OCR A-Level Computer Science Project

A Timetabling System

This project was initially marked at **67/70**

This was then moderated to **60/70**

This makes up **20%** of A-Level Computer Science

Examination Series: **2022**

Contents

1 Introduction

1.1 **Navigating the Document**

1.1.1 Structure of the Document

The document is organised into **5** sections.

1.1.1.1 The Development Process

The first **4** sections correspond to the parts of the development phase.

- Analysis
- Design
- Development
- Evaluation

While this is approximately in chronological order, some sections were written out of order to maintain this structure. Where this could affect the content of the section, effort has been made to indicate this with a note at the top of the section.

1.1.1.2 The Final Code

Snippets of the current state of the code are given throughout the document, where appropriate. Where code is given, it will be clearly distinguishable with a dark background and appropriate syntax highlighting. Code is present as text rather than images. An example is as follows:

```
if __name__ == '__main__':
    print("Hello World!")
```

There is also a comprehensive list of the final state of the code at the end of the document, should this be of use.

NOTE: Code in this project was limited to a line length, however that limit is greater than the width of this document. This means some code will wrap across lines in ways that may not be clear. This is not a feature of the code. The final code is also available on GitHub should this be easier to view.

1.1.2 References

All headings are numbered. Where a reference is in bold, e.g. **1.1.2**, it is clickable and will jump directly to the section being referenced. It is usually possible to then use a back button to go back to the previous position in the document. This should aid navigation throughout the document.

2 Analysis

2.1 Initial Overview

2.1.1 Problem Description

Currently, timetabling is done as follows:

- Divide lessons into option blocks according to subject preferences
- Spit up the day into fixed chunks called periods
- Allocate each option block to a period as to minimise clashes

This method is effective at simplifying the timetabling process, however has a number of drawbacks:

- Fixed length lessons are unproductive:
    - If the lesson is too short, the content will either:
        - be rushed, or
        - may be delayed until the next lesson, resulting in a gap
    - If the lesson is too long, either:
        - time will be wasted at the end of the lesson, or
        - the next topic will be started, again leaving a gap
- Lunch break must be long enough to allow all students to get to lunch, as the fixed structure does not allow lunch time to differ between people (aside from finishing an entire period earlier)
    - A shorter lunch break would result in a shorter school day so people can get home earlier
- The timetable is unable to adapt to school trips and other events that miss lesson time.
    - Some lessons (e.g. those on Fridays) are disproportionately affected and therefore do not end up with as much lesson time as other lessons
    - Teacher absences lead to notoriously unproductive cover lessons, as the timetabling process does not allow modifying the timetabling to accommodate for it
- These option blocks are sometimes inflexible and do not allow every single combination

Essentially, the timetabling system is fixed and inflexible. This means it simply cannot respond to the changing conditions present in a school environment.

2.1.2 Outline of the Solution

The key problem with the current timetabling system is that it is fixed and inflexible. The new software will instead be dynamic and flexible.

The idea is as follows:

- Teachers would input the amount of time they need for each lesson in advance and the software would dynamically produce the timetable

- The timetable would be constantly changing day-to-day according to the optimal lesson lengths as determined by teachers
- If teachers choose to schedule longer lessons, they will have fewer lessons to ensure each subject has the correct amount of time (as determined by the school)
- The timetable will minimise clashes and should be spread out appropriately

This has a number of benefits, including:

- Wasted lesson time is reduced as lessons will be scheduled for the correct length
- If a teacher was absent, their lessons could be reallocated with minimal disruption
- Lunch time could be scheduled appropriately along with lessons
- The flexibility allows more subject combinations to be offered, while potentially at the expense of a few clashes

There are also potentially many more benefits that arise from making the switch to a flexible timetable, many of which may be specific to just a few schools.

2.1.3 Primary Objective

The primary objective for the project is to create a timetabling system that will increase the productivity in schools. To achieve this, the software must deliver a well-balanced timetable and be easily accessible to everyone.

The amount of development required to create a product that is reliable enough to roll out in schools far exceeds the scope of an A-Level programming project. This means that the project will focus on the more critical aspects of the implementation and some of the desired features will not be feasible to implement into the project. However, the solution reached at the end of the project should still improve on the current timetabling model.

The features that will be removed from the solution (to maintain the core functionality and finishing within the time constraints) will be decided throughout the project when it is clearer what the timings of the project are.

2.2 Stakeholders

2.2.1 Overview

The stakeholders in the project are as follows:

Teachers	Consumer	Using the system to schedule lessons
Students	Consumer	Using the system to view lessons
Head of Timetabling	Consumer & Customer	Responsible for implementing and maintaining the system
Governors		Interested in the performance of the school
Senior Management		Interested in the performance of the school
Parents		Interested in students' education

It is clear that the major stakeholders are the teachers, students and the head of timetabling as they will be directly interacting with the project. All of these stakeholders will need their own interface. These will be collectively described as ‘users’.

2.2.2 Students

Most of the users will be students, so it is critical that they have a good user experience. Students will need to be able to view their timetable.

Some students will not be familiar with technology, so the interface must be very easy-to-use to make it appropriate to their needs.

2.2.3 Teachers

This is the second most common group and as with students, some are less experienced with technology so the interface must be easy-to-use. Teachers will need to view their timetable but will also need to schedule lessons. As having to schedule lessons could be seen as a drawback of the new solution, this must be very easy to do to avoid this extra effort outweighing the potential gains.

2.2.4 Head of Timetabling / Admin

This user is an exception to the rule – they can be expected to navigate more complex interfaces as they will spend some time understanding the interface. The main priority is that all functionality is available – it would be very bad if something went wrong with the timetabling algorithm and the school was unable to function for a short while. This user should have full control to override any decisions made by the algorithm and check things to ensure they are all running smoothly.

2.2.5 Other Stakeholders

Other stakeholders, such as the parents, governors, senior management, will not directly interact with the product however are still interested in the performance of the school. Satisfying the needs of the three users will automatically satisfy the needs of these stakeholders as they are primarily interested in the performance of the school. Therefore, for the project to be a success, it must satisfy the needs of the three types of users.

2.3 Investigating Limitations

2.3.1 Methods of Data Collection

As the main concept has been decided, it would be very beneficial to investigate further to find any potential limitations of this concept. Opinions must be gathered from the 3 main stakeholders – students, teachers and the head of timetabling. If identified early in the development cycle, these issues can be mitigated against where possible.

Four potential methods of data collection are as follows:

- Observation
- Interview
- Questionnaire
- Document Collection

Observation involves shadowing employees and recording information that seems important. This method would not be appropriate for the given scenario as most students / teachers would not be happy with this and there is not much information to gain from doing so. Observing the Head of Timetabling during the timetabling process could be very beneficial, however the timetabling for the current academic year is already complete so this is not a feasible option.

Interviews would be very appropriate for all three types of user. Interviews often give insights which could not have been predicted beforehand and allow for an in-depth discussion with each person to understand their needs fully. A disadvantage is that it is time consuming and so is unable to provide a breadth of opinions.

A questionnaire is a great way to extract information from large numbers of people with some understanding of what information is desired. As there is only one Head of Timetabling, this method would be completely inappropriate to collect data from this group. This is more feasible for teachers, however many teachers are too busy and do not have enough time to answer a survey. However, this would be exactly right for students as there are a large number of students and many have free time to answer a short survey. However, the survey would be best created after the interview as the interview is more likely to give insights which may not have been predictable, which could then be tested against a wider number of opinions.

Document collection would not be appropriate for this scenario as any documents that could be relevant would not be public information.

