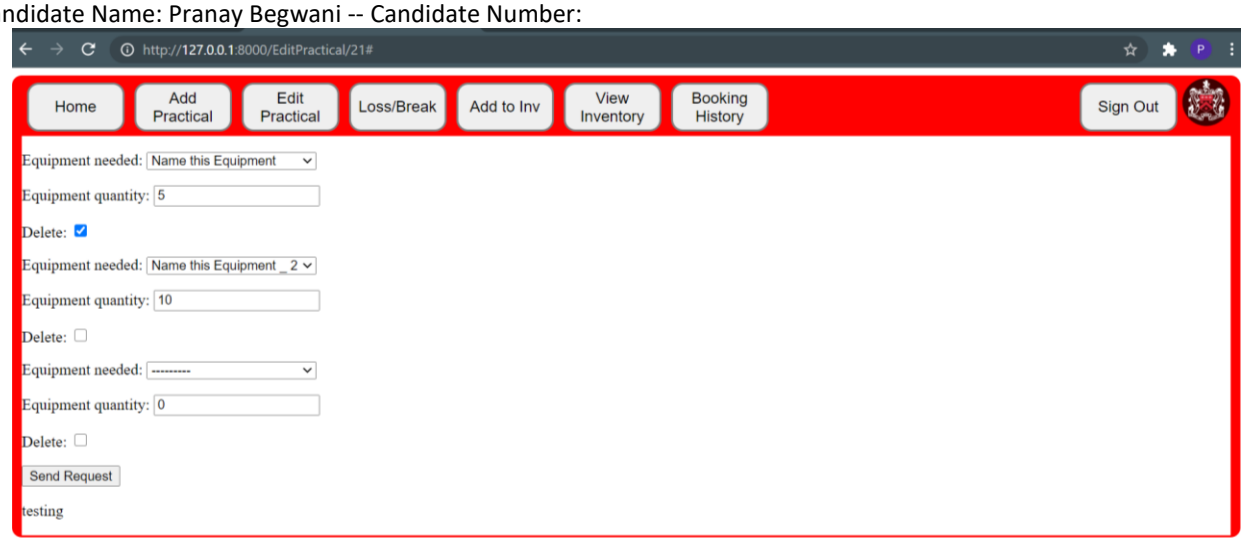
```
# form to input a single equipment and its quantity for a practical
class New_Practical_Detail_Form(forms.ModelForm):   # declaring the form
    class Meta:
        model = Practical_Equipment_Needed      #the model that this form is connected to
        fields = ('equipment_needed', 'equipment_quantity', )   # fields needed in the form

# formset which creates multiple instances of the form above
Add_Practical_Formset = modelformset_factory(   #declaring the formset
    Practical_Equipment_Needed,                 #the models that this formset is connected to
    New_Practical_Detail_Form,                  #the form (above) whose copies this formset creates
    can_delete=True
)
```

Doing so now gives the user an option to select equipment to delete. This works as follows:

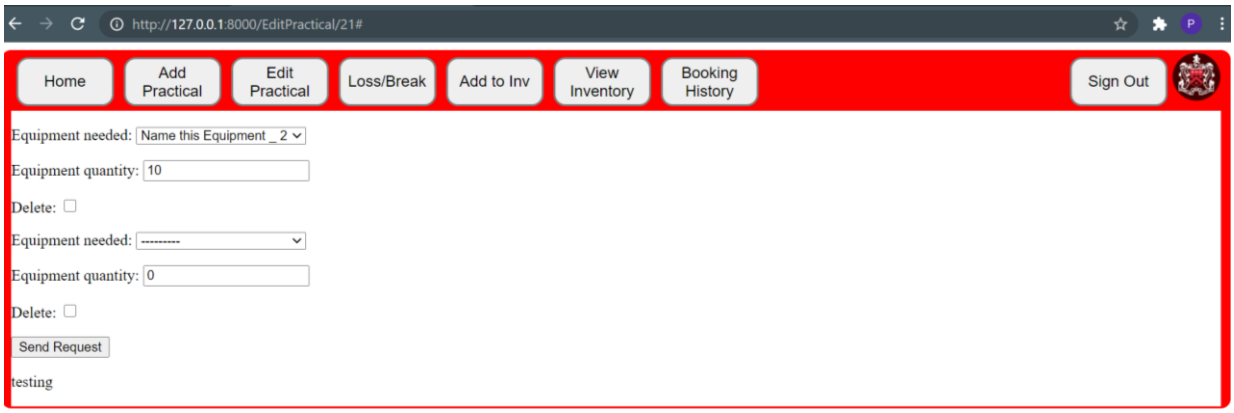
Select practical to edit:



Equipment of the practical is displayed. Select the equipment to delete entirely:



Clicking send request will delete the first equipment permanently:



The id in the URL clearly shows that the page is displaying the same practical.

9. For the final part of this iteration, I fixed the style of the page that is used to edit a practical's equipment and their quantities.

ITERATION 6 REVIEW:

What has been done?

The primary focus for this stage of development was to develop the functionality that allows thew user to delete equipment from a practical – in the edit practical section of the application – and to ensure that the styling across the application was uniform.

How was it tested?

The testing undertaken ensured that all the previous features of the application worked perfectly fine even after modifications were made.

I then checked if deleting equipment and practical worked.

Testing methods undertaken:

Clicking 'DELETE' in the view inventory table deletes the equipment from inventory	
Display a textbox and dropdown pair every time plus sign is clicked	
Dropdown for practical name displays names of all the practical in the database	
Clicking Edit displays all the current equipment and quantities of the chosen practical in dropdowns and textboxes respectively on a new page	
Clicking the checkbox deletes the respective equipment from the database	
Display an empty/default textbox and dropdown pair every time plus sign is clicked	
Dropdowns of equipment contains all the equipment added in the inventory	
Display confirmations before deleting	
Successfully delete the selected practical from the database when delete is clicked by the user	
All the buttons in the navbar work as intended	
Database reflects the changes made when 'Send Request' clicked by user.	

Success Criteria and Stakeholder demands met:

Apart from checking that the previously completed success criteria worked, I completed the below tasks. The application could now add a new practical to the database, allow the user to delete equipment and details of an existing practical.

6.	Delete an existing practical	This involves successfully selecting a practical from a drop-down menu, deleting it, receiving a confirmation and viewing it in the database.

Changes in the design section:

No changes to the design section were made in this iteration.

Summary of the whole Project as a prototype at this stage:

The project works as expected and in the next iteration, I will strive to complete the project after adding some simple validation for the fields and forms. I will also aim to create a simple login for the project to restrict access to features of the application.

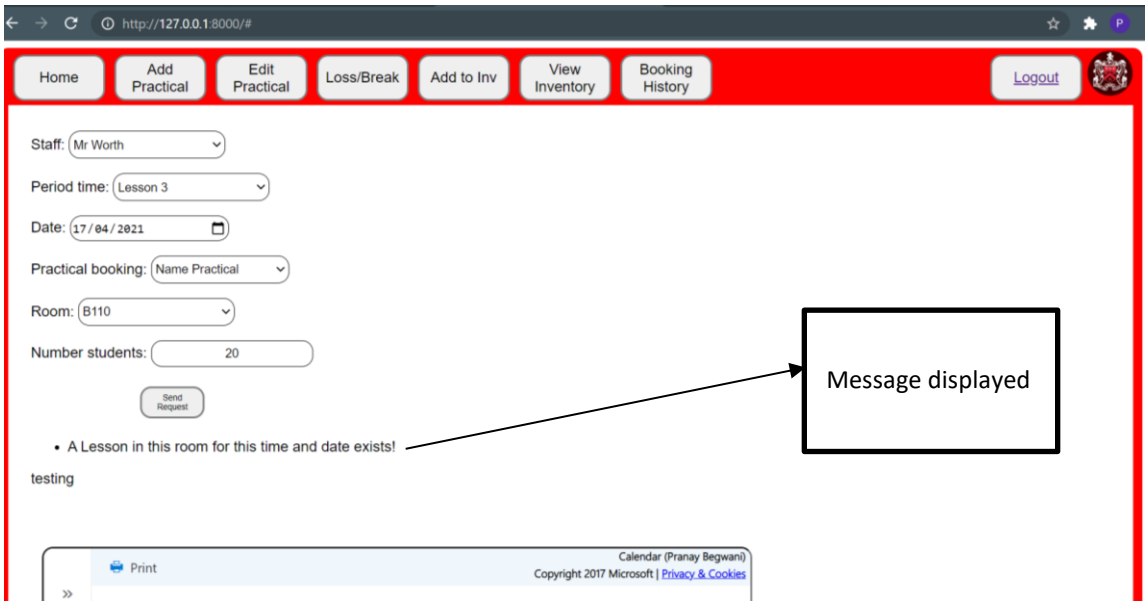
ITERATION 7 – LOGIN + BOOKING CHECK

As suggested by the heading, the primary purpose of this iteration was to develop the functionality that is used to avoid room booking clashes and to create the login and logout features of the application. Although making a login was initially part of the plan, I did not go through an extensive design stage for the same at that stage because it was one of the less important features of the application.

1. Firstly, I added clauses to check for bookings. This involved looking for whether a teacher is busy at a given lesson on a given date. Checking this was as simple as querying a record for that particular time and then checking if the teacher is busy. This code was added to the `homepage()` function of the `views.py` file before the other functions to calculate booking totals were calculated. The lines of code to check the same are as follows:
   This was used to ensure if a room was booked during a given time.

```
if (Lesson.objects.filter(date=booking_details['date'], period_time=booking_details['period_time'], room=booking_details['room']).exists()):
    messages.info(request, 'A Lesson in this room for this time and date exists!')
```

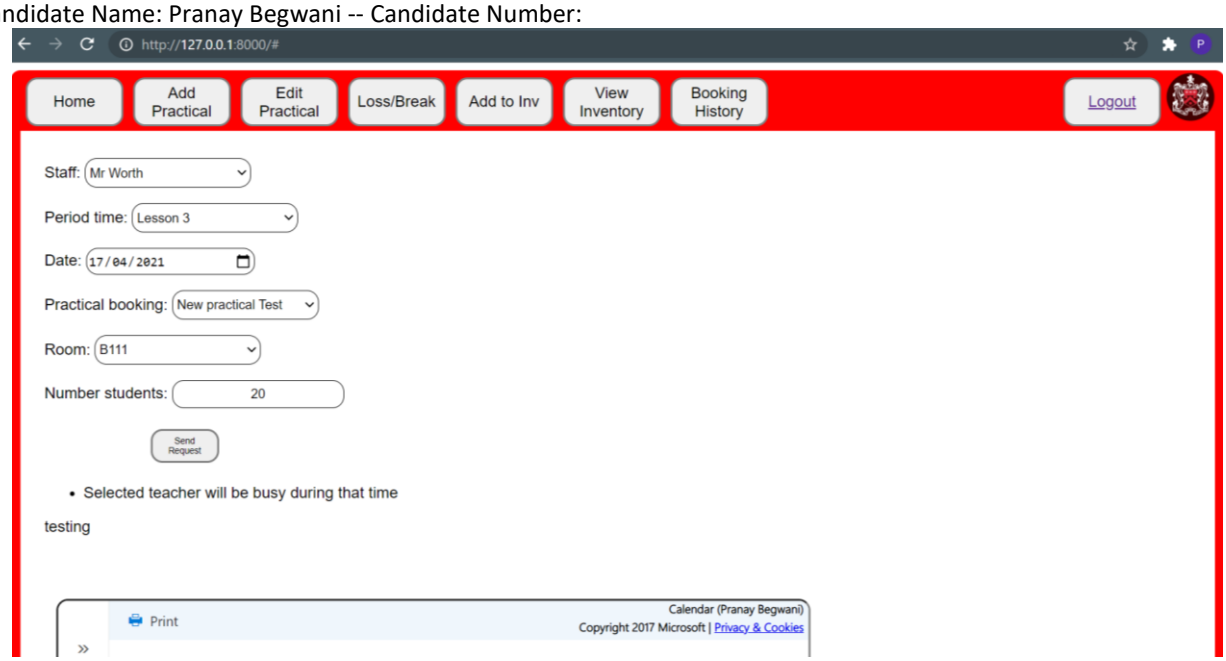
   On the website, this code of a clashed booking results in the following message:



2. Next role was to check if a particular teacher will be busy at a particular time. The code for the same is below:

```
elif (Lesson.objects.filter(staff=booking_details['staff'], date=booking_details['date'], period_time=booking_details['period_time']).exists()):
    messages.info(request, 'Selected teacher will be busy during that time')
```

   On the website, making a booking with a busy teacher results in the following:



3. After this, I was trying to add the code to check if equipment at a particular time is available or not. This required a simple scheduling algorithm, but I was not able to implement it at this stage and hence, I skipped this stage for the time being.
4. Therefore, there are certain types of **validations** that are being undertaken in this case:
   - Essentially, it is a look-up table check which ensures that there are no repeats in the databases and teachers cannot be multiple times for the same lesson on the same date. It also makes sure that the same room isn't booked multiple times.
5. Because of the issue above, I moved on to build the login system of the website. This involved three major pages:
   - Login: Used to login existing users
   - Logout: Used to logout a person – leads back to the login page
   - Register: Used to create new users that may then login after a successful Registering session