The final process of data collection will be:

- Student interview
- Teacher interview
- Head of timetabling interview
- Student survey

(The student interview should be done first as the student survey can be created once this is finished)

2.3.2 Interviews

2.3.2.1 Student

This interview was conducted with Luke Fleming, a student taking Maths, Physics and Computer Science on 22/09/21

2.3.2.1.1 Summary

First of all, Luke was asked for his thoughts on the current system. He commented that it was inefficient and could be improved. He commented that double periods are more productive than

single periods, suggesting lessons should be longer. He also stated that he would prefer it if study periods were more evenly distributed throughout the day/week, rather than having lots of doubles.

About the new timetabling system, Luke said that the most major issue for him would be that it disrupts his routine. He was then asked whether games lessons should be at fixed times, to which he responded yes. When asked about whether he looks at his timetable on his phone or on paper, he responded that he uses his phone but also raised a point about his parents wanting to see his timetable. This needs to be investigated further through a survey to assess whether a parent interface is necessary.

When asked about 55-minute lessons, he commented that it's not just about being longer or shorter – they need to be of a variable length so that the benefits of both longer and shorter lessons can be enjoyed. The conversation then moved to scheduling lunch times, which clearly needs improvement (currently, lunch is at 2pm). This could be added to the algorithm, such that lunch times are staggered with lesson times. He is not concerned that the whole school has morning break at the same time. More opinions can easily be gathered on this through surveys. He is also keen on being able to schedule separate meetings with teachers within the timetable.

2.3.2.1.2 Full Transcript

[Jump to bottom **2.3.2.1.3**]

[Key: Me Luke]

"Are you finding the current timetabling system to be an efficient way of scheduling lessons?"

"I'm finding it to be very inefficient – we've got single periods all the time (double periods are way better), physics lesson in no physics lab, which is very disruptive towards my learning."

"You mentioned double periods being better – can you focus for 2 hours?"

"Well if there's a break in the middle then yeah."

"And is that more productive than having two single periods?"

"Yeah because you don't waste 10 minutes in between walking around"

"Would you say there's lots of wasted time in the school day? Would you prefer to get home earlier and have a shorter lunch break, for example?"

"Yeah. Lunch break is too long"

"How effectively do you use your free periods? Can you work for a full 2-hour period? Would you prefer to have double frees or single frees?"

"You mean more frequent single frees or less frequent double frees?"

"Yeah"

"More frequent single frees"

"Because you can focus while doing work?"

"Yeah because, if you actually want to work, you're gonna probably do more because you realise you don't have as much time, whereas in doubles you're like 'I've got two hours', so you're less productive"

[The proposed solution was then explained here]

"How do you know when prep is due?"

"That's a good question. A provisional timetable would be available a few days in advance although it could change so you couldn't be absolutely certain about when the prep is due in. Teachers could set a date for it to be due which isn't necessarily a lesson, although this is a good issue that needs to be resolved"

"Maybe the timetable should be fixed for a couple of weeks in advance so students can plan ahead with when their prep is due. Would this help?"

"Yeah"

"Would having an uncertain timetable affect you much?"

"I feel like whenever you have a set timetable you get into a bit of a routine, you kinda know what's coming up – you feel prepared. Whereas I feel like the new system would be a bit 'wonky'"

"And what about lessons where you need to bring in something extra like P.E. kit? Would that make a difference if that was all over the place"

"Yeah it would because loads of people would be like 'Oh, I thought it was this day' and loads of people would forget"

"So you'd like that to be scheduled to fixed times?"

"Yeah. Tuesdays and Thursdays"

"Do you usually use it on your phone or do you have a paper copy?"

"I usually just check it the night before on my phone"

"Are there people who would prefer to have it on paper than on their phone"

"Maybe some people. What my mum wants me to do is to print out my timetable and put it in the kitchen so she can see it"

"That's interesting – I didn't think about parents actually"

"What would you say about the lesson length? Is 55 minutes too short, too long or about right?"

"I don't really know, I haven't ever had lessons that aren't about an hour"

"Do you find it to be inefficient to have 55 minute lessons"

"Yeah, lessons can often only be 45 minutes anyway once we actually get there and start. You don't want everything to be a double though, so it would be good to have more flexibility with lesson lengths"

"Are there any other ways you can see the current system better than the proposed solution?"

“Not that I can think of. The only thing I guess is routine.”

“And you wouldn’t say that outweighs the benefits of the new solution?”

“No”

“Any other ways you’d improve the new solution?”

“Earlier lunch”

“Would you like lunch times to be scheduled as part of this solution?”

“So the lunch times change every day?

“The lunch times would change based on your lesson times”

“How would that work with catering?”

“It would essentially work how it does now, but you would be allocated a lunch slot based on when your lessons finish”

“What do you think about the current system of allocating lunch times?”

“Awful. Lunch at 2 is way too late. Sometimes people are late because of lunch and lots of people just go to co-op to get lunch because it’s not a good time.”

“Do you like the fact that the whole school has morning break at the same time, or does that not matter much to you?”

“No, it doesn’t really bother me much”

“And what about when meeting separately with teachers, would you like that to be handled by the timetabling system? You could just choose to meet with a teacher and it would find times when you are both free”

“Harry was talking to Mrs Jones and asking when they were free and it took a while so yeah that would be a good idea to make that better”

2.3.2.1.3 Analysis

Overall, the interview mostly confirmed the viewpoint that the current system is inefficient. Luke was supportive of the concept, although did raise some concerns:

- Lessons where you have to bring extra kit (e.g. PE) should be at fixed times
    - Otherwise, it would be easy to forget and would form an easy excuse for students
- Luke mentioned that his parents were interested in seeing his timetable. Following confirmation by means of a survey, a parental interface be added.
- Luke also suggested that lunch break should be shorter. This would be decided by senior management when it is implemented, however the flexible approach would make it very easy to make changes like this.
    - He also suggested that allocating lunch timings as part of the solution would be a good idea. This can be explored further.
- The system could be used to schedule meetings with teachers. This will also be explored further in the survey.

2.3.2.2 Teacher

Interview conducted with Mr Gibbons, Teacher of Computer Science and Physics

2.3.2.2.1 Summary

The following is a summary of the interview written soon afterwards, in approximate chronological order.

The first question asked Mr Gibbons for his thoughts on the current timetabling system. He commented that it is fine but inflexible and doesn't make best use of the time available. Regarding room allocation, he noticed that he was being sent to rooms across the site while colleagues were using rooms much closer, which is inefficient and requires lots of walking around. When asked about whether choosing lesson lengths would be helpful, he said that it would be very helpful. When teaching, some topics just don't require a whole lesson to teach which wastes time. He then mentioned that some things could be taught with a teacher lecturing the whole year, which would reduce teacher contact time and make the school more efficient. This can vary based on subject. Computer science lessons can occur in basically any room as all students have laptops, however it's important to try and reduce the amount of time spent walking around as it is a large site – too much walking around can be frustrating. Physics lessons can require labs although these aren't always required, so a flexible schedule would definitely be beneficial.

With scheduling lessons, Mr Gibbons said that he wouldn't find it inconvenient as he has an Excel spreadsheet with all lesson plans so inputting it into another system wouldn't be a major inconvenience. It's also not required when planning to know exactly when each lesson will be, just the duration of the lesson. Some teachers with a more set routine will not immediately appreciate the flexibility and won't enjoy using it. When asked about how to mitigate it, he suggested highlighting how it could reduce teacher contact time as teachers would appreciate that.

Regarding students, he thought that students would be fine with the new system provided there was enough time in advance to know what's happening (e.g. the timetable changing on the day would be very difficult to keep track of). Most students have not memorised their timetable, so aside from a few obvious patterns being broken (e.g. double every Wednesday), it would not have a major impact on students' (or teachers') routines.

An additional concern was with the school having a large site, the system should be mindful of travel times. An amazing solution would estimate the distance between rooms and ensure people don't have to travel too far, although this may be beyond the scope of the project.

2.3.2.2.2 Transcript

A full transcript was not recorded for this interview.

2.3.2.2.3 Analysis

Mr Gibbons was very positive about the solution and gave responses that were for the most part expected, however he did raise one unexpected point. He suggested that some lessons would be suitable to teach with lectures instead of lessons, which could reduce teacher contact time. This could be followed up further via the student survey to check how students would react to this.

Mr Gibbons was very keen on room allocation that minimised the travel time around site. This will also be considered further, however may be beyond the scope of the project.

2.3.2.3 Head of Timetabling

Interview conducted with Mr Chard, Head of Timetabling on 28/09/21

2.3.2.3.1 Summary & Analysis

Below is a summary of the interview along with some analysis about how it will be incorporated into the final solution.

Initially, the discussion was focused on how timetabling is done right now. He explained how the day is first and foremost divided into fixed time slots. Then, starting with the oldest year groups, all of the subjects fit underneath such that there are no conflicts. He described it as a 3-dimensional structure, with everything fitting underneath each other such that it all lines up. This isn't done by hand, instead is significantly accelerated through computation by using Edval. Edval is a big improvement over doing it by hand, although it is essentially still modelling traditional timetabling methods. It's important to note that while Edval provides many powerful tools to timetable itself, it still allows the user full control over if lessons should be at fixed times, in specific rooms or with specific teachers. This should also be the case in the proposed solution, to ensure that the head of timetabling still has full control over the final timetable. Edval will be explored further later. The first step while timetabling is to construct the option blocks and allocate students to classes. This is done entirely by an algorithm. Mr Chard is not an expert into how these algorithms work, and so was unable to provide details on this. The blocks often have a width that is slightly more than the number of lessons that need to be scheduled which gives flexibility. He inputs all of this data into the program (including any lessons that have to occur at fixed times) and, one year group at a time, the program will output a possible combination and the score of it. The score is essentially the sum of the weightings of each issue, with spread issues having a low weighting but teacher conflicts having a very high weighting, for example. The algorithm attempts to find the combination with the lowest score. By the time the lower years are reached, there is a higher chance of conflicts although there are also far fewer constraints as many students are in the same class for many subjects. Unfortunately, this is never the perfect solution and Mr Chard has to spend 7-8 hours tweaking things until the timetable is adequate. Overall, this involves a combination of using Edval, many spreadsheets and lots of work.

Note that the number of classes for each subject is not fixed, and the school is constantly trying to get the lowest number of classes which still has an acceptable number of students in each class. This can lead to some subjects which typically have 2 classes only having 1, or other subjects having to increase the number of classes to get the timetable to work. The number of classes provides flexibility while timetabling using a traditional method, however with the proposed solution the number of classes will have to be fixed in advance. This removes another element of flexibility while timetabling, which may make timetabling more difficult.

Regarding particular complications with the timetable, he mentioned a few particular ones to look out for. At the school some lessons in Sixth form are taught with another school, and the dates for these have to be agreed in advance. They also have to be long enough to be worth the journey required to get there. This means once those are fixed, the rest of the timetable simply has to work around it. Part-time teachers can also be more difficult to work with as the timetable has to be

condensed into the days that the teacher works. Mr Chard mentioned that he had to make some teachers change their working days to satisfy the timetable. Other issues include games lessons and PE (teacher availability), DT (room availability) and other requirements from departments.

Clashes aren't the only thing that must dealt with. There are also many smaller issues such as rating the spread of lessons, whether teachers can teach their preferred thing, teachers in their preferred rooms and various other departmental requests.

Another point was that the time required for each lesson also varies on a departmental basis – MFL would love to have lots of 30 minute lessons, whereas DT would prefer only 2 hour lessons. This is great as it means even if teachers do not want to choose a different lesson length of each lesson, they can still take advantage of a different lesson length for every lesson which will still improve productivity (albeit not by as much).

2.3.2.3.2 **Transcript**

A full transcript was not recorded for this interview

2.3.2.3.3 Conclusions

Overall, the biggest takeaway was that timetabling still involves much human effort. Mr Chard mentioned that it takes him 7-8 hours of work, satisfying many complex requirements until the timetable is suitable. The proposed solution will attempt to remove this element such that it is fully automated.

2.3.3 Survey

Despite conducting multiple interviews, these still only represent a very small proportion of those who will be using it. A single computer science student will not represent the views of all students in the school across all year groups. To gather more opinions, a survey will be devised based on the results from the interview to understand whether these interviews represent the views of more people.

2.3.3.1 Questions Asked

The questions asked, including justification, are detailed in this section.

What year are you in?

This question helped assess how diverse the opinions are and whether it has been successful in reaching a wide audience

Which subjects interest you most?

Again, this question seeks to assess what kind of people are answering the survey to ensure there are a wide range of opinions

How many A-Levels do you take?

This helps understand how much free time people have which could influence their responses in the survey and will affect how easy timetabling is.

Do you have additional commitments that cause you to miss lessons?

It is helpful to know whether there are a significant proportion of people with certain commitments as it may become a priority for the timetabling algorithm

To what extent would you say the current timetabling system is inefficient

This is a general option that asks people to think critically about the current timetabling system. This data will be useful to get a general opinion on whether the current system is efficient or not.

Select any issues with the proposed solution that would affect you

This quantifies how many people would be affected by the issues raised so far. Respondents are encouraged to only choose options that will impact them in a noticeable way, rather than simply checking all options (as it is already known that these are issues). Quantifying this allows for quick analysis of the results.

Are there any other issues?

This allows respondents to input other issues they can think of that aren't in the above choices. It's important to still allow respondents to raise new issues that may not have been considered already.

Do you like having a long lunch break?

This is a simple question that assesses whether the school day should be shorter with a shorter lunch break.

Rank the options based on how productive you are

This contains multiple situations to be ranked in order, although the most interesting options are whether people are more productive in lunch break or during supervised homework sessions, and homework sessions vs being at home. This could lead to changes in how long the school day is to increase productivity.

Would you be more productive if the school day was shorter and you had fewer study periods?

This attempts to assess how effectively students use study periods without explicitly asking. This also reinforces the question above about whether they would work more productively at home or at school.

What do you think about 55 minute lessons?

This is a very simple question to help understand the general opinion on the duration of lessons.

Are your parents interested in seeing your timetable?

Luke raised a point that his parents were interested in seeing his timetable. A question in the survey will assess whether this is consistent with other students, which may determine that a parental interface is necessary.

Do you like that the whole school has morning break at the same time?

A potential issue with the new timetabling system is that the school will no longer all have morning break at the same time. Although Luke said that he did not find this to be an issue, more opinions are required.

Would you like to have games lessons at fixed times?

This question determines whether there needs to be functionality to schedule fixed lessons for when extra stuff is required (e.g. PE kit)

Do you ever use a physical copy of your timetable?

Physical copies are potentially problematic with the new system as it would change all the time, so students would have to print out their timetable weekly. This question seeks to understand how significant a problem this would be.

How have you found trying to arrange to meet with teachers outside of lessons?

This is a potential feature that could be added to the timetabling system. This will ask about people's current experiences

Would you like it if this was added as a feature?

This will gauge the interest in this feature

How would you feel about some lessons being converted into lectures?

Mr Gibbons suggested using some lectures as well as lessons to reduce the teacher time required while still delivering a good education. This question seeks to assess students' opinions on the matter. Respondents are also encouraged to give a reason for their answer.

Overall, how much of an improvement do you see this system having?

After being given time to think critically about both timetabling systems, this question gives respondents a chance to evaluate how much of an improvement they think the system will really have, and whether it will be worth doing

Other comments

A final text box allows for other comments / concerns to be mentioned

2.3.3.2 Results

2.3.3.2.1 Responses gained and diversity of responses

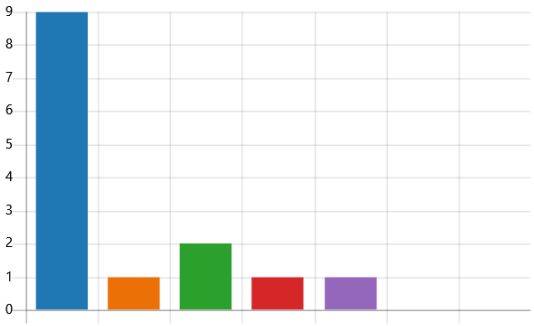
Overall, there were not quite enough responses. There were 14 responses, 9 of which are from year 13 / upper 6. The diversity in subject choices is equally poor, with 11 of the people selecting that they are interested in sciences.