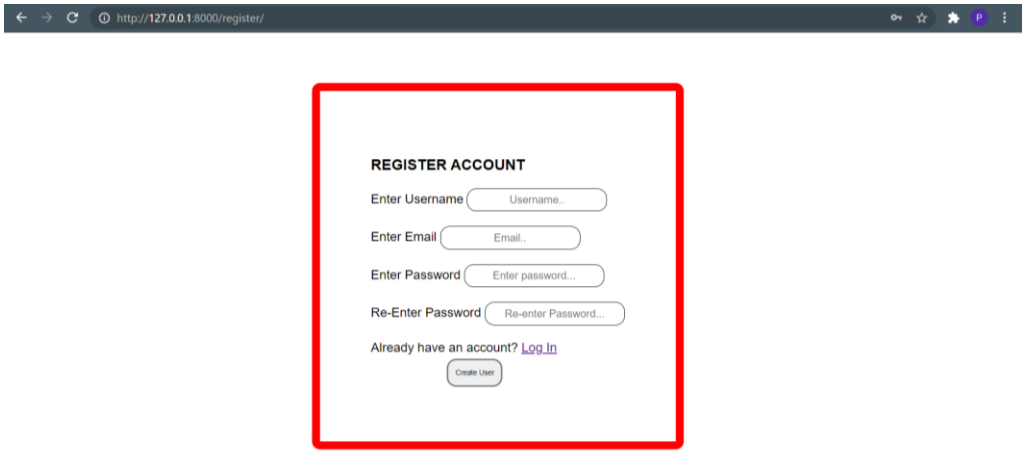
   The first job in the development of this section was to create a form called the `UserCreationForm` in Django. This form is used for creating new users and is implemented in the `forms.py` file as below:

```
from django.contrib.auth.forms import UserCreationForm
from django.contrib.auth.models import User


class CreateUserForm(UserCreationForm):
    class Meta:
        model = User    #built in model inside a django databse needed to store al users
        fields = ['username', 'email', 'password1', 'password2', ]  #the fields that a user needs to fill in to register themselves
```

This form above would then be rendered on the `register.html` file which is the template needed for a user to register. The `register.html` template to create a new user had relatively simple code which looked like this:

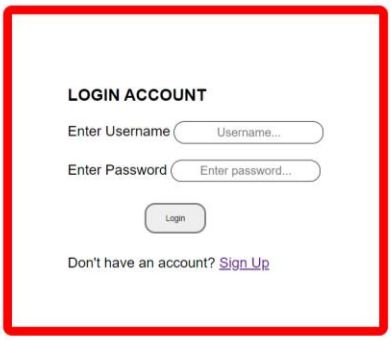
```
<body>
    <!-- Below is a div section for a form -->
    <div class='page' id='login'>
        <form class='FormForRegister' method="POST" action="#">
            <h3 id="form-title">REGISTER ACCOUNT</h3>
            {% csrf_token %}
            <label>Enter Username</label>
            <input class='text-boxes' type="text" name="username"/>
            <br>
            <br>
            <label>Enter Email</label>
            <input class='text-boxes' type="text" name="email"/>
            <br>
            <br>
            <label>Enter Password</label>
            <input class='text-boxes' type="password" name="password1"/>
            <br>
            <br>
            <label>Re-Enter Password</label>
            <input class='text-boxes' type="password" name="password2"/>
            <br>
            <br>
            {{ form.errors }}
            Already have an account? <a href="{% url 'login' %}">Log In</a>
            <button class="buttons" id="send-request" type="submit">Create User</button>
        </form>
    </div>
    <script>

        //Query All input fields
        var form_fields = document.getElementsByTagName('input')
        form_fields[1].placeholder='Username..';
        form_fields[2].placeholder='Email..';
        form_fields[3].placeholder='Enter password...';
        form_fields[4].placeholder='Re-enter Password...';
        for (var field in form_fields){
            form_fields[field].className += ' form-control'
        }
    </script>
</body>
```

To add styling to this I used inline CSS which was copied from some of the previous elements of the website – to add consistency between the different parts of the website. Post this styling, the register page of the website looked like this:



6. Similar to the above role, I also create the login page of the website using html. The html code and the final login page after styling is as follows:

```
<body>
<div class='page' id='login'>
    <form class="FormForLogin" method="POST" action="">
        <h3 id="form-title">LOGIN ACCOUNT</h3>
        {% csrf_token %}
        <label>Enter Username</label>
        <input class='text-boxes' type="text" name="username"/>
        <br>
        <br>
        <label>Enter Password</label>
        <input class='text-boxes' type="password" name="password"/>
        <br>
        <br>
        <button class="buttons" id="send-request" type="submit">Login</butt
        {% for message in messages %}
        <p>{{ message }}</p>
    {% endfor %}
    <br>
    <br>
    Don't have an account? <a href="{% url 'register' %}">Sign Up</a>
    </form>
</div>
<script>
    //Query All input fields
    var form_fields = document.getElementsByTagName('input')
    form_fields[1].placeholder='Username...';
    form_fields[2].placeholder='Enter password...';
    for (var field in form_fields){
        form_fields[field].className += ' form-control'
    }
</script>
```



7. After this, I added urls to the above pages in the `urls.py` file.

```
path('login/', views.loginPage, name="login"),
path('logout/', views.logoutUser, name="logout"),
path('register/', views.register, name="register"),
```

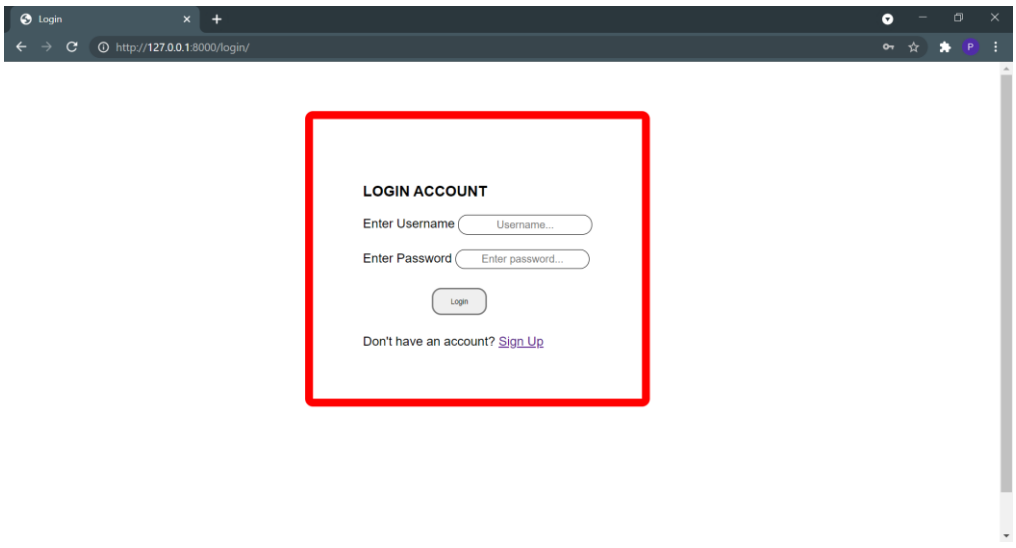
8. Finally, the last task was to create views.py functions for each of the above pages. These would be required to authenticate a user logging in and make sure that the emails and passwords meet the required criteria. And lastly I had to add this line:

`@login_required(login_url = 'login')` – above every function which was connected to the front-end to ensure that only users who are logged in can access those pages!

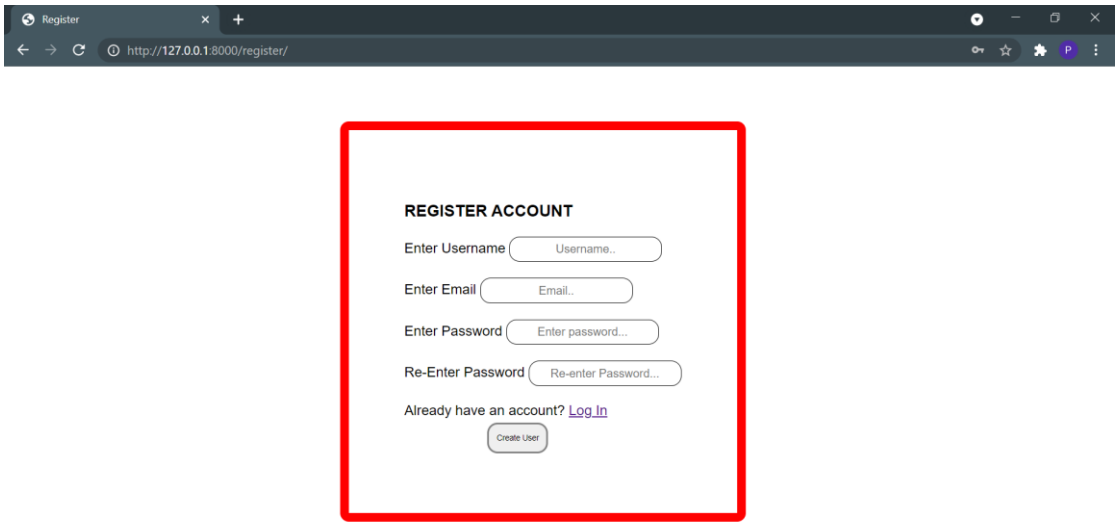
All the code of the `views.py` file for the login system is as follows:

```python
# function to register a new user
def register(request):
    if request.user.is_authenticated:   #if user is already logged in
        return redirect('/')            # takes user to homepage
    else:
        form = CreateUserForm()         #empty registration form

        if request.method == 'POST':
            form = CreateUserForm(request.POST)
            if (form.is_valid()):       # checks if email and passwords, etc., meet the criteria
                form.save()
                username = form.cleaned_data.get('username')
                messages.success(request, 'User Created Successfully for ' + username)  # success message
                return redirect('login')    #redirects to the login page so that new user can login to website
    return render(request, 'registration/register.html', {'form': form})

#function to login existing users
def loginPage(request):
    if request.user.is_authenticated:
        return redirect('/')
    else:
        if request.method == 'POST':
            username = request.POST.get('username')
            password = request.POST.get('password')
            user = authenticate(request, username=username, password=password)  # used to compare username and password to that in database
            if user is not None:    #checks if user is not blank
                login(request, user)
                return redirect('/')    #redirects to homepage
            else:
                messages.info(request, 'username or password is incorrect')

    return render(request, 'registration/login.html', {})

#function to logout user
def logoutUser(request):
    logout(request)         #logouts user when button clicked
    return redirect('login')
```

9. Last part of this iteration was to test user creation and logging in functionalities. Some of the screenshots of the application working are below:
   - Opening the website directly takes to the login page. This page gives the user an option to ‘sign in’ for the first time.

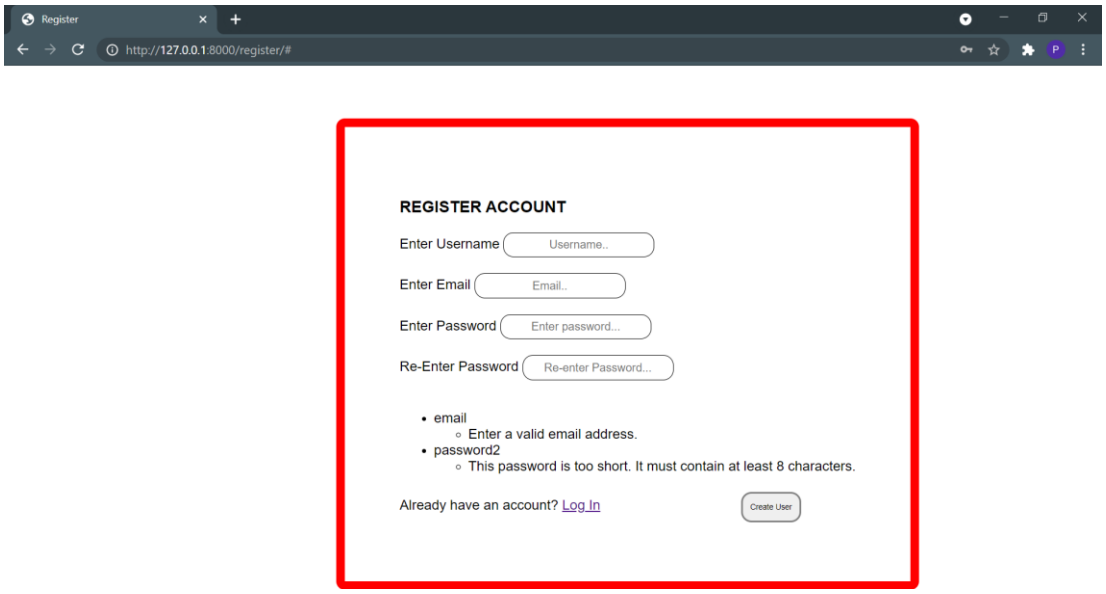


- Clicking the sign up button leads to the register page where the user can create a new account.