This limited amount of data does restrict the conclusions that can be drawn from the data. Nonetheless, the survey still contains much data that is very usable.

1. What year are you in?

More Details

Year 13 / Upper 6	9
Year 12 / Lower 6	1
Year 11 / Upper 5	2
Year 10 / Lower 5	1
Year 9 / Form 4	1
Year 8 / Form 3	0
Year 7 / Lower 3	0



2. Which subjects interest you most?

More Details

Sciences - Maths, Physics, Co...	11
Social Sciences - Geography, E...	1
Languages - English, French, L...	1
Arts - Music, Drama, Art	1
Other - None of the above fit ...	0

2.3.3.2.2 Assessing how much free time people have and other complications

The responses here were to be expected, with most people in sixth form taking 3 A-Levels, a few also taking EPQ and one person taking 4 A-Levels. This means for most people in sixth form, there is room for flexibility with timetabling. Most respondents do not have additional complications that cause them to miss lessons, however 3 mentioned music lessons and 3 mentioned sporting commitments. This means it is something to consider in the final timetabling system, however it is not a major issue if it isn't done as it doesn't affect everyone.

2.3.3.2.3 Opinions on the proposed solution

This question had mixed responses, with all the potential issues getting at least 2 people choosing them and one issue getting 7. These issues will be discussed in order of importance.

The most important issue was 'Inconsistent homework schedule', in which 7 people (half of those who filled in the survey) reported that they would consider it to be an issue that would affect them. It is clear that people like having their homework set and due in on fixed times, which would be a potential issue with the new solution. This will be addressed later.

Next, 4 people said that the new system disrupting their routine would be an issue for them. Unfortunately, this issue is intrinsic to the proposed solution so does not have a clear solution. The issue is not particularly major and will be outweighed by the benefits of the solution.

2 people also said that they could end up forgetting things to bring to lessons. The main lesson that requires bringing something else in is PE, which will be scheduled at fixed times so this shouldn't be a major issue. Another 2 people noted that arranging commitments outside of school could prove an issue. This should be incorporated into the timetabling system to the maximum extent possible if there is enough time to do so.

1 person also raised a new issue that people can't memorise their timetable. By speaking to a few people, most people do not know their timetable so this should not be a big issue. However, this would've been a valuable question in the survey.

2.3.3.2.4 Structure of the day – Length of the day, length of lessons, lunch break etc.

When asked about lunch break, all students apart from 3 commented that they prefer the long lunch break over going home earlier. Of those people, all but 3 said that it was because they wanted time to relax, rather than partake in the activities the school offers during that period. A long lunch break will make timetabling easier as there is more time for lessons to be scheduled in (as lunchtime can move).

The next question has lots of information available, asking respondents to rank in order how productive they are in the following 5 situations:

- Supervised study period
- Unsupervised study period
- Lunch/break times
- In the evenings
- On the weekend

The results for this question were very mixed, with marginally more people saying that they are more productive at home than at school (suggesting the school day should be shorter). Most respondents agreed that lunch/break times are the least productive time of day, although this may be for good reason as time to relax is important. Almost everyone put supervised prep as more productive than unsupervised prep, although very few put it significantly above, suggesting that it would be somewhat beneficial to allocate a teacher for study periods although it is not required.

The question afterwards was more direct, specifically asking if people would be more productive if the school day was shorter. Interestingly, this question was completely balanced, confirming the results of this question in that everyone is different.

11. Would you be more productive if the school day was shorter and you had fewer study periods?

More Details

Yes, significantly	2
Yes	2
Wouldn't make a difference	3
I would be less productive	6
I would be significantly less pr...	1

This suggests that the length of the school day is about right and does not need to be changed. This leaves plenty of time to schedule lessons into which should make timetabling easy.

Regarding the lesson length, half of people noted that it was about right and half noted it was not. Of those who commented it should be changed, about half commented it should be longer and half commented it should be shorter. This confirms the idea that there is no one perfect lesson length, and they should be of variable lengths.

12. What do you think about 55 minute lessons?

More Details

Too Long	4
Too Short	3
About Right	7

2.3.3.2.5 Specific Questions

As Luke mentioned that his parents were interested in seeing his timetable, a question was added to the survey to assess if this is representative of all students. Overwhelmingly, students decided that a parental interface to the timetable would not be necessary.

13. Are your parents interested in seeing your timetable?

More Details

Yes - They often like to know ...	2
Sometimes - They would appr...	1
Occasionally	3
Never	8

Another potential issue is morning break not being at the same time. Generally, people did not mind and it made no difference to them.

14. Do you like that the whole school has morning break at the same time?

More Details

Yes - It's essential to me	0
Yes - It's nice to have everyon...	4
Indifferent	8
No - I'd prefer it if it was stag...	2

Unanimously, people decided that games lessons should be at fixed times to avoid forgetting PE kit.

15. How would you find having games lessons at variable times? Would you be more likely to forget your PE kit?

More Details

As for using technology to access their timetable, everyone in the survey predominantly uses their phone over a paper copy, making it very easy to switch to the new system. It's important to note that this survey does consist of mainly people doing sciences, so this may not be representative of everyone in the school.

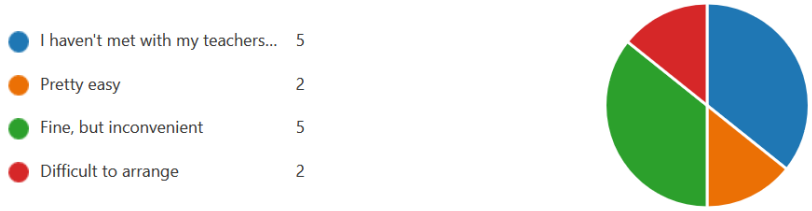
16. Do you ever use a physical copy of your timetable?

More Details

Finally, respondents were asked about the possibility of arranging meetings being added. Most respondents said that it would be a nice to have but not essential feature.

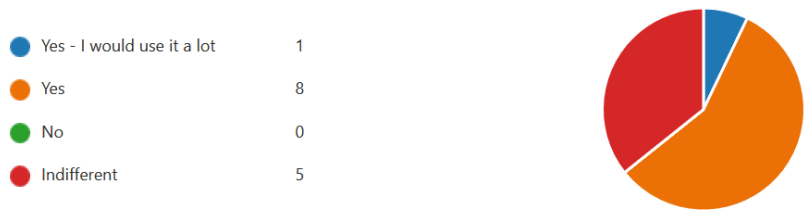
17. How have you found trying to arrange to meet with teachers outside of lessons?

More Details



18. Would you like it if scheduling meetings with teachers was added as a feature in the timetabling system?

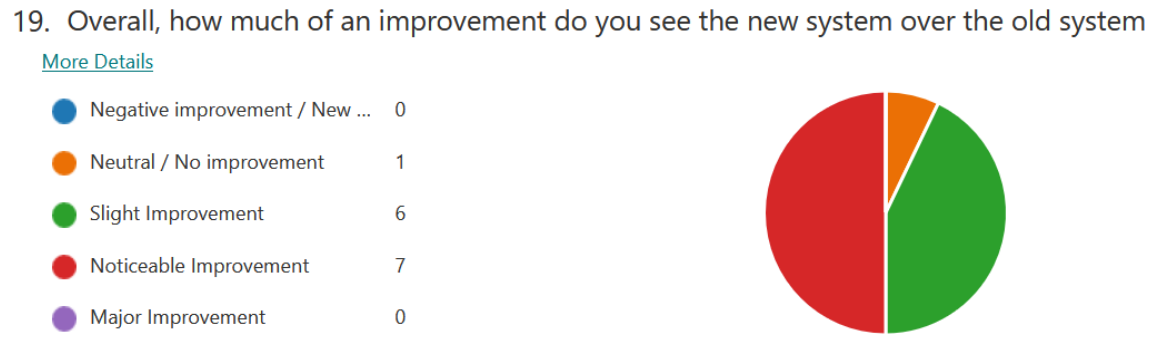
More Details



When asked if they would enjoy having lectures instead of lessons, most indicated that they prefer to have lessons, but that the occasional lecture would be tolerated. Most respondents reported that they would find lectures unengaging and prefer to have the variety of actual lessons where there are interactive tasks to complete, rather than having to just listen to a teacher.

2.3.3.2.6 Final Question – General thoughts

Finally, respondents evaluated how much of an improvement they thought the new system would have. Responses were positive, with no-one answering that it would have a negative improvement and only one person saying that it would not have any improvement. 6 people said it would have a slight improvement and 7 said there would be a noticeable improvement.



There was nothing relevant in response to the final question asking for other comments / concerns about the timetabling system.

2.4 Mitigating these Limitations

Limitation	Mitigation	Justification
Lots of uncertainty – disruptive to routine.	Timetable will be fixed for 2 weeks from the current date. Should it have to change within the 2-week period, users would be notified. It should be tested further what the optimal duration is: - Longer than 2 weeks would be less flexible, but would allow for more forward planning - Shorter than 2 weeks would be even more flexible, however could lead to confusion	This allows users to plan ahead, while also being able to adapt and not conflicting with the goal of the project.
Students will lose the opportunity to become accustomed to patterns in the timetable.	There is no clear mitigation for this limitation – this problem is intrinsic to the solution.	While this is a slight issue, it is not major and will be outweighed by the benefits of the solution.

Teachers may focus on the inconvenience over the benefits	As suggested by Mr Gibbons, the flexible timetable could offer reduced teacher contact time, which would be attractive for teachers	This would help teachers to see the value of the solution if they don't otherwise.
No regular homework schedule	Again, this issue is intrinsic to the solution offered. This should not be an issue if teachers give students enough time to do their homework and if students are able to think ahead.	If students have enough time to do their homework, they can plan around any variations in the timetable.
Students could forget things to bring to lessons	Lessons that require specific things (e.g. PE) to be brought in could be scheduled at specific times so students can plan ahead for it	If these lessons occurred at fixed times, then students should not be any more likely to forget kit than currently. Games sessions also do not need to take advantage of variable-length lessons.

A big challenge still unresolved is how the new system will manage to schedule the same (or slightly less as the time is more productive) lesson time into the same length of day while allowing for a more flexible timetable. Some flexibility will be gained from break times and lunch times, however this needs to be explored further in **3.4**

2.5 Why computers?

2.5.1 Summary

Timetabling is a very difficult problem with many possibilities to be checked. For obvious reasons, doing this by hand would be a very tedious process.

Computers are able to solve this problem significantly quicker than a human by checking through possibilities much faster. While humans will have better heuristics as they have more understanding about timetabling, computers are able to check through possibilities so much quicker that a computer is the most viable option for timetabling. Current methods of timetabling use a hybrid approach with the computer performing the computation and a human reviewing the output and making changes accordingly.

This is only exemplified by the fact that the proposed solution is due to timetable dynamically – if done by hand, this would require a significant amount of time, possibly even a full-time employee, to keep generating the timetable. There are also even more possibilities to consider in the proposed solution such that timetabling by hand is essentially infeasible.

Clearly, a computational approach is very well suited to the problem.

2.5.2 Computational Thinking

These methods of computational thinking will help to solve problems throughout the project.

2.5.2.1 Thinking Abstractly

Many areas of the project will be abstracted, including, but by no means limited to:

- Time will be in discrete 5-minute chunks rather than continuous
    - This reduces computational complexity and may reduce development time
- Some things will not be considered in timetabling, such as if people feel more 'in the mood' for a certain subject at certain times
    - This reduces computational complexity and the time to program
- In the interface, some information will be hidden, such as the other people in the class (on the student timetable)
    - This makes the solution easier to use

2.5.2.2 Thinking Ahead

The timetabling algorithm takes in a list of classes, including:

- Students in the class
- Teacher of the class
- Subject
- Past lessons with that class

The algorithm outputs a number of lessons, each including:

- The time the lesson is scheduled for
- Whether it is still subject to change
- Details of the lesson (students, subject, class etc.)

Some preconditions for timetabling lessons are as follows:

- The list of subjects
- The list of students
- Which subjects each student has chosen
- Length of the school day

2.5.2.3 Thinking Procedurally

The problem is broken down in **3.1.1**

2.5.2.4 Thinking Logically

Some logical decisions will be taken in the user interface, such as:

- If a user clicks log in, check if they have input any credentials
    - If not, output an error
    - If so, check if the credentials are correct
        - If so, log in
        - If not, output an error 'access denied'

2.5.2.5 Thinking Concurrently

Many parts of the project must occur at the same time, such as:

- Multiple users will need to be logged in at the same time to view their timetable, otherwise many people would be unable to access their timetables during the school day.

The benefits of concurrent processing are very clear. The drawbacks are minimal, but could include:

- It will be more complicated to program
- It will be slightly more computationally expensive

2.5.3 Computational Methods

Computational methods allow problems to be solved in a computational manner. Problems suited to these methods are likely suited to a computational solution.

2.5.3.1 Problem Recognition

Problem recognition is critical to ensure the full scope of the problem is fully understood. This is detailed in **2.1.1**.

2.5.3.2 Problem Decomposition

Problem decomposition involves breaking down a large problem into a series of smaller sub-problems which are easier to solve. This is executed in **3.1.1**.

2.5.3.3 Backtracking

Backtracking will be very important while timetabling.

It is particularly relevant for class allocation:

- Try a potential solution
- If it does not work, go back and try another one

This could be implemented recursively to check many possible allocations and achieve the best possible allocation. This is very easy to do on a computer, so the problem is suited to a computational approach.

2.5.3.4 Heuristics

The timetabling problem is NP-complete, so it is essential that heuristics will form a big part of the solution. It will not be possible to check every solution within a feasible amount of time, so heuristics will have to be used to make an algorithm that converges quickly to a ‘good enough’ solution.

2.5.3.5 Abstraction

Abstraction will be used extensively throughout the design and development of the solution. Some examples are detailed in **2.5.2.1**. Abstraction can transform problems which were completely infeasible into problems which are very well suited to a computational solution.

2.6 Exploring Existing Solutions

2.6.1 Edval

Edval is a piece of timetabling software that is used in many schools. As mentioned by Mr Chard in the interview, it very much mirrors the traditional method of timetabling while accelerating it through computation.

Edval has many features that allow the user to customise all the lessons, rooms and teachers. It has an easy-to-use interface meaning even those who aren't good with technology will be able to understand how to use it.

The display shows the timetable as coloured blocks, clearly organised into rows and columns. The user interface is explored in more detail in **3.2.1.1**

The key features include:

- Storing the classes / students / teachers
- Automatically allocating staff and rooms
- Managing spread of timetables
- Minimising other minor issues with timetables

It allows adjusting the weighting of the issues depending on the school's needs.

However, while computers are already speeding up timetabling significantly, the computer still seeks to mirror the method of timetabling by hand, in that it is all done in order. Each year group is done in turn, with a person having to manually adjust it to make sure it all works. This is still a traditional approach to timetabling which can be improved. An approach where the computer would take more control away would be desirable as it would involve less work for the school to create a timetable.