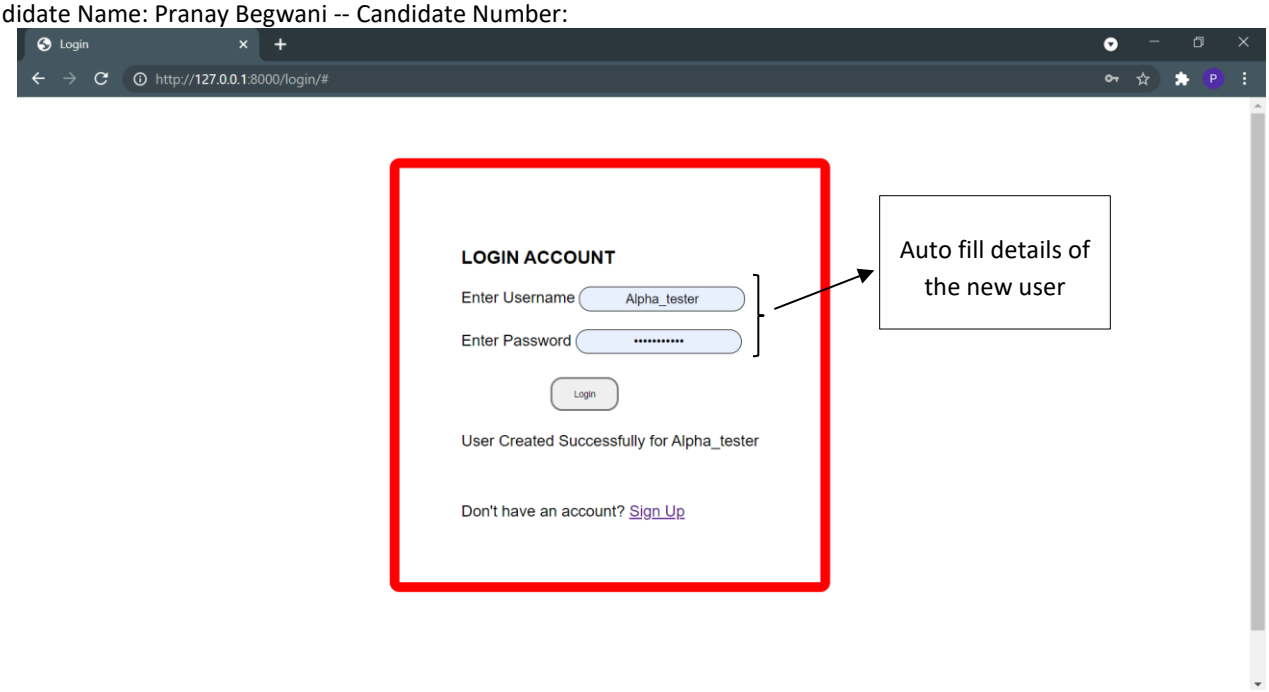


- Here, the user can enter their details and the create a new account. The app checks for valid details. These include:
    - Email in the correct format
    - Password of the correct length
    - Both passwords matching

An example of an incorrectly filled form:



A form correctly filled will lead directly to the login page:



10. Confirming and entering the correct details leads the user to the homepage of the website.
11. In conclusion, **validation** for the register page:
    - '@' – format check for the email address.
    - '._' – format check for the email address.
    - 8 characters minimum – length check for the password.
    - Password re-entry matches the original entry.
    - Password can only contain certain types of symbol.
    - Password cannot be the same as the username and cannot be similar to it.
12. **Validation,** for the login page:
    - Checks if username and password match the value in the database.

ITERATION 7 REVIEW:

What has been done?

Throughout this iteration, the primary focus was to add some validation to the booking system of the application to avoid repetitive bookings and clashes – a key demand of the stakeholder.

The next job was to create a login and sign up system and restrict access to the all parts of the website to only those who have logged in.

How was it tested?

Apart from testing and ensuring whether all the previously implemented features worked or not, I conducted rapid and repetitive testing on the login and registering engine of the website. Furthermore, I ensured whether or not appropriate validation was taking place when booking a practical for a lesson.

Testing methods undertaken:

New user is created using register page	
System ensures user input meets criteria	
Existing user can successfully login to the website	
Website access limited to only logged in user	
User can logout successfully using button	
Booking system on homepage shows a clash as a message	
Previously created features work as intended	
Database reflects the changes made when ‘Send Request’ clicked by user.	

Success Criteria and Stakeholder demands met:

Apart from checking that the previously completed success criteria worked, I completed the below tasks. The application could now add a new practical to the database, allow the user to delete equipment and details of an existing practical.

1.	Successfully login to the system	Successfully being able to login in the system using the school credentials.
18.	Log out	This involves successfully logging out of the application once the logout button is clicked.

Summary of the whole Project as a prototype at this stage:

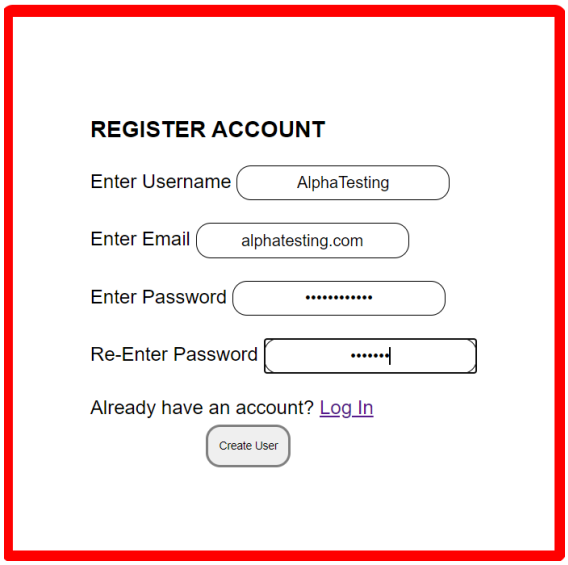
The website now has a fully functional, but simple, booking system for practicals. This system is capable of preventing some minor booking clashes and meets a key stakeholder demand. The

website also has a fully functional booking system which restricts access to users who have logged in.

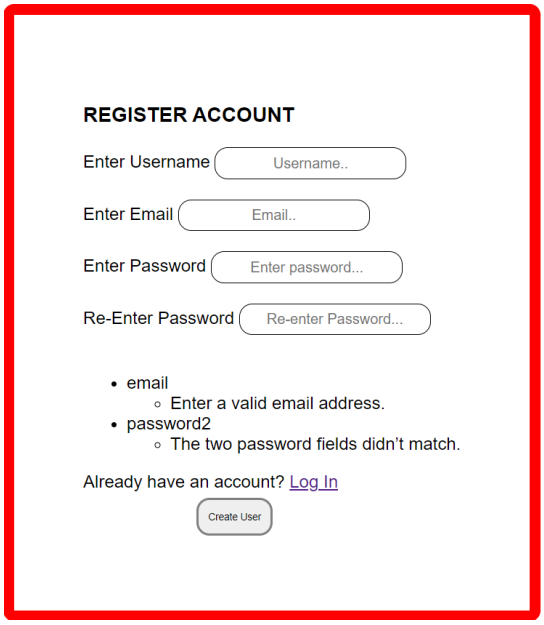
ITERATION 8 – FINAL TESTING

The sole purpose of this stage was to fully test every individual feature of the application.

1. First thing to test is the registration page. I need to make sure that it is clear what is expected by the data entered by the user.

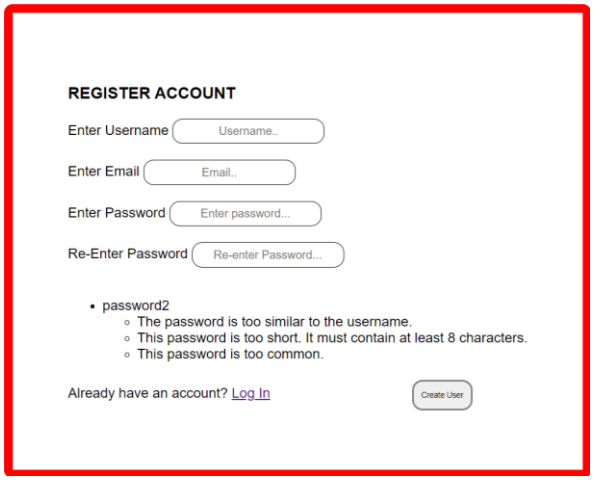


   a. Errors displayed should be about an incorrect email and different passwords. This occurs after input validation in my code takes place to ensure that the email contains '@' symbol and is followed by a '.'. In case of the password it checks if the password is at least 8 characters long and doesn't match the username too closely.
   In case of the repeated entry of the password it makes sure that the two passwords are the same.

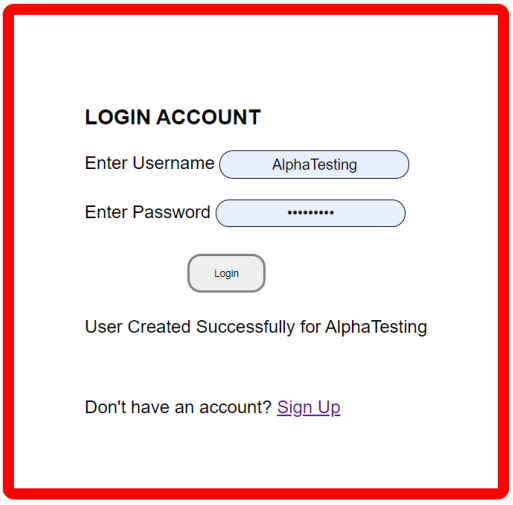


The above works.

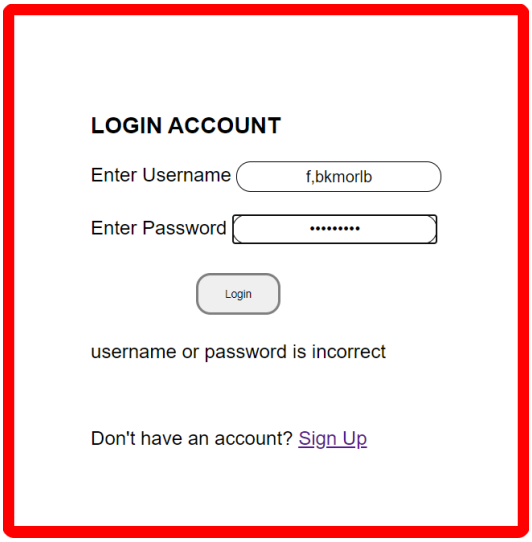
b. Checking by entering too simple a password:



c. Entering correct details:



d. Checking if incorrect login details prompt a message. It essentially checks if the username and password match each other in the database.



Correct username and password lead to the homepage.

2. I then checked if all the buttons of the navbar worked and led to the respective correct pages. This was working.
3. Next part was to add a new piece of equipment to the inventory.
    a. Leaving some inputs blank:
    Adds an equipment to the inventory with quantity None.

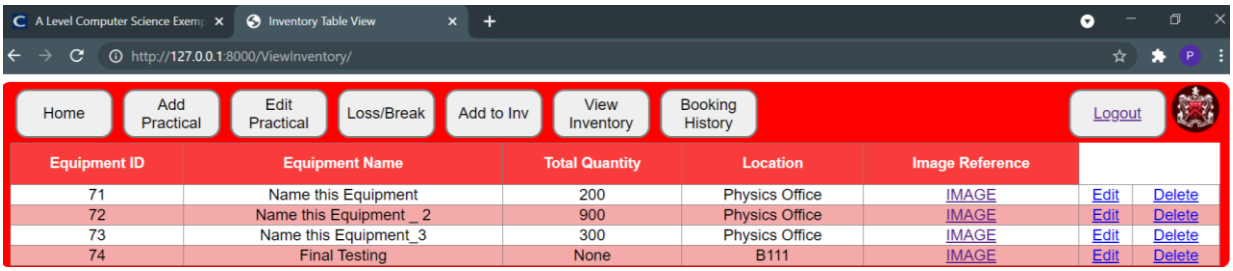
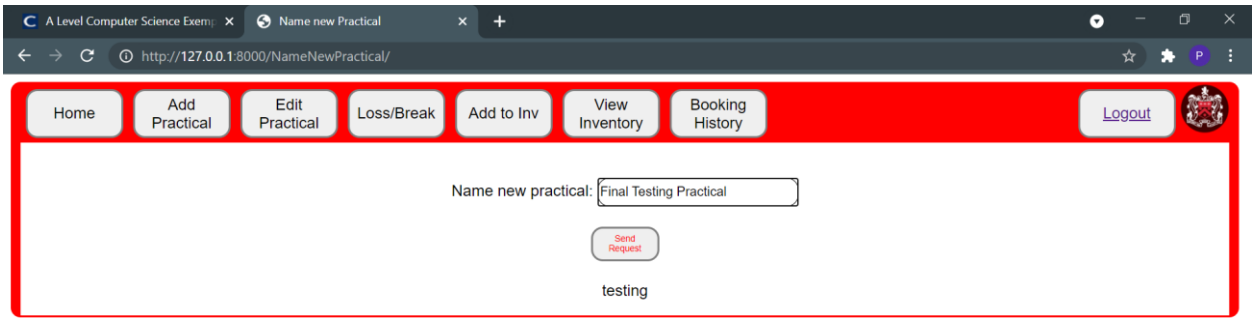
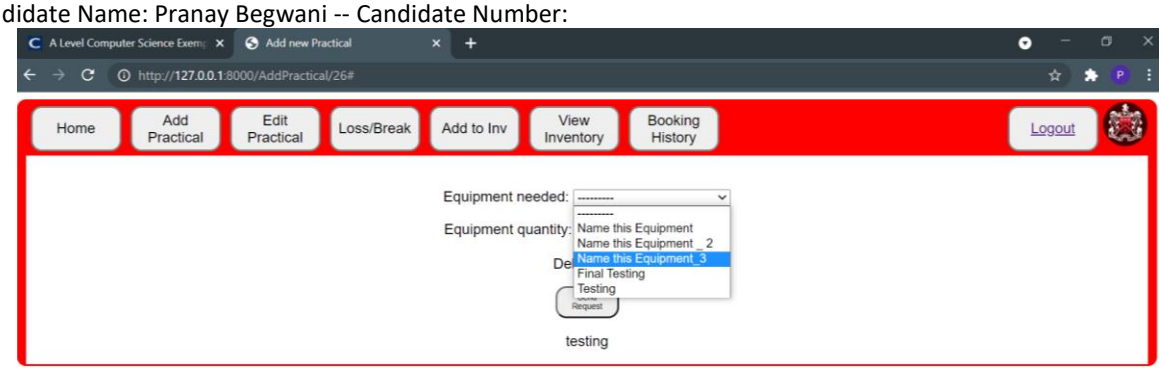