2.6.2 Other Solutions

While it could be beneficial to investigate more existing solutions, they are all very similar. There is simply no other software that implements a timetabling algorithm in a similar manner to the proposed solution, so exploring other solutions is not relevant. The user interface of these solutions is only relevant for the admin interface, however the proposed solution will include an interface for all users, including students and teachers.

2.7 A Refined Solution

2.7.1 Summary

2.7.1.1 Teacher Interface

The teacher interface will:

- Allow teachers to schedule lessons, choosing the length of time they want. Optionally, they could put the title of the lesson to aid scheduling.
- It would allow scheduling the entire year of lessons from day 1, although this may not be practical.
    - Some teachers will prefer to plan out the whole year at the beginning and not have to worry about it later, however others will prefer the flexibility offered by doing it throughout the year
- Show teachers which lessons they have and when

2.7.1.2 Student Interface

The student interface will:

- Show students which lessons they have and when

2.7.1.3 Admin Interface

The admin interface is not required for the functionality of the product however it will serve as a backup option in case there are any issues. It will:

- Make amendments to the timetable in a way that bypasses the underlying algorithm
    - This ensures there is always a backup plan if something goes wrong
- Provide full control over the whole system
    - Again, the head of timetabling needs to have full control of the system to ensure the school can remain functional.
    - Existing timetabling solutions provide full control, so this must be matched in the new solution.

2.7.1.4 Class Allocation

This algorithm must take in a list of students and their choices and divide them into classes in such a way that is conducive to effective timetabling throughout the year.

The criteria which maximise the effectiveness of the timetabling algorithm will be determined after the criteria for the timetabling algorithm itself. This is explored further in **3.3**

2.7.1.5 Timetabling Algorithm

This algorithm will schedule lessons which have been chosen by teachers based on sensible criteria that lead to a balanced timetable. The exact criteria will be decided in the development stage, but they could include:

- Minimising clashes
- Spread of lessons throughout the day
- Minimising short breaks (e.g. 10 mins)
    - Either a long break (30+ mins) where work can be done, or a 5 minute break to give time to get to the next lesson
- Ensuring each subject gets the correct amount of time across the year

These criteria are decided in **3.4.2**

2.7.2 Hardware & Software Requirements

While users will interact with the software on their machines (through a browser), a server will also be needed to run the backend code for the software to function. These have entirely separate requirements.

2.7.2.1 Client

2.7.2.1.1 Hardware – Desktop

The following only applies when interacting via a desktop machine

Component	Requirement	Justification
CPU	Any modern CPU capable of running a modern browser	Without a CPU, the browser would not be able to run
GPU	Any dedicated or integrated GPU capable of running a graphical operating system	A GPU is needed to render the website and display it to the user
RAM	Any amount of RAM that satisfies the minimum requirements for the operating system and browser being used.	Memory is required to run programs
Monitor	Any monitor with resolution at least 720p	A monitor is used to display the website
Mouse	Any mouse / trackpad / touchscreen	Used to click on UI elements
Keyboard	Any keyboard / on-screen keyboard	Used to type in information

2.7.2.1.2 Hardware – Mobile

The following only applies when interacting via a mobile device

Component	Requirement	Justification
SoC	Any modern SoC containing a competent CPU, GPU and RAM that can run a modern browser	This is integral to the system running

Display	Any display at least 480p	This allows the user to view the interface
Touch Screen	Any touch screen (ideally capacitive)	This will allow users to interact with the website

2.7.2.1.3 Software

Software	Requirement	Justification
Web Browser	Any modern web browser. The following will be specifically tested (both desktop and mobile versions): - Chrome - Firefox - Safari	The web browser is needed to render the HTML, CSS and JS sent by the server.
Operating System	Any operating system	An operating system is required for the computer to function

2.7.2.2 Server

2.7.2.2.1 Hardware

Component	Requirement	Justification
CPU	A desktop class consumer grade CPU with at least eight cores. The performance must meet or exceed all of the following: - **1000** in Cinebench R23 single-core - **10,000** in Cinebench R23 multi-core	All of the computation for the timetabling algorithm will occur on the CPU so it is essential that this is fast. A CPU that meets these requirements is easily available for testing.
RAM	At least 2 GB of RAM	This amount of RAM is very standard and will suffice for the program.
Network Connection	A reliable server-grade network connection with a latency of 50ms or lower. Bandwidth is not a requirement, however may be a consideration in a large school with many users.	A poor network connection will render the website unusable and users will not be able to log into the website and view their timetable.

NOTE: Another computer is likely required, along with peripherals, to interface with the server. No GPU is required for the server as it only has to be accessible via a command line.

2.7.2.2.2 Software

Software	Requirement	Justification
Python	Python run-time environment.	The RTE is needed to run the code which the project is developed in.
Django / Libraries	Django, and other Python libraries, will need to be installed.	The project relies on APIs provided by these libraries.

Operating System	Any operating system. It will be specifically tested on a Linux-based OS.	An OS is required for the computer to function. Linux is often used for servers.

2.7.3 Requirements Specification / Success Criteria

NOTE: Where the test is very apparent from the details in the 'Requirement' column, it has been omitted from the 'Test' column.

- *For example, for requirement 1, the test would consist of navigating the interface and ensuring both teachers and students are able to view their timetable as described in the requirement.*

2.7.3.1 High Priority Requirements

The final product must satisfy all the following requirements:

#	Requirement	Test	Justification / Notes
1	Students and teachers must be able to log in securely and view their timetable		Core functionality
2	It must be resistant to the following forms of attack: - Cross-site request forgery - SQL injection	SQL injection can be tested by inputting SQL special characters into the username field on the login page. CSRF can be tested by manually sending a POST request to the server and ensuring it is rejected	While this data is not very sensitive, it is always good practice to keep data private.
3	The student and teacher interfaces must be very accessible. It must be clear and easily usable for those with less experience using IT. No prior knowledge of the software is required.	An average student / teacher should, given relevant credentials, be able to log in and understand their timetable within **one** minute. Out of at least **3** randomly selected students, all **3** should be able to complete this task. The software should satisfy modern accessibility standards.	Many students and teachers will be used to the traditional system so it must be a positive experience to migrate to the new system. If the UI is confusing, it will be difficult for them to adjust.
4	All major browsers should be supported, including both desktop and mobile versions. This includes, at minimum:		There are many students and teachers in a school which will be using many different devices with many different browsers. It should work on as many as possible so that it is convenient to view the

	- Chrome (and other chromium-based browsers) - Firefox - Safari All must be supported on the latest version at the time of release.		timetable (and not any more difficult than currently)
5	The website must load fast	**Two** of **three** attempts must pass: - The latency to the server should be calculated by pinging the IP address - The server must respond in less than 50ms more than this latency *NOTE: Each test must be conducted on a different page*	Accessing timetable should not be any less convenient than currently, otherwise users may respond negatively to the change. A slow interface will make the interface frustrating to use. *NOTE: The latency to the server depends on implementation which is not in the scope of the project*
6	Teachers must be able to input lessons to be scheduled along with their duration		Core functionality
7	The administrator(s) must have full control over the entire system. This must include, but is not limited to: - Manually changing lesson times / overriding algorithmic decisions - Adding / removing users - Changing classes	Listed functionality should be tested	The head of timetabling is responsible for ensuring the timetable is working properly and, in the event that something goes wrong, may be held accountable. A school requires 100% uptime, so if something did go wrong there must be full control to override any decisions made and ensure temporary functionality of the system. *NOTE: The admin interface is not bound by requirement **3***
8	The timetabling algorithm must ensure that, over a given academic year, each subject gets approximately the correct total lesson time, as input by the admin	Over the course of a year, no subject should differ more than 1% from its desired allocation. Additionally, in an average day, the average student should have at least **44** 5-minute slots of lesson time	The school has an obligation to provide a certain amount of time with each subject which must be fulfilled. Teachers must be able to plan based on a given amount of available teaching time.

		(out of a 96-slot day). At least **10** days must be considered randomly to produce an average. Lesson lengths can be assumed to be randomly and uniformly distributed.	Calculations are available below (marked *)
9	The timetabling algorithm must run within a feasible time frame.	The timetable for the next day to be timetabled (in two weeks) must be complete by 8am. The algorithm can begin as early as is necessary, but not before 6pm on the previous day.	If the algorithm does not complete by the start of the school day, students will not be able to view their upcoming timetable.
10	The timetabling algorithm must minimise teacher / student clashes.	The algorithm can be run with sensible scenarios. No clashes should be observed. Given more challenging scenarios, few clashes should be observed.	Teacher clashes are very bad as the lesson cannot be taught without a teacher. Student clashes are bad as students will miss out on part of their learning.
11	The software must schedule lessons for a sixth form with up to 100 students in each year group, each choosing 3 or 4 subjects out of up to 15 subject choices.		Core functionality *NOTE: Classes will be predetermined and input into the algorithm*

** A typical school schedules 6 blocks of 55 minutes each, leading to 330 minutes of lesson time per day (including any free periods). If the school has 4 blocks and each student chooses 3 subjects, this works out to roughly 225 minutes (3 hours 45 minutes) of lesson time to be scheduled (excluding free periods). A school day includes about 8 hours of potential lesson time. This means that just under half of the available time should be made up of lessons, for each student.*

2.7.3.2 Desirable Aspects

It would be desirable if the final product satisfied the following requirements:

#	Requirement	Success Criteria / Test	Justification / Notes
12	Teachers could be able to input a topic for each lesson to identify what each lesson will be about		This helps teachers plan ahead and organise what they are planning to teach in each lesson.
13	Teachers could be able to modify the topic for lessons they have already scheduled		This would help correct typos or change their plan. *NOTE: Modifying the duration should not be supported, as this interferes with timetabling*

14	Teachers should be able to cancel lessons they have scheduled accidentally		Accidentally scheduled lessons would be inefficient and would likely waste teacher / student time if they have to be carried out at an undesired length, or the admin's time if it is to be cancelled.
15	The timetabling algorithm should deliver a balanced timetable that is conducive to productive work, where lessons are spread out appropriately according to sensible criteria.	This requirement is vague and very difficult to test. The best way of testing would be to take sample outputs and judge whether it seems like a good, balanced timetable.	The principal objective for the project is to improve productivity. The algorithm must achieve this. *NOTE: This is not more specific as the criteria have not yet been decided. The criteria are decided in **3.4.2.1***
16	The software could notify students if their timetable has changed within the next two weeks		Students should be able to rely on the timetable for the next two weeks remaining constant. *NOTE: Timetables could change for unexpected reasons, such as teacher absences.*

2.7.3.3 Potential Extra Features

Should there be sufficient time, the final product could include:

#	Requirement	Why would this improve the final product?	Why is this not a requirement?
17	The software could be able to assign students to classes	Requiring a third-party solution may cost money and is inconvenient. If done in the solution, it can be customised to minimise clashes throughout the year.	There are already many solutions on the market offering this functionality. Optimal class allocation to minimise clashes still appears to be a traditional block-based method.
18	User registration / password management	If users could register themselves, the head of timetabling would not have to manually input each user.	This can be achieved on the admin interface. Data input by the admin will be more reliable and less vulnerable to malicious activity.
19	Timetabling for all year groups (7-11)	It could be rolled out across schools with more year groups.	As the system needs 100% uptime, a limited rollout for sixth form colleges would help to test the system to ensure it works properly.
20	Room allocation	Lessons must take place in rooms. These need to be allocated to everyone knows where to go.	This is not a focus of the project. Room allocation can be done randomly (however this will lead to lots of moving around site)

3 Design

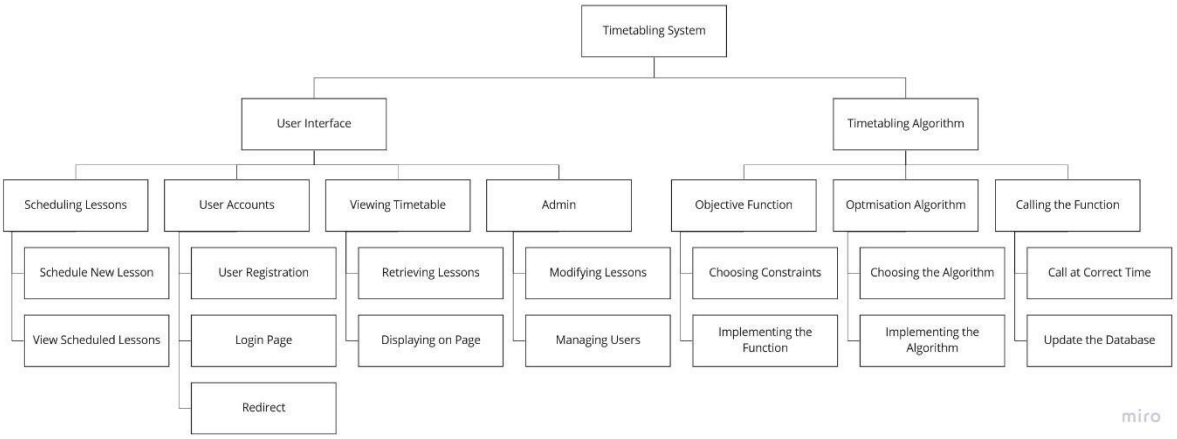
3.1 Structure of the Solution

3.1.1 Top-Down Modular Design

A timetabling system is a very large project that cannot be easily handled in one go. To make the project more manageable, it must be broken down into smaller problems using a top-down modular design. These smaller problems can then be completed much more easily and independent of each other.

Below is a top-down modular design that breaks the project down into a series of smaller problems which are easier to deal with

*NOTE: The text in this diagram is small. If it is not readable, close-ups have been provided below (**3.1.1.1**)*