Image additions also work. **I have changed this so that the default quantity – if left blank – is 0. This form of validation was important because equipment can't be added to None type.**

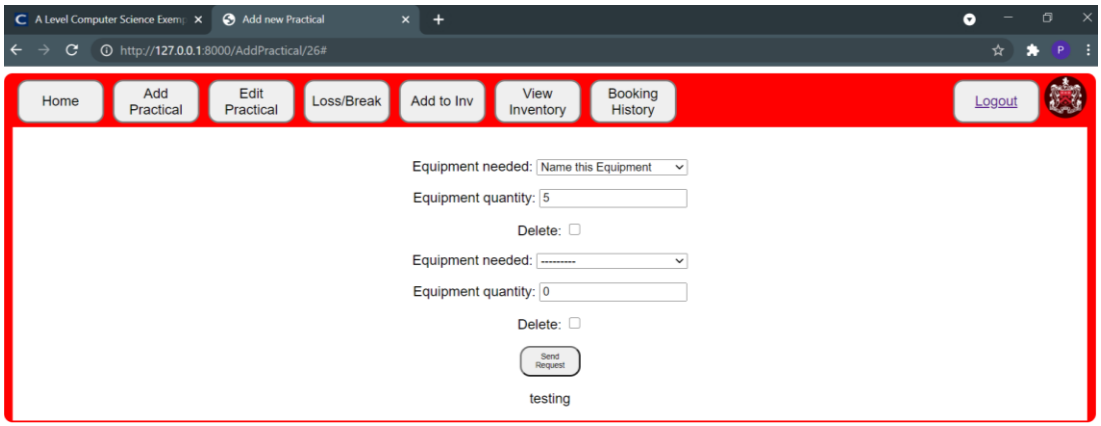
    b. Checking if quantity can be added to each equipment.
    This works!
4. After this I tested if quantities could be removed from individual equipment.
    a. The dropdown was displaying list of all equipment!
    b. Quantities were being removed successfully. If left blank, the application did nothing.
5. Next role was to create a new practical and add equipment to it.
    a. Practical name was input successfully and clicking the 'send request' button opened the form to select and enter equipment and quantities.



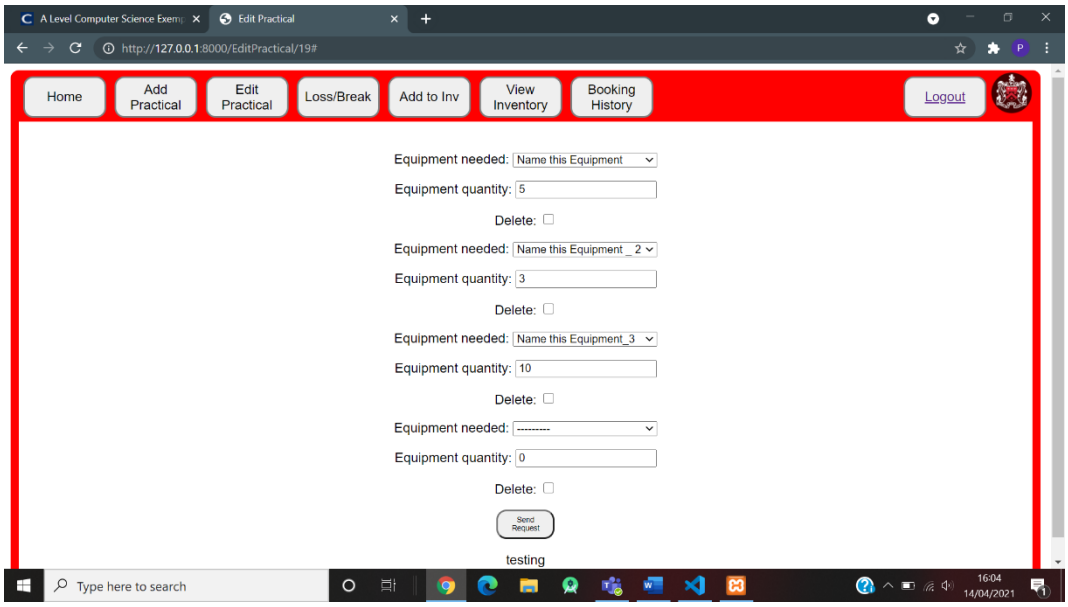
    b. Doing so displayed an empty formset where fields could be added dynamically after the save button was clicked. This form allowed user to select inventory equipment from dropdown and enter their quantities in a field.



After clicking send request, a new pair appears:



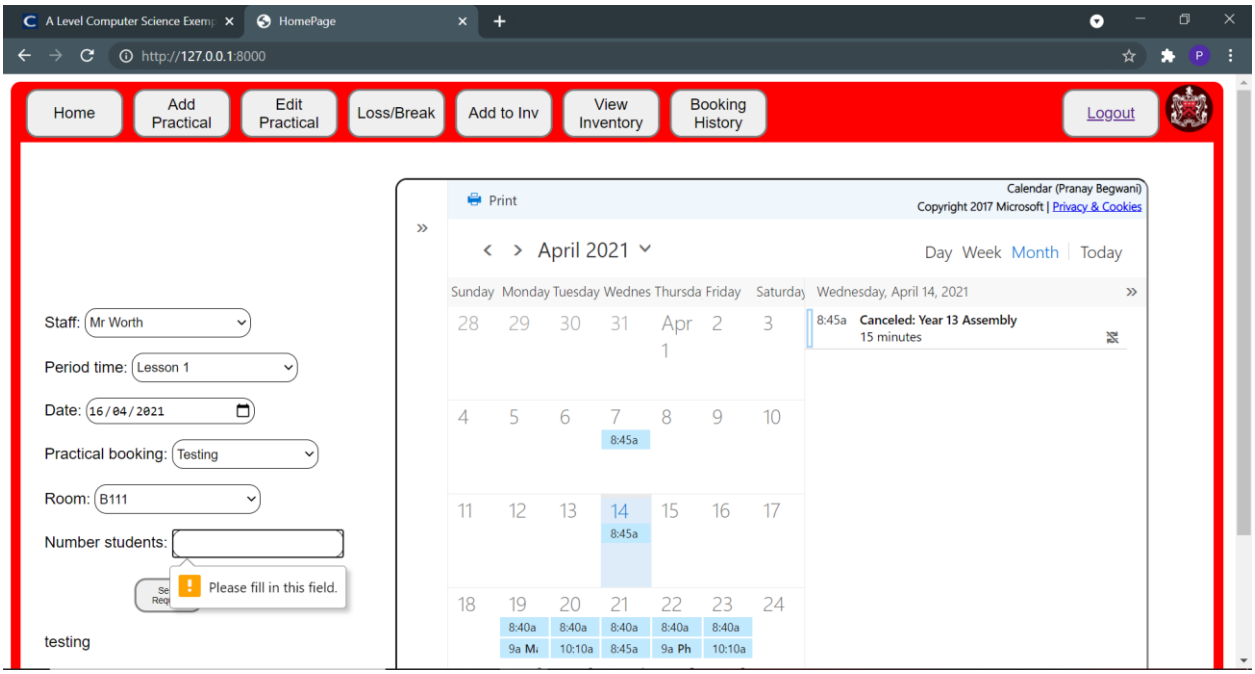
6. Next test was to check if existing practicals could be edited.
    a. I was able to select a practical to edit from the dropdown.
    b. The output on the page displayed the practical's equipments and their quantities in editable textboxes.



    c. New equipment could be added and existing equipment could be changed. Therefore practical editing worked.

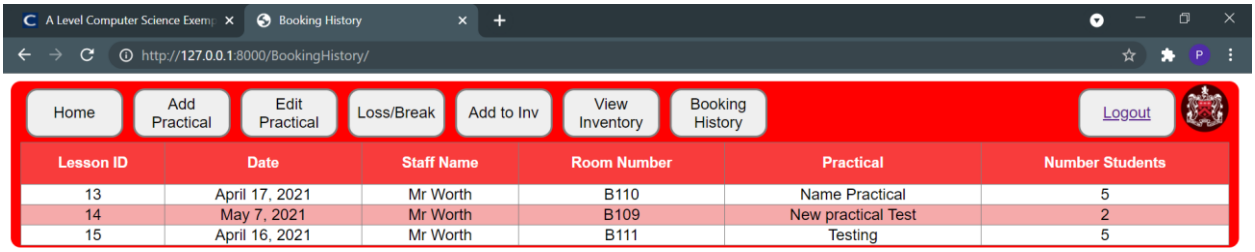
7. Finally, the next test was to make sure that a booking could be made.

a. Leaving any field blank when submitting would give a prompt, asking the user to fill in the field.



b. If all the fields were filled in, bookings were being made.

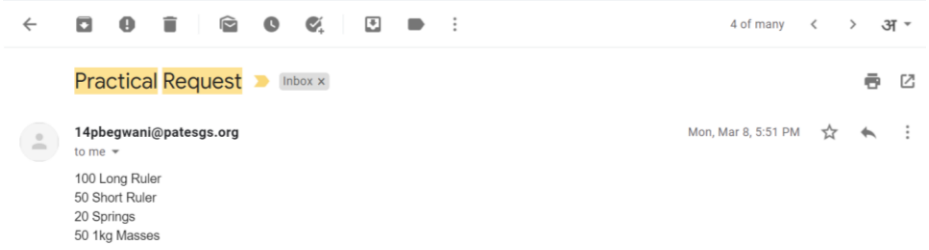
c. The application was notifying of all clashes as well.



Lesson ID	Date	Staff Name	Room Number	Practical	Number Students
13	April 17, 2021	Mr Worth	B110	Name Practical	5
14	May 7, 2021	Mr Worth	B109	New practical Test	2
15	April 16, 2021	Mr Worth	B111	Testing	5

d. The table above is the booking history table and it clearly shows the practical that I was trying to add. Therefore the booking history page also works.

e. The homepage form then sends an email to the address that is currently hardcoded in the code. The email looks as shown below:



8. Final test was to make sure that details of an inventory item could be edited and an item could be deleted – using the View Inventory page of the website. All of this worked successfully!

9. Testing Table Summary:

Register page taking valid input from user	
Login page taking valid username and password from the user	
Login page prompting if username and/or password are incorrect	
Logout button leading back to login page	
Access to the main pages of the website only available to logged in users	
All buttons of the navbar leading to the respective pages	
Successfully add a new inventory item	
Successfully add new stock to existing equipment in inventory	
Successfully remove from an existing item's inventory quantity	
Homepage form takes input	
Homepage form conducts validation for the input	
Homepage sends email for a valid form	
Homepage calculates total equipment correctly	
Textboxes display prompts if validation is failed	
Checking if new practical is already in the database and prompting user if it is	
Lead to page to enter details of practical after button click	
Display dropdown of equipment and textbox for quantities of the equipment	
Display a textbox and dropdown pair every time 'Send Request' is clicked	
Dropdown for practical name displays names of all the practical in the database	
Clicking Edit displays all the current equipment and quantities of the chosen practical in dropdowns and textboxes respectively on a new page	
Clicking the 'dustbin' deletes the respective equipment from the database	
Display an empty/default textbox and dropdown pair every time 'send request' is clicked	
Dropdowns of equipment contains all the equipment added in the inventory	
Successfully delete the selected practical from the database when delete is clicked by the user	
Database reflects the changes made when 'Send Request' clicked by user.	

10. Validation features implemented:

- To avoid erroneous inputs from the user, I have made the most use of field features like choice fields and dropdowns. Doing so not only makes the user interface more intuitive and to avoid input errors from the user. These make it easier for the user to input data and avoid the need of a lot validation features in the code.
- Validation features by the website page:

Homepage:

WHAT?	INPUT TO TEST	EXPECTED	ACTUAL	PASSED?

Staff	No Selection Selecting a staff	- Prompt to fill - Success	BOTH as expected	PASS
Period Time	No Selection Select Lesson	- Prompt to fill - Success	BOTH as expected	PASS
Date	No Selection Select Date	- Prompt to fill - Success	BOTH as expected	PASS
Practical Booking	No Selection Select Practical	- Prompt to fill - Success	BOTH as expected	PASS
Room No.	No Selection Select Room	- Prompt to fill - Nothing	BOTH as expected	PASS
No. of Students	No Selection String/Char Integer	- Prompt to fill - Prompt to change types - Success	- As Needed - Nothing - Success	Pass Fail Pass

Add Practical:

In the quantity fields of this page, I utilized **Django positive integer fields**. These fields are essentially dropdowns of positive integers and therefore, do not allow the user to enter illegal data. Thus, these help to reduce the need to encoded validation.

The name of the practical is a string which is entered by the user and the only validation checks run on this are presence check – to check if it isn't blank – and length check – to see if it has more than 0 characters and less than 255. **Both these validation tests were successful.**

Finally, the equipment name is a dropdown from where the user can select equipment that is added in the inventory. This again eliminates the need for any validation because the user is only selecting something. The only validation check needed is **presence check to see if something is selected and it works successfully.**

WHAT?	INPUT TO TEST	EXPECTED	ACTUAL	PASSED?
Practical Name	No Selection Name as String	- Prompt to fill - Success	- Nothing - Success	Fail Pass
Equipment Name	No Selection Select Name	- Prompt to fill - Success	Both as expected	PASS
Equipment Quantity	Nothing Quantity int	- 0 default - Success	Both as expected	PASS

Edit Practical:

The validation features implemented for this page are very similar to those implemented for the previous page. This is because, the formset rendered to edit a formset is the same as that for adding one. The only difference is in how the practical is

selected. The Add Practical page required user to type a name, but the Edit practical page requires user to select a practical from a dropdown.

Report Loss/Breakage:

This page only requires two inputs:

The name of the equipment which is selected from the dropdown.

The quantity of the equipment to remove. Both these inputs required a certain amount of validation and the tests undertaken were as follows:

WHAT?	INPUT TO TEST	EXPECTED	ACTUAL	PASSED?
Equipment Name	- No Selection - Selecting Equipment	- Prompt to fill - Success	- Nothing - Success	PASS
Equipment Quantity	- Nothing - Quantity int - String	- 0 default - Success - Prompt + No change	- Success - Success - Fail + Success	PASS PASS FAIL

Add to Inventory Page:

This page was divided into two parts – to add more of an existing equipment + the part to add a totally new piece of equipment.

The inputs and the validation for the form needed to add more of an existing equipment were the SAME as those for the form needed in the loss/breakage page.

For both these pages I have maximized the use of dropdowns to make the website as intuitive and as easy as possible for the user to navigate through. **The stakeholder also found the use of dropdowns helpful because they avoided the repetitive use of validation methods and were intuitive.**

The inputs and validation for the second part of the form are as follows:

WHAT?	INPUT TO TEST	EXPECTED	ACTUAL	PASSED?
Equipment Name	Blank String	- Prompt to fill - Success	BOTH as expected	PASS
Equip. Qty.	Blank Int String	- Prompt to fill - Success - Prompt	- Prompt - Success - Nothing	PASS PASS FAIL
Location	No Selection Select Room	- Prompt to fill - Success	BOTH as expected	PASS
Image Reference	No Selection Select Practical	- Success + Default img. - Success	BOTH as expected	PASS

Register:

This part of the website required a fair bit of validation.

- Username – Presence check, length check
- Password – Length Check, Format Check, Presence Check
- Email – Presence check, format check
- Password2 – Presence check, format check

WHAT?	INPUT TO TEST	EXPECTED	ACTUAL	PASSED?
Username	Blank String	- Prompt to fill - Success	BOTH as expected	PASS
Email	Blank Without '@' Without '.___' Correct email	- Prompt to fill - Invalid email - Invalid Email - Success	All as expected	PASS
Password1	Less than 8 chars Strange symbols like ¦ Same as username Valid password	- Prompt to change - Prompt to change - Invalid Prompt - Success	All as expected	PASS
Password2	Different from password1 Same as password1	- Prompt - Success	BOTH as expected	PASS

Login:

Lastly, with the login, the only validation really needed is that the username and password pair must match that stored in the database.

Apart from the above, the other two pages of the website were just the ones which had table to view booking history and inventory equipment. Therefore, they did not require any validations.

In conclusion, throughout this stage of the development process, I have strived to test the features of the application and implement validation features. The amount of validation features I could implement was limited because of me striving to enhance user experience and ease navigation across the website by implementing widgets like dropdown menus. These are overall more intuitive than textboxes and clearer for the user to comprehend because they limit the number of inputs that the user can enter. Overall, this has the effect of providing a simpler and more enhanced user experience which is free from prompts caused due to invalid inputs.

(4) EVALUATION (20 MARKS)

(I) TESTING TO INFORM EVALUATION

(a) Provide annotated evidence of testing the solution of robustness at the end of the development process.

(b) Provide annotated evidence of usability testing (user feedback).

(III) DESCRIBE THE FINAL PRODUCT

The testing to inform evaluation is present in Iteration 8 of the previous section.

The application was then sent to Naveen Begwani – second stakeholder who volunteered for testing the application due to their experience in using similar applications – for some usability testing. Naveen's feedback is as follows:

"

- **User Interface:** The overall design of the application looks very simple to use and easy to navigate. I find the layout extremely easy to navigate through and the color scheme is also matching that of the school which is a nice touch. Overall, some of the stylistic elements of the website could be made even more attractive and the subtle feature like the alignment of form fields and buttons styles could be improved.

- **Homepage:** The calendar is a helpful reference tool for this application, and I can foresee this as being helpful when making bookings. The usability of the calendar could be expanded further though. For example, it will be useful to be able to directly view all the bookings made on the calendar itself and it would be amazing details of the form could be filled directly by selecting a lesson on the calendar. Overall, the calendar is a good addition to the website and could be a key feature to build on in the future. Next, the form on the homepage looks good and seems to have all the necessary details that a lesson could require. The layout also looks decent and simple to use. I particularly like the use of dropdown menus because of how easy they make it to fill in the form. In many websites, blank fields cause a lot of errors when filling in and I like how you have avoided them by using dropdown menus. Lastly, the email system is great and the email is clean and concise which is nice.

- **Add/Edit Practical Pages:** Very simple to use these pages! One useful feature would be to invalidate the practical if NO equipment is added to a new practical. Furthermore, I really like

these pages and the feature which adds dropdowns and textboxes dynamically is particularly good-looking. The styling could be better but apart from that these two pages are perfect.

- **Loss/Break Report + Add to Inventory Pages:** Again, layout is simple to navigate through and very easy to use. I particularly like the ability to either add to existing equipment or create new equipment in the Add to Inventory page. Furthermore, the field needed to upload the image is also a nice addition. Lastly, I like the fact after a successful inventory item addition the application leads me to the table where I can view details of my recent addition.

- **View Inventory + Booking History Pages:** These two pages look good from an aesthetic viewpoint and are helpful additions to the application as a whole. The addition of the ability to edit inventory item details is helpful and I would have preferred having something similar for the Booking History pages as well. Similarly, the delete equipment is also a helpful addition and I would have liked for it to be available in the booking history page as well.

Overall, I believe that this is a very sophisticated and powerful application. However, this would be a valuable addition to a school like environment with some further additions.

”

(II) SUCCESS OF THE SOLUTION

(a) Use the test evidence from the development and post development process to evaluate the solution against the success criteria from the analysis.

Successfully login to the system	Successfully being able to login in the system using the school credentials.	Y
Physics Department/Any required calendar visible on the homepage	Successfully displaying a required calendar on the homepage of the application.	Y
Functional User Interface elements and simple layout of the application	Navbar of the application works properly, user can navigate between all pages of the application easily. The layout of all the UI elements like textboxes, dropdowns, buttons is simple and can be navigated easily.	Y
Ability successfully to add a practical experiment	Successfully adding a practical to the database. This will involve selecting a piece of equipment/creating a new equipment, selecting quantity of this equipment, confirming the addition, being able to visualise this change in the database, and receiving an output that confirms the amendment.	Y
Ability to amend an existing practical	This involves, successfully entering the page to make the change, select the name of the practical, make the needed change – this could be the name, equipment, or the quantity – save it, receive a confirmation, and view it in the database table.	Y
Delete an existing practical	This involves successfully selecting a practical from a drop-down menu, deleting it, receiving a confirmation and viewing it in the database.	Y
Add new stock	Select an equipment or type in a new equipment, add the quantity, and be able to view the change in the stock/inventory database.	Y
Report a loss or breakage	This involves successfully selecting a piece of equipment, entering quantity of the broken equipment, clicking the button which then reduces the quantity of the equipment from the school stock/inventory, and this change should be visible in the database that the viewer views.	Y
Viewing the details of a selected equipment in the inventory	This involves viewing quantity, availability and bookings of an item selected by the user from the inventory.	Y
Edit details of selecting an item from inventory	Successfully editing the details of a selected item in the inventory. This includes the name, qty, location, and image reference.	Y

Successfully calculate the total equipment needed to be ordered	This involves, selecting the name of the practical, adding number of teacher and student sets, calculating the total equipment needed.	Y
Successfully make a list that needs to be emailed to the technician	This involves making a list of all the equipment that was calculated and placing them like a message.	Y
Successfully emailing the created list to the technician	This involves successfully sending an outlook email of the list created to the technician.	Y
Select a lesson in the calendar on the homepage	This involves clicking a lesson, the name of the room, and the number of students automatically appearing on the enter information section, and the request being placed accordingly from there.	N
Adding a tag to the application's calendar	This involves successfully adding a 'tag' to the app's calendar This will show all details of the booking – teacher name, room number, lesson time, practical name, equipment details - when clicked on. It will be visible to the whole of physics department so that clashes maybe avoided. Clicking will give a chance to edit/ amend booking.	N
Successfully Edit/modify bookings using a tag	This involves successfully being able to amend a booking made by clicking on the calendar tag for that particular booking.	N
Viewing a comprehensive booking history	This involves successfully being able to view the booking history.	Y
Log out	This involves successfully logging out of the application once the logout button is clicked.	Y

Evidence for all the above success criteria being met is in Iteration 8 of the previous section.

(IV) MAINTENANCE AND DEVELOPMENT

(a) Discuss the maintainability of the solution.

(b) Discuss potential further development of the solution.

LIMITATIONS:

There are certain limitations in the final solution of the project. These include:

- A chat-system for the teachers and other members of staff to communicate using.
- A fully functional calendar. Initially, I had hoped to implement the calendar such that staff members could select a lesson ON the calendar and its details such as the time, date, teacher, number of students would be automatically retrieved by the app. Due to the calendar being a shareable Microsoft Outlook link I wasn't able to find a way to manipulate it to my needs effectively in the given time. Furthermore, I wasn't able to implement a 'tag' system on the calendar as envisioned initially. I wanted that to be a system to mark and store booking details such that they could be accessed by anyone using the application. Currently, the calendar's function is limited to that of a reference, which allows the staff members to simply look at their timetable and avoid making the wrong booking.
- Scheduling system for bookings. This was one of the biggest limitations of the solution! The stakeholder requested for a system to show how equipment at a particular time was not available due to other bookings needing those pieces of equipment. Due to shortage of time, I was not able to implement a 'first come first served' scheduling algorithm such that the application would keep track of the number of equipment lent out and therefore, it would staff, to know if equipment is available, when making a booking.
- Another major limitation was administrator level control to the technician. Although not a stakeholder requirement, I felt that a system for the technician to approve all additions to the database, confirm receipt of all equipment, etc., would make the application more robust and fulfil its purpose even more thoroughly.
- Lastly, due to time constraints I was unsuccessful in implementing the functionality that allows the user to modify booking details – this I believe is the first thing that I will add should I get more time.

AVOIDING LIMITATIONS:

With majority of the above limitations, they could be accomplished with more time for development.

- Calendar: This could be implemented using an external calendar rather than an Outlook one. Events/Lessons could then be added to that calendar using the .ics calendar links. These calendars would be easier to manipulate, receive data from and send data to because they won't be bound by Outlook rules.
- Modifying the booking details would be done in a similar method to modifying the details of an inventory equipment and again, this needed more time.
- Furthermore, the scheduling for the booking equipment could be developed using the querysets in Django. These could be used to store the equipment that the user has booked and then a temporary subtraction from the inventory for the given booking time can be used to check if equipment WILL be available at that time.

MAINTABILITY + FURTHER DEVELOPMENT:

In terms of the maintenance of this website, the code is full of comments with each comment providing details about what each section and module of the code does. Future maintenance could involve the use of a more sophisticated scheduling algorithm for bookings and to avoid clashes for equipment. The user interface of the application could be improved drastically to make it further appealing for the users.

Furthermore, based on stakeholder requests, I could add a calendar which is dynamic and changes based on the user logging in. To further develop this part – I need to link the website to the school's server and the Microsoft suite of applications. This will help in accessing calendars of the users logging in. Connection to the school servers is necessary so that teachers can make bookings using any school system.

In the future, I believe that this project will be a very good idea for a mobile app. This would allow teachers and staff members to make bookings from their mobiles away from the school server as well.

For the most part, the code is divided into intuitive files each of which is self-explanatory. The code is also well annotated with comments making expandability much easier.

Other features could include a notebook system and chat system like in labspace.io from the research section of the coursework. This would help the staff communicate about lessons in advance and would also help them to store details like practical methods in the application.

THE CODE (FOR ALL THE BACKEND PROCESSING):

VIEWS.PY FILE

```
from django.shortcuts import render, redirect
from django.http import HttpResponse
from .models import Inventory_Equipment, Practical, Practical_Equipment_Needed, Room, Staff, Lesson,
 Period
from .forms import Add_Inventory_Form, Remove_Inventory_Form, Add_Practical_Formset, New_Practical_F
orm, Select_Practical_Form, Book_Lesson_Form, CreateUserForm
from django.forms import inlineformset_factory
from django.contrib import messages
from django.core.mail import send_mail
import time
from django.contrib.auth import authenticate, login, logout
from django.contrib.auth.decorators import login_required
from django.contrib.auth.forms import UserCreationForm

# Create your views here.

# function to register a new user
def register(request):
    if request.user.is_authenticated:   #if user is already logged in
        return redirect('/')            # takes user to homepage
    else:
        form = CreateUserForm()         #empty registration form

        if request.method == 'POST':
            form = CreateUserForm(request.POST)
            if (form.is_valid()):       # checks if email and passwords, etc., meet the criteria
                form.save()
                username = form.cleaned_data.get('username')
                messages.success(request, 'User Created Successfully for ' + username)  # success me
ssage
                return redirect('login')    #redirects to the login page so that new user can login
to website
    return render(request, 'registration/register.html', {'form': form})

#function to login existing users
def loginPage(request):
    if request.user.is_authenticated:
        return redirect('/')
    else:
        if request.method == 'POST':
            username = request.POST.get('username')
            password = request.POST.get('password')
            user = authenticate(request, username=username, password=password)  # used to compare us
ername and password to that in database
            if user is not None:    #checks if user is not blank
                login(request, user)
                return redirect('/')    #redirects to homepage
            else:
                messages.info(request, 'username or password is incorrect')

    return render(request, 'registration/login.html', {})
```

```
#function to logout user
def logoutUser(request):
    logout(request)         #logouts user when button clicked
    return redirect('login')

#########################################################################################################
######################

# homepage function needed to book a practical
@login_required(login_url = 'login')
def homepage(request):
    practical_list = Practical.objects.all()
    room_list = Room.objects.all()
    period_numbers = Period.objects.all()
    staff_list = Staff.objects.all()
    # get staff from the person the name of user who has logged in

    if request.method == "POST":
        form = Book_Lesson_Form(request.POST)
        if form.is_valid():
            booking_details = {
                'staff': '',
                'period_time': '',
                'date': '',
                'practical_booking': '',
                'room': '',
                'number_students': '',}

            staff_name = form.cleaned_data.get('staff')
            #   get the data from form
            staff_obj = Staff.objects.filter(staff_name=staff_name)[0]
            #   get the object of that data from the respective table
            booking_details['staff'] = staff_obj
            #   add the object from above to the dictionary

            period_number = form.cleaned_data.get('period_time')
            period_obj = Period.objects.filter(period_number= period_number)[0]
            booking_details['period_time'] = period_obj

            booking_date = form.cleaned_data.get('date')
            booking_details['date'] = booking_date

            practical_to_book = form.cleaned_data.get('practical_booking')
            practical_booking_obj = Practical.objects.filter(practical_name=practical_to_book)[0]
            booking_details['practical_booking'] = practical_booking_obj

            room_to_book = form.cleaned_data.get('room')
            room_obj = Room.objects.filter(room_name = room_to_book)[0]
            booking_details['room'] = room_obj

            booking_details['number_students'] = form.cleaned_data.get('number_students')

            # Add checks to see if a booking already exists
            if (Lesson.objects.filter(date=booking_details['date'], period_time=booking_details['per
iod_time'], room=booking_details['room']).exists()):
                messages.info(request, 'A Lesson in this room for this time and date exists!')
            elif (Lesson.objects.filter(staff=booking_details['staff'], date=booking_details['date']
, period_time=booking_details['period_time']).exists()):
```

```
                messages.info(request, 'Selected teacher will be busy during that time')
            else:
                # add clause such that it saves only if total calculated equipment is available.
                equipment_names, equipment_quantities = calculate_total_equipment(booking_details)

                # practical_id = int(Practical.objects.filter(practical_name = practical_to_book).va
lues('id')[0]['id'])
                # inventory_equipment_id = Practical_Equipment_Needed.objects.filter(practical_id=pr
actical_id).values('equipment_needed_id')
                # for i in range (0, len(equipment_quantities)):
                #     if equipment_quantities[i]>inventory_equipment_id[i]:
                #         messages.info(request, 'Inufficient Equipment in the inventory')
                #     else:
                #         break

                message = create_message(equipment_names, equipment_quantities, booking_details['roo
m'])
                email_message(message)
                Lesson.objects.create(
                    staff=booking_details['staff'],
                    period_time=booking_details['period_time'],
                    date=booking_details['date'],
                    practical_booking=booking_details['practical_booking'],
                    room=booking_details['room'],
                    number_students=booking_details['number_students'])

    else:
        form = Book_Lesson_Form()
    return render(request, 'main/HomePage.html', {'form': form, 'practical_list':practical_list, 'ro
om_list': room_list, 'period_numbers': period_numbers, 'staff_list': staff_list})

# function to calculate the total equipment needed to perform the practical
def calculate_total_equipment(booking_details):
    number_students = booking_details['number_students']    #var to store number of students
    practical_to_book = booking_details['practical_booking']    #var to store the practical to be do
ne
    practical_equipment = 0
    equipment_names_list = []   #list to store names of all equipment in the practical that user wan
ts to do
    equipment_quantities_list = []  #list to store quantities of all the equipment in the above list
    practical_id = int(Practical.objects.filter(practical_name = practical_to_book).values('id')[0][
'id']) #id of practical
    equipments = Practical_Equipment_Needed.objects.filter(practical_id=practical_id).values('equipm
ent_needed_id')
    #list of dictionaries that stores ID of each equipment

    # for loop needed to retrive name of equipment and their quanities
    for i in range (0, len(equipments)):

        # creating lists which store the name and quanities of each equipment needed in a practical
        equipment_name = Inventory_Equipment.objects.get(id = equipments[i]['equipment_needed_id']).
name
        equipment_names_list.append(equipment_name)

        equipment_quantities = Practical_Equipment_Needed.objects.get(equipment_needed_id=equipments
[i]['equipment_needed_id'], practical_id=practical_id).equipment_quantity
        equipment_quantities_list.append(equipment_quantities)
```

```
    print (equipment_names_list)        #check print
    print (equipment_quantities_list)   #check print

    total_equipment_needed = [i * number_students for i in equipment_quantities_list]   #list to sto
re the total quantity of each equipment
    print (total_equipment_needed)      #check print

    return total_equipment_needed, equipment_names_list

# Function to create a message that needs to go to the technician
def create_message(equipment_names, equipment_quantities, room_number):
    message_text = 'Room Number '+ str(room_number)+'\n'    #initial string with string of the messa
ge
    for i in range (0, len(equipment_names)):   # cycling through the equipmetn lists to add them to
 the message
        name_quantity_pair = ''
        name_quantity_pair = str(equipment_names[i]) + ' ' + str(equipment_quantities[i]) + '\n'
        message_text += name_quantity_pair

    print (message_text)
    return message_text

    '''
        The message looks something like this:

        Room Number B101
        10  Testtubes
        15  Rulers
        20  Clamps
        30  Bosses

        ##  I still need to add the time and date of the lesson in this message!
    '''

def email_message(message):
        send_mail(
            'Practical Request',
            message,
            '14pbegwani@patesgs.org',
            ['pranaybegwani@gmail.com'],
            fail_silently=False,
        )

#function to delete a certain quantity of the selected equipment_name
@login_required(login_url = 'login')
def report_loss(request):
    equipment_name = Inventory_Equipment.objects.all()  # querys all the data from the inventory tab
le into a queryset - used to render the dropdown
    if request.method == "POST": #checks if data is being SENT to template
        form = Remove_Inventory_Form(request.POST) #holds all the information from our form.
        # When submit is clicked this gets a dictionary of all attributes and inputs, creates a new
form with all values you entered.
        if form.is_valid(): #check if data entered in the form meets the constraints for forms.py fi
le
            if (form.cleaned_data.get("quantity_to_remove") is not None): # use is not null instead
                equipment_name_selected = form.cleaned_data.get("equipment_name")
                # below line querys the quantity existing in the database
```

```
                exisitng_quantity = Inventory_Equipment.objects.filter(name = equipment_name_selecte
d).values('total_quantity')[0]['total_quantity']
                new_quantity = exisitng_quantity - form.cleaned_data.get("quantity_to_remove")  #sub
tract the quantity
                Inventory_Equipment.objects.filter(name = equipment_name_selected).values('total_qua
ntity').update(total_quantity = new_quantity)
            return redirect('/ViewInventory')
            # link to a page which shows the full table inventory
    else: #basically if method is GET
        form = Remove_Inventory_Form()
    return render(request, 'main/LossOrBreakReport.html', {'form': form, 'equipment_names': equipmen
t_name})

@login_required(login_url = 'login')
def add_new_to_inventory(request):
    equipment_name = Inventory_Equipment.objects.all() #gets all the equipment stored in the invento
ry table of the db
    room_list = Room.objects.all()

    if request.method == "POST": # checks if data is being SENT/POSTED to the template

        form = Add_Inventory_Form(request.POST, request.FILES) #holds all the information from our f
orm.
        #When submit is clicked this gets a dictionary of all attributes and inputs, creates a new f
orm with all values you entered.

        if form.is_valid(): #built-in menthod to verify the constraints in forms.py file

            if (form.cleaned_data.get("new_quantity") is not None): # checks if the field to ADD to
existing qty is blank or not
                equipment_name_selected = form.cleaned_data.get("existing_name")
                print(equipment_name_selected)  #for debugging purposes
                exisitng_quantity = Inventory_Equipment.objects.filter(name = equipment_name_selecte
d).values('total_quantity')[0]['total_quantity']
                print(exisitng_quantity)    #for debugging purposes
                new_quantity = exisitng_quantity + form.cleaned_data.get("new_quantity")    #adds th
e user entered qty to exisitng_qty
                Inventory_Equipment.objects.filter(name = equipment_name_selected).values('total_qua
ntity').update(total_quantity = new_quantity)
                print (new_quantity)    #for debugging purposes
            #add else here and check
            else:
                form.save() # if the qty to add is blank, saves the botton part of the form, creatin
g a new record in database
            return redirect('/ViewInventory')   #displays the new record in a table - this was added
 later
    else: #basically if method is GET
        form = Add_Inventory_Form()
    # below line contains the context being sent to the template AddToInventory.html
    return render(request, 'main/AddToInventory.html', {'form': form, 'equipment_names': equipment_n
ame, 'room_list' : room_list})

# function to get all records from the inventory table and store them in a context dictionary
@login_required(login_url = 'login')
def view_inventory(response):
    inventory_obj = Inventory_Equipment.objects.all()   #querys every single record into this identi
fier
```

```
    return render(response,"main/ViewInventory.html", {'inventory_obj': inventory_obj}) #template re
ndered & inventory_obj passed as context to template

# in both the functions below, id is the ID for the item in inventory to be edited

# function to get the record that needs to be edited and pass it to the template
@login_required(login_url = 'login')
def edit_inventory(request, id):
    inventory_item = Inventory_Equipment.objects.get(id = id) # get the item with the given id from
the inventory table
    return render(request,'main/EditInventoryDetails.html', {'inventory_item': inventory_item})

# function to edit the exisiting item
@login_required(login_url = 'login')
def update(request, id):
    inventory_item = Inventory_Equipment.objects.get(id = id)  # get the item with the given id from
 the inventory table
    if (request.method == 'POST'):  # if data is being sent to the server/POSTED to server
        # below line renders the form, Add_Inventory_Form with data already filled in - the data is
the instance
        form = Add_Inventory_Form(request.POST or None, request.FILES or None, instance = inventory_
item)
        if form.is_valid(): #if the form meets validation criteria
            form.save() #save updated details to the database
        return redirect("/ViewInventory")   # once completed lead the inventory table page
    else: #basically if method is GET
        form = Remove_Inventory_Form()  #empty form if GETTING the page from database
    return render(request, 'main/EditInventoryDetails.html', {'inventory_obj': inventory_item })

# function to delete exisiing inventory item from the table
@login_required(login_url = 'login')
def delete_inventory_item(request, id):
    equipment_delete = Inventory_Equipment.objects.get(id = id) # stores the quereyset of the item t
hat needs to be deleted
    equipment_delete.delete()   # deletes the item
    return redirect('/ViewInventory')   #redirects to the inventory page

    return render(request,'main/DeleteEquipment.html', {})

# function to create new empty practical
@login_required(login_url = 'login')
def name_new_practical(request):
    practical = Practical.objects.all()

    if (request.method == 'POST'):  #if connection method is POST
        form = New_Practical_Form(request.POST) #creates new form to send data to server
        if form.is_valid():
            new_name = form.cleaned_data.get('name_new_practical')  #gets name of the new practical
in the variable
            if (Practical.objects.filter(practical_name = new_name)).count() != 0:
                messages.error(request, 'This Practical Exists')
                return redirect('/NameNewPractical')
                print ('Exists')
            else:
                new_practical = Practical.objects.create(practical_name = new_name)    # creates new
 practicalin the database
                new_id = Practical.objects.filter(practical_name= new_name).values('id')[0]['id']
#gets id of the newly added practical
```

```
        print (str(new_id))
        practical = Practical.objects.get(pk = new_id)  # querys the new practical using its ID to p
ass to template
        return redirect('/AddPractical/%d'%new_id)  #once all tasks complete, leads to the AddPracti
cal page to enter details of new practical
    else:
        form = New_Practical_Form() # if the method is GET, displays the empty form
    return render(request, 'main/NewPractical.html', {'form': form, 'practical': practical})

# function to add details of newly added practical to database
@login_required(login_url = 'login')
def add_new_practical(request, id): # ID of the newly added practical is a parameter so that equipme
nt is added for that practical only

    if (Practical_Equipment_Needed.objects.filter(practical_id = id).count()) == 0: # checks if the
practical already exists in the database
        new_practical_details = Practical_Equipment_Needed()    # if it doesn't exist, creates empty
 object for details

    if (request.method == 'POST'):  # if method is POST
        formset = Add_Practical_Formset(request.POST)   # creates new empty formset to add details f
or the selected practical
        if formset.is_valid():
            instances = formset.save(commit=False)  # saves data but doesn't send it to the database
            for instance in instances:  # querys through every individual instance of the formset
                instance.practical_id  = id
                instance.save() #saves individual instance to the Practical_Equipment_Needed databas
e table
            return redirect('/AddPractical/%d'%id) #return to the same page after save to be able to
 add more equipment

    formset = Add_Practical_Formset(queryset = Practical_Equipment_Needed.objects.filter(practical__
id = id))   # empty formset

    return render(request, 'main/AddNewPractical.html', {'formset': formset})

# function to enable user to choose the practical whose details they want to edit
@login_required(login_url = 'login')
def select_practical_to_edit(request):
    practical_names = Practical.objects.all()   # storing objects of all practical to render names i
n the dropdown on webpage

    if (request.method == 'POST'):  # if POSTING data to the backend
        form = Select_Practical_Form(request.POST)  # rendering the Select_Practical_Form needed to
take input of practical name
        if form.is_valid(): # checking if form meets validation constraints
            selected_practical_name = form.cleaned_data.get('name_practical')   # storing the name s
elected by user in a variable

        id_selected  =  Practical.objects.filter(practical_name= selected_practical_name).values('id
')[0]['id'] # getting id of the selected practical
        return redirect('/EditPractical/%d'%id_selected)    # redirecting user to the page needed to
 edit the details of the selected practical

    else:   # if GETTING data/information
        form = Select_Practical_Form()  # displays an empty form
```

```
    return render(request,'main/SelectPracticalToEdit.html', {'form': form, 'practical_names': pract
ical_names})    #renders webpage and sends data to template

# function to edit details of practical selected using select_practical_to_add
@login_required(login_url = 'login')
def edit_practical(request, id):
    practical_names = Practical.objects.all()   # storing all practical objects in a queryset

    if (request.method == 'POST'):
        formset = Add_Practical_Formset(request.POST)   # renders the formset to allow user to edit
practical selected
        if formset.is_valid():
            instances = formset.save(commit=False)  # commit is false so that the user can make mult
iple additions at the same time
            for instance in instances:  # cycle through all instances and save them individually to
database
                instance.practical_id = id  # making sure instance is saved for the practical with t
h correct id
                instance.save()
            for object in formset.deleted_objects:
                object.delete()
        return redirect('/EditPractical/%d'%id) # leads to the same page to allow user to add/edit m
ore details

     # displaying formset with existing details of practical
    formset = Add_Practical_Formset(queryset = Practical_Equipment_Needed.objects.filter(practical__
id = id))

    return render(request, 'main/EditPractical.html', {'formset': formset, 'practical_names': practi
cal_names})

# function to get all the bookings made
@login_required(login_url = 'login')
def booking_history(response):
    lesson_obj = Lesson.objects.all()   #querys every single record into this identifier
    return render(response,"main/BookingHistory.html", {'lesson_obj': lesson_obj}) #template rendere
d & lesson_obj passed as context to template
```

MODELS.PY

```
from django.db import models


# class to represent the INVENTORY table in the database
class Inventory_Equipment(models.Model):
    #id = models.IntegerField(primary_key=True, null=False, blank=True)
    # The above field now not in the code so that the records for each equipment are unique
    name = models.CharField(
        max_length=255,     #maximum length of input
        blank=True,         #the field can be blank
        null=True,          #the field can be null
        default="Name this Equipment")  # the defualt value if it is left blank
    total_quantity = models.PositiveIntegerField(
        null=False,
        blank=False,
        default=0)
    location = models.CharField(
        max_length=20,
```

```
        default='Physics Office',
        blank=True)
    img_reference = models.ImageField(
        null=True,
        blank=True,
        upload_to='images_uploaded/',   #the location of where to store uploaded images
                                        #currently the location is local device
        default='images_uploaded/default.jpg')  #the default image stored if no image is uploaded by
 user

    class Meta:
        db_table="inventory"    #name of the table in the database to which this model links to

    def __str__(self):
        return self.name    #when queryed, the 'name' is returned

    #add validation functions - validation is a step i will do after barebones of the program are ma
de

# Table to store list of all the practical - just their names
class Practical(models.Model):
    practical_name = models.CharField(
        max_length=100,
        blank=True)
    equipment_needed = models.ManyToManyField(  # this is used to link to the link table to normaliz
e the many to many relationship
        'Inventory_Equipment', # name of the table to which this has a many to many relationship wit
h
        blank=True,
        through='Practical_Equipment_Needed')   # name of the link table to normalize the many to ma
ny relationship

    class Meta:
        db_table="practical"

    def __str__(self):
        return self.practical_name

# Table to store the equipment needed in every practical and their quantities
class Practical_Equipment_Needed(models.Model):
    equipment_needed = models.ForeignKey(   # a foreign key of the inventory table to store the equi
pment
        'Inventory_Equipment', # table of which this is a foreign key
        on_delete=models.CASCADE)   # needed to reflect any deletions all across the database
    practical = models.ForeignKey(  #foreign key of the Practical table
        'Practical',
        on_delete=models.CASCADE)
    equipment_quantity = models.PositiveIntegerField(   # integer field needed to store the quantity
 of an equipment
        default=0)  # the quantity if no number is entered by the user

    class Meta:
        db_table="practical_equipment_needed"

    def __str__(self):
        return self.practical.practical_name

# Model for table with names of all staff members
```

```
class Staff(models.Model):
    staff_name = models.CharField(max_length=255)# something like a dropdown of the staff names from
 the database

    class Meta:
        db_table="Staff"

    def __str__(self):
        return self.staff_name

# Model for the table of all the rooms in the school science dept.
class Room(models.Model):
    room_name = models.CharField(max_length=30)#something like a dropdown of all rooms from the data
base...

    class Meta:
        db_table="Room"

    def __str__(self):
        return self.room_name

# Model for the lesson time (1 - 5)
class Period(models.Model):
    PERIOD_NUMBER_CHOICES = [
            ('Lesson 1','Lesson 1'),
            ('Lesson 2','Lesson 2'),
            ('Lesson 3','Lesson 3'),
            ('Lesson 4','Lesson 4'),
            ('Lesson 5','Lesson 5'),
        ]

# choice field because I need this to be a dropdown which holds the above 5 ooptions - it is a const
ant
    period_number = models.CharField(
                                    choices=PERIOD_NUMBER_CHOICES,
                                    max_length = 10)

    class Meta:
        db_table = 'period_number'

    def __str__(self):
        return self.period_number

# Model which stores the booking details if a booking is made
class Lesson(models.Model):
    staff = models.ForeignKey(Staff, on_delete=models.CASCADE, default= 1, null=True, blank=True)
    # name of staff who is booking a lesson - foreign key of the staff model
    period_time = models.ForeignKey(Period, on_delete=models.CASCADE)
    # period number of the booking - foreign key of the Period model
    date = models.DateField()
    # date of the booking - needed to check for clashes in bookings
    practical_booking = models.ForeignKey(Practical, on_delete=models.CASCADE)
    # Foreign Key for Practical model - needed to store the name of the practical that the user want
s to book
    room = models.ForeignKey(Room, on_delete=models.CASCADE)
    # room number that the member of staff wants to book - foreign key of the Room model
    number_students = models.PositiveIntegerField(default = 0)
    # number of students that will be doing the practical
```

```
    class Meta:
        db_table = 'Lesson_Bookings'
```

FORMS.PY

```
from django import forms
# importing the forms module from django
from .models import Inventory_Equipment, Practical, Practical_Equipment_Needed, Lesson, Staff, Room,
 Period
# importing all the models created in models.py
from django.forms import modelformset_factory
# importing library to use input validation in django
from django.core import validators
from django.contrib.auth.forms import UserCreationForm
from django.contrib.auth.models import User


class CreateUserForm(UserCreationForm):
    class Meta:
        model = User    #built in model inside a django databse needed to store all users
        fields = ['username', 'email', 'password1', 'password2', ]  #the fields that a user needs to
 fill in to register themselves

#############################################################################################

class Add_Inventory_Form(forms.ModelForm):
    new_quantity = forms.IntegerField(  #field to enter the qty needed to be added to existing equip
ment
        required = False,
        widget=forms.HiddenInput(),
        initial=0)
    existing_name = forms.CharField(    #field to select the equipment to add qty to
        max_length=255,
        required = False)
    class Meta:
        model = Inventory_Equipment     #name of the model that this form represents
        fields = "__all__"              #means that all attributes of the model named above will hav
e a field

#form to report loss of equipment from inventory
class Remove_Inventory_Form(forms.ModelForm):   #name of the form to remove equipment
    quantity_to_remove = forms.IntegerField(    #integer field for the quantity that is to be remove
d
        required = True,
        widget=forms.HiddenInput(),
        initial=0,
        min_value=0)
        # max_value=Inventory_Equipment.objects.filter(name = equip_nam).values('total_quantity')[0]
['total_quantity'],   #fix this
    equipment_name = forms.CharField(   #name of the equipment to be removed from - this is rendered
 as a dropdown
        max_length=255,
        required = True)
    class Meta:
        model = Inventory_Equipment     #name of the model that this form is connected to
        fields = "__all__"              #means that all attributes of the model named above will hav
e a field
```

```
class New_Practical_Form(forms.Form):
    name_new_practical = forms.CharField(
        max_length=100,
        widget=forms.TextInput(attrs={'class' : 'text-box'}))

# form to input a single equipment and its quantity for a practical
class New_Practical_Detail_Form(forms.ModelForm):   # declaring the form
    class Meta:
        model = Practical_Equipment_Needed      #the model that this form is connected to
        fields = ('equipment_needed', 'equipment_quantity', )   # fields needed in the form

# formset which creates multiple instances of the form above
Add_Practical_Formset = modelformset_factory(   #declaring the formset
    Practical_Equipment_Needed,                 #the models that this formset is connected to
    New_Practical_Detail_Form,                  #the form (above) whose copies this formset creates
    can_delete=True
)

#Form to select the practical that the user wants to edit
class Select_Practical_Form(forms.Form):
    name_practical = forms.CharField(max_length=255)

# Form to book a lesson in a particular room for some number of students
class Book_Lesson_Form(forms.Form):
    staff = forms.ModelChoiceField(
        queryset=Staff.objects.all(),
        empty_label='Select',
        widget=forms.Select(attrs={'class' : 'small-drop-downs'}))
    # dropdown to select staff name
    period_time = forms.ModelChoiceField(
        queryset=Period.objects.all(),
        empty_label='Select',
        widget=forms.Select(attrs={'class' : 'small-drop-downs'}))
    # dropdown to select choice of lesson
    date = forms.DateField(widget=forms.widgets.DateInput(
        attrs={'type': 'date', 'id': 'date-widget'}))
    # date field that display a calendar allowing the user to select a specific date
    practical_booking = forms.ModelChoiceField(
        queryset=Practical.objects.all(),
        empty_label='Select',
        widget=forms.Select(attrs={'id' : 'practical-list'}))
    # dropdown to select what practical to boo for
    room = forms.ModelChoiceField(
        queryset=Room.objects.all(),
        empty_label='Select',
        widget=forms.Select(attrs={'class' : 'small-drop-downs'}))
    # dropdown to select room number to book
    number_students = forms.IntegerField(
        widget=forms.TextInput(attrs={'class' : 'text-boxes'}))
    # integer field to enter the number of students in the lesson being booked
```

URLS.PY – THE WEBSITE

```
from django.urls import path
from django.conf import settings
from django.conf.urls.static import static
```

```
from . import views # . is from 'this app' import views

urlpatterns = [
     path("", views.homepage, name="home"),
     #name must be same as the function name in the views.py.
     #The path('') in the urls.py of mysite looks for the same path in '' and then renders the page
which follows that
     path("LossReport/", views.report_loss, name="loss_report_page"),
     path("AddInventory/", views.add_new_to_inventory, name="add_new_to_inventory"),

     # below is link to the page that helps adding new equipment to the inventory
     path("AddPractical/<int:id>", views.add_new_practical, name="add_new_practical"),
     # below is link to the page that allows user to create a new practical. This page leads to the
above url
     path("NameNewPractical/", views.name_new_practical, name="name_new_practical"),

     # the url which lead to the page needed to select a practical to edit
     path("SelectPractical/", views.select_practical_to_edit, name="select_practical_to_edit"),
     # the url which leads to the page to edit details of the practical with id - id - in the Practi
cal table
     path("EditPractical/<int:id>", views.edit_practical, name="edit_practical"),

     path("ViewInventory/", views.view_inventory, name="view_inventory"),

     # new urls added to be able to edit the inventory equipment - based on their ID
     path("EditInventory/<int:id>", views.edit_inventory, name="edit_inventory"),
     path("Update/<int:id>", views.update, name="update"),

     path("Delete/<int:id>", views.delete_inventory_item, name="delete_equipment"),

     path("BookingHistory/", views.booking_history, name="booking_history"),

     path('login/', views.loginPage, name="login"),
     path('logout/', views.logoutUser, name="logout"),
     path('register/', views.register, name="register"),

]
```

URLS.PY – WHOLE PROJECT

```
from django.contrib import admin
from django.urls import path, include
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('inventory.urls')),
    #path('accounts/', include('django.contrib.auth.urls')),
]

urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

SETTINGS.PY

```
from pathlib import Path
import os
```

```
# Build paths inside the project like this: BASE_DIR / 'subdir'.
BASE_DIR = Path(__file__).resolve().parent.parent


# Quick-start development settings - unsuitable for production
# See https://docs.djangoproject.com/en/3.1/howto/deployment/checklist/

# SECURITY WARNING: keep the secret key used in production secret!
SECRET_KEY = 'x#a7t)gat)vo@ngwn=9ocs-bd-yx4mjl^70l28x1q^ymsauzjz'

# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = True

ALLOWED_HOSTS = []


# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'inventory.apps.MainConfig'
]

MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]

ROOT_URLCONF = 'physics_db.urls'

TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
                'django.template.context_processors.media',
            ],
        },
    },
]
```

```
WSGI_APPLICATION = 'physics_db.wsgi.application'


# Database
# https://docs.djangoproject.com/en/3.1/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',   #link to the backend database engine
        'NAME': 'physics laboratory',           #name of the database
        'HOST': 'localhost',                    #name of the host- localhost while development
        'PORT': '3306',                         #port number for connection to the server
        'USER': 'root',                         #username for connection to the database server
        'PASSWORD':'',                          #password for connection to the database server
        'OPTIONS': { # additional settings for the connection to the database server
            'init_command': "SET sql_mode='STRICT_TRANS_TABLES'"
        },
    }
}

# Password validation
# https://docs.djangoproject.com/en/3.1/ref/settings/#auth-password-validators

AUTH_PASSWORD_VALIDATORS = [
    {
        'NAME': 'django.contrib.auth.password_validation.UserAttributeSimilarityValidator',
    },
    {
        'NAME': 'django.contrib.auth.password_validation.MinimumLengthValidator',
    },
    {
        'NAME': 'django.contrib.auth.password_validation.CommonPasswordValidator',
    },
    {
        'NAME': 'django.contrib.auth.password_validation.NumericPasswordValidator',
    },
]


# Internationalization
# https://docs.djangoproject.com/en/3.1/topics/i18n/

LANGUAGE_CODE = 'en-us'

TIME_ZONE = 'UTC'

USE_I18N = True

USE_L10N = True

USE_TZ = True


# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/3.1/howto/static-files/

STATIC_URL = '/static/'
```

```
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
MEDIA_URL = '/media/'

# EMAIL_HOST = 'smtp.mailtrap.io'
# EMAIL_HOST_USER = '7d5570643d039e'
# EMAIL_HOST_PASSWORD = '8267fd98891e45'
# EMAIL_PORT = '2525'

# this was later replaced by outlook link
```

THE CODE (THE FRONT-END):

BASE.HTML

```
 {% load static %} <!-- essential to be able to add css to page -->
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>{% block title %}HomePage{% endblock %}</title>
        <!-
- The block means that the content between these tags can be different for each page that inherits f
rom base -->
        <meta charset="UTF-8">
        <link rel="stylesheet" href="{% static 'css/HomePageDesign.css' %}">
        {% block extrahead %}{% endblock %} <!-
- Extrablock is needed so that I can add CSS for individual pages that inherit-->
    </head>

    <body id="bg">
        <div class="mainWindow" style="color:black;" >

            <div id="TopBar">
                <div id="BarRight">
                    <button class="buttons" id="LogOutButton"><a href="{% url 'logout' %}">Logout</a
></button>
                    <a href="https://www.patesgs.org/"><img id="logo" alt="Pate's Logo"
                        src="{% static 'images/schoolLogo.png' %}" width="50px" height="50px"></a>
                </div>

                <div id="BarLeft"> <!-- All the individual buttons for the navbar -->
                    <a href= "/">
                        <button class="buttons" id="Home">Home</button>
                    </a>
                    <!-- <a href="/AddPractical"> -->
                    <a href="/NameNewPractical">

                        <button class="buttons" id="Add">Add Practical</button>
                    </a>
                    <a href="/SelectPractical">
                        <button class="buttons" id="Edit">Edit Practical</button>
                    </a>
                    <a href="/LossReport">
                        <button class="buttons" id="Report">Loss/Break</button>
                    </a>
                    <a href= "/AddInventory">
```

```
                        <button class="buttons" id="Inventory">Add to Inv</button>
                    </a>
                    <a href= "/ViewInventory">
                        <button class="buttons" id="View_Inventory">View Inventory</button>
                    </a>
                    <a href= "/BookingHistory">
                        <button class="buttons" id="Booking_History">Booking History</button>
                    </a>
                    </div>
            </div>

            {% block content %}
                <!-
- All the content between these tags will be unique to the page that inherits from base -->
            {% endblock %}

        </div>
    </body>
</html>
```

HOMEPAGE.HTML

```
{% extends 'main/base.html' %}
{% load static %}

<html>

    <head>
        {% block title %}HomePage{% endblock %}
        <link rel="stylesheet" href="{% static 'css/HomePageDesign.css' %}">
    </head>

    <body>
        {% block content %}
            <div class = "OrderEquipPage"> <!-- This is a row -->

                <div class = "TheForm"> <!-
- Splitting the row above into columns, this one is for taking input-->
                    <form action="#" method="POST" enctype="multipart/form-data">
                        {% csrf_token %}

                        {{ form.as_p }}
                        <!-- Renders the form which takes the booking-->

                        <button name="save" type="submit" class= "buttons" id="send-
request">Send Request</button>

                        {{ form.errors }}
                        <!-
- Used to display all the error messages in the form like clashes in bookings, etc.-->
                    </form>

                    {% if messages %}
                        <ul class="messages">
                        {% for message in messages %}
                            <li  {% if message.tags %} class=" {{ message.tags }} " {% endif %}> {{
message }} </li>
                        {% endfor %}
```

```
                    </ul>
                {% endif %}

                <p>testing</p>
            </div>

            <div id = "TheCalendar" unselectable="on"> <!-
- Splitting the row above into columns, this one is for putting the calendar -->
                <iframe id="calendar" src="https://tinyurl.com/yxf3v8pm" height="700" width="750
"></iframe>
            </div>
        </div>
    {% endblock %}
</body>
</html>
```

The other individual pages of the application are very similar and I am therefore, not adding the code over here.

LOGIN.HTML

```
{% load static %}

<html>
    <head>
        <title>Login</title>
        <!-- Inline CSS for the styling -->
        <style>
            .mainWindow{
                border-style: solid;
                border-color: red;
                border-radius: 10px;
                border-top-width: auto;
                border-bottom-width: auto;
                border-left-width: 10px;
                border-right-width: 10px;
                width: 98%;
                height: 98%;
            }

            .buttons{
                color: black;
                border-style: solid;
                border-color: grey;
                border-radius: 12px;
                font-size: 16px;
                cursor: pointer;
                margin: 5px;
                width: 100px;
                height: 50px;
                padding: 5px;
                text-align: center;
            }

            .buttons:hover{
                color: red;
                border-radius: 12px;
```

```
            }

            #login{
                margin-top: 5%;
                margin-bottom: 5%;
                margin-left: 30%;
            }

            .FormForLogin{
                border-style: solid;
                border-color: red;
                border-radius: 10px;
                border-top-width: 10px;
                border-bottom-width: 10px;
                border-left-width: 10px;
                border-right-width: 10px;
                padding: 5%;
                margin: auto;
                display: inline-block;
                position: absolute;
                font-family: arial;
            }

            .text-boxes{
                width: 170px;
                border-radius: 12px;
                background-color: transparent;
                height: 25px;
                border-width: 1px;
                border-color: black;
                border-style: solid;
                text-align: center;
            }

            #send-request{
                border-radius: 12px;
                width: 70px;
                height: 35px;
                font-size: 9px;
                text-align: center;
                margin-left: 30%;
            }

        </style>
    </head>

    <body>
    <div class='page' id='login'>
        <form class="FormForLogin" method="POST" action="">
            <h3 id="form-title">LOGIN ACCOUNT</h3>
            {% csrf_token %}
            <label>Enter Username</label>
            <input class='text-boxes' type="text" name="username"/>
            <br>
            <br>
            <label>Enter Password</label>
            <input class='text-boxes' type="password" name="password"/>
            <br>
```

```
            <br>
            <button class="buttons" id="send-request" type="submit">Login</button>
            {% for message in messages %}
            <p>{{ message }}</p>
        {% endfor %}
        <br>
        <br>
        Don't have an account? <a href="{% url 'register' %}">Sign Up</a>
        </form>
    </div>
    <script>
        //Query All input fields
        var form_fields = document.getElementsByTagName('input')
        form_fields[1].placeholder='Username...';
        form_fields[2].placeholder='Enter password...';
        for (var field in form_fields){
            form_fields[field].className += ' form-control'
        }
    </script>
</body>
</html>
```

REGISTER.HTML

```
<!DOCTYPE html>
<html>

<head>
    <title>Register</title>
    <style>
        .mainWindow{
            border-style: solid;
            border-color: red;
            border-radius: 10px;
            border-top-width: auto;
            border-bottom-width: auto;
            border-left-width: 10px;
            border-right-width: 10px;
            width: 98%;
            height: 98%;
        }

        .buttons{
            color: black;
            border-style: solid;
            border-color: grey;
            border-radius: 12px;
            font-size: 16px;
            cursor: pointer;
            margin: 5px;
            width: 100px;
            height: 50px;
            padding: 5px;
            text-align: center;
        }

        .buttons:hover{
            color: red;
```

```
            border-radius: 12px;
        }

        #login{
            margin-top: 5%;
            margin-bottom: 5%;
            margin-left: 30%;
        }

        .FormForRegister{
            border-style: solid;
            border-color: red;
            border-radius: 10px;
            border-top-width: 10px;
            border-bottom-width: 10px;
            border-left-width: 10px;
            border-right-width: 10px;
            padding: 5%;
            margin: auto;
            display: inline-block;
            position: absolute;
            font-family: arial;
        }

        .text-boxes{
            width: 170px;
            border-radius: 12px;
            background-color: transparent;
            height: 25px;
            border-width: 1px;
            border-color: black;
            border-style: solid;
            text-align: center;
        }

        #send-request{
            border-radius: 12px;
            width: 70px;
            height: 35px;
            font-size: 9px;
            text-align: center;
            margin-left: 30%;
        }

    </style>
</head>

<body>
    <!-- Below is a div section for a form -->
    <div class='page' id='login'>
        <form class='FormForRegister' method="POST" action="#">
            <h3 id="form-title">REGISTER ACCOUNT</h3>
            {% csrf_token %}
            <label>Enter Username</label>
            <input class='text-boxes' type="text" name="username"/>
            <br>
            <br>
            <label>Enter Email</label>
```

```
            <input class='text-boxes' type="text" name="email"/>
            <br>
            <br>
            <label>Enter Password</label>
            <input class='text-boxes' type="password" name="password1"/>
            <br>
            <br>
            <label>Re-Enter Password</label>
            <input class='text-boxes' type="password" name="password2"/>
            <br>
            <br>
            {{ form.errors }}
            Already have an account? <a href="{% url 'login' %}">Log In</a>
            <button class="buttons" id="send-request" type="submit">Create User</button>
        </form>
    </div>
    <script>

        //Query All input fields
        var form_fields = document.getElementsByTagName('input')
        form_fields[1].placeholder='Username..';
        form_fields[2].placeholder='Email..';
        form_fields[3].placeholder='Enter password...';
        form_fields[4].placeholder='Re-enter Password...';
        for (var field in form_fields){
            form_fields[field].className += ' form-control'
        }
    </script>
</body>
</html>
```

All the CSS files are mostly similar and I am including the homepage CSS file below:

```
.mainWindow{
    border-style: solid;
    border-color: red;
    border-radius: 10px;
    border-top-width: auto;
    border-bottom-width: auto;
    border-left-width: 10px;
    border-right-width: 10px;
    width: 98%;
    height: 98%;
}

#TopBar{
    background-color: red;
    overflow: auto;
}

.buttons{
    color: black;
    border-style: solid;
    border-color: grey;
    border-radius: 12px;
    font-size: 16px;
```

```
    cursor: pointer;
    margin: 5px;
    width: 100px;
    height: 50px;
    padding: 5px;
    text-align: center;
}

.buttons:hover{
    color: red;
    border-radius: 12px;
}

#BarRight{
    display: flex;
    float: right;
}

#BarLeft{
    display: flex;
    float: left;
}

.OrderEquipPage:after {
    display: flex;
    content: "";
    flex-direction: row;
    clear: both;
}

.TheForm{
    padding: 1%;
    margin: auto;
    margin-right: auto;
    margin-left: 1%;
    display: inline-block;
    align-self: center;
    vertical-align: middle;
    font-family: arial;
}

#practical-list{
    width: 150px;
    border-radius: 12px;
    background-color: transparent;
    height: 30px;
    border-color: black;
    border-style: solid;
    text-align: center;
}

.text-boxes{
    width: 170px;
    border-radius: 12px;
    background-color: transparent;
    height: 25px;
    border-width: 1px;
    border-color: black;
```

```
    border-style: solid;
    text-align: center;
}

.small-drop-downs{
    width: 170px;
    border-radius: 12px;
    background-color: transparent;
    height: 30px;
    border-color: black;
    border-style: solid;
    text-align: center;
}

#TheCalendar{
    padding: 3%;
    margin: auto;
    display: inline-block;
    align-self: middle;
    vertical-align: middle;
}

#calendar{
    border-radius: 12px;
    border-color: grey;
    width: 60vw;
    height: 100vh;
    position: relative;
}

#send-request{
    border-radius: 12px;
    width: 70px;
    height: 35px;
    font-size: 9px;
    text-align: center;
    margin-left: 30%;
}

#date-widget{
    width: 170px;
    border-radius: 12px;
    background-color: transparent;
    height: 25px;
    border-color: black;
    border-style: solid;
    border-width: 1px;
}
```

APPENDIX A – BIBLIOGRAPHY

Django Documentation: https://docs.djangoproject.com/en/3.2/

https://djangobook.com/mastering-django-2-book/

https://stackoverflow.com/questions/tagged/django

https://www.youtube.com/c/HarithaComputersTechnology/playlists

https://www.youtube.com/channel/UC-QDfvrRIDB6F0bIO4I4HkQ

https://www.youtube.com/user/CodingEntrepreneurs