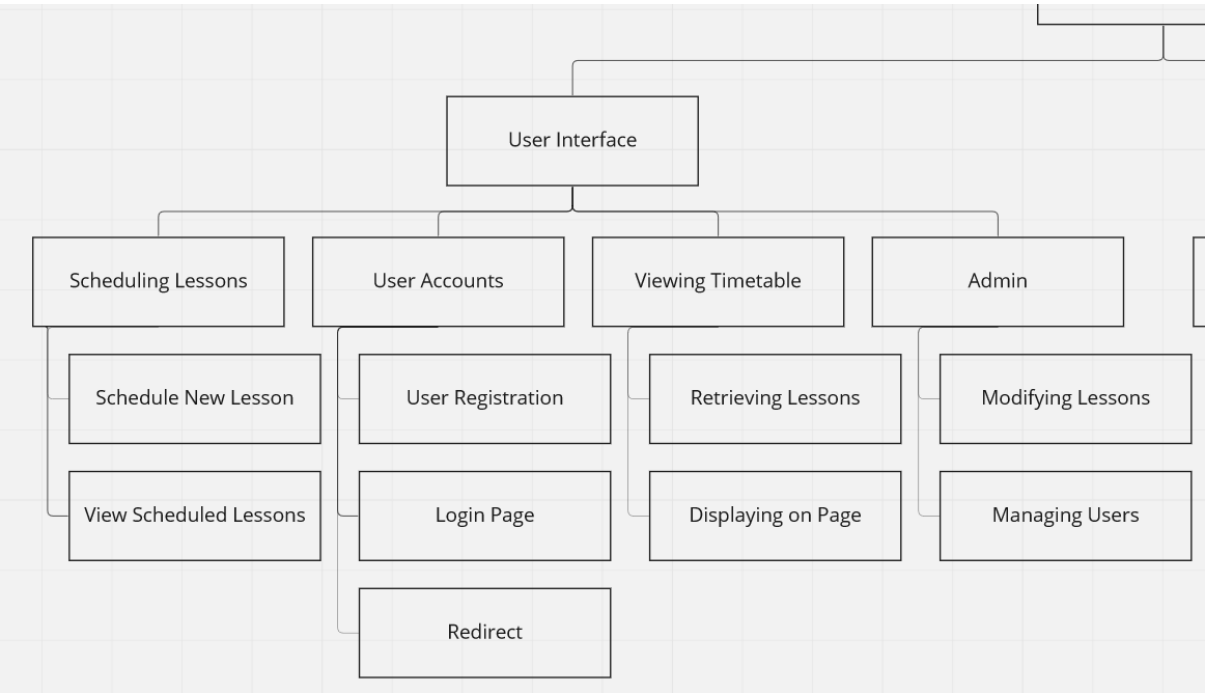


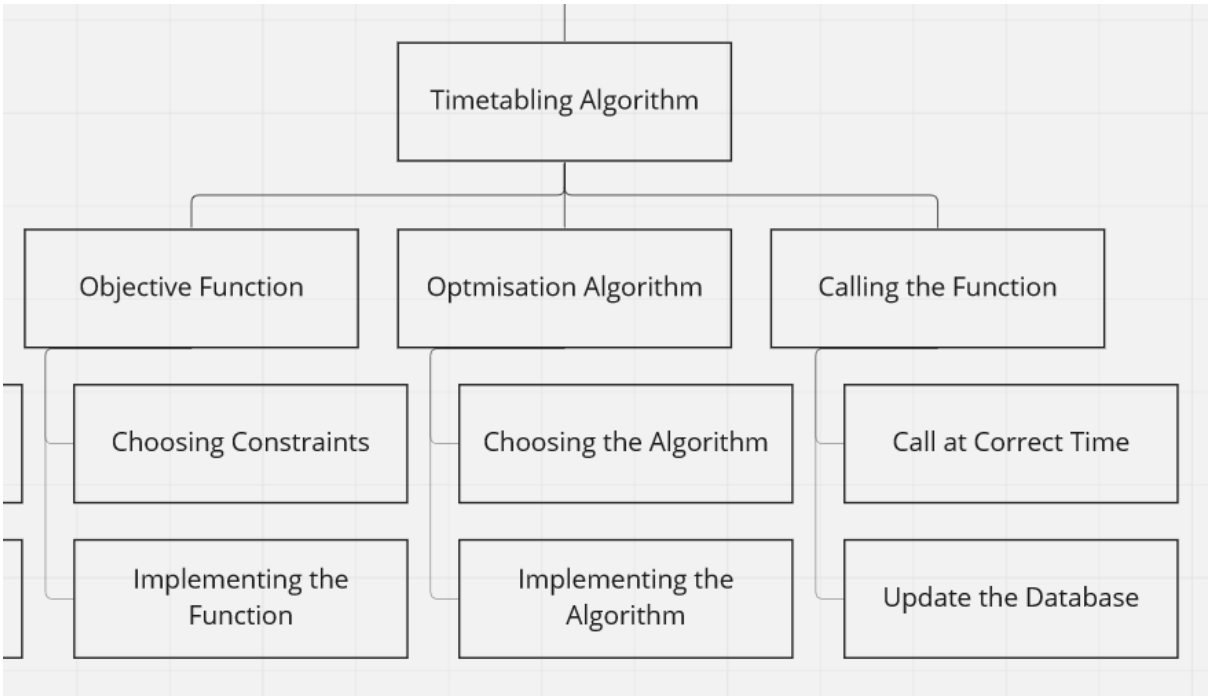
*NOTE: It is later decided in **3.3.6** that class allocation will not form a part of the final solution. It has therefore been omitted from this diagram*

The process of breaking down the problem is explained and justified throughout this section.

3.1.1.1 Close-Ups

The text in this diagram may be too small to be readable. Close-ups have been included below in case (however these include the same content present above):





3.1.2 User Interface

The user interface will comprise of a few key parts, each corresponding to a different one of the three main stakeholders. However, this is not the most effective way to break down the problem as some components can be reused between users (e.g. the timetabling interface) and the teacher interface has two parts, which would have to be broken down on the diagram.

Instead, it was broken down according to the four key parts that need to be worked on:

- Viewing timetable
    - Students and teachers
- Scheduling lessons
    - Just teachers
- User accounts
    - All types of user
- The admin interface
    - This is entirely separate from the rest of the interface

Each user interface can be broken down into three key stages:

1) Templates
    - The HTML tags are created to define the structure of the page and the elements present
    - Placeholder data is used
2) Style
    - CSS is used to make the webpage look professional and easy-to-use
    - Placeholder data is still used
3) Function
    - The interface is programmed to carry out the desired function

3.1.2.1 Viewing the Timetable

For every lesson on a given day, the interface should display:

- The subject
- The teacher
- The room
- The time

In addition to this, the interface must:

- Allow the user to choose other days in the future to view their timetable for
- Be easy-to-use on a mobile device

It should:

- Default to the current day upon page load
- Be easy-to-use on a desktop device

The interface may:

- Allow users to view their timetable for days in the past

Programmatically, this requires:

1) Retrieving all relevant lessons from the database
2) Displaying these on the page

3.1.2.2 User Accounts

For there to be user accounts, they must first be created. This will be done on the admin interface. Next, users should be able to login to their accounts via a login page. The credentials need to be checked against those stored in the database. Once authenticated, the user should be redirected to the appropriate page.

Extra care should be taken to ensure industry standard security is being used to protect user information, particularly with those attacks specifically mentioned in the requirements specification.

3.1.2.3 Scheduling Lessons

This will be exclusively available to teachers, allowing them to choose the duration of all their upcoming lessons. This will be divided further into two parts:

- The form to input the duration of upcoming lessons
- The list of lessons themselves

The form must take in the duration of the lesson and the class to allocate it to. It could also take in a title of the lesson to help the teacher plan their lessons – this is in the requirements specification as an optional requirement.

The list of lessons should clearly indicate when lessons are fixed (so cannot be modified), and should not display any lessons that have already happened.

3.1.2.4 Admin Interface

The goal for the admin interface is to provide full control over the timetabling process. As it is not entirely clear exactly how the solution will be implemented, this will be broken down further and implemented later on.

3.1.3 Timetabling Algorithm

The timetabling algorithm will take the form of a function that can be called each day to update the timetable. A heuristic algorithm is then required to achieve a feasible solution in a reasonable amount of time. However, this is impossible if there is no objective function to rate how good each solution is. These can be treated independently, with the heuristic algorithm calling the objective function where appropriate. The database can then be updated with the new lessons created as a result of this function.

The timetabling algorithm will consist of:

- A heuristic algorithm
- An objective function
- Other code to integrate this algorithm into the wider project

Some constraints are given to the input of the algorithm to make it easier to find a solution:

- Time will be in discrete chunks of 5 minutes

    - Even if it was continuous, most teachers would input multiples of 5 anyway for lesson times. It is difficult to predict to a greater precision than 5 minutes how long the lesson will last.
- It can be assumed that the input will represent a normal school, so, for example:
    - Each class will not contain more than 30 students

3.1.3.1 Objective Function

Formulating an objective function can be broken down into the following steps:

1) Decide on the features of a good solution
2) Assign these each a score based on how important they are to a good solution
3) Implement the function in code

The first two steps are explored in the design phase, whereas the third is in the development phase.

It is important to break it down like this as it is not possible to complete each step until the previous has been completed. Carefully choosing the features before implementing the function will lead to better results from the timetable.

3.1.3.2 Algorithm

The implementation of this algorithm will depend on which type of algorithm is chosen. This is explored in **3.4.1**

3.1.3.3 Implementing into the Wider Project

The timetabling function must be called early enough to have the timetable ready for the next day. It can be assumed that the server will be running 24/7 so can begin during the night to be ready for the next morning.

Once the result is received, it must be added to the database by iterating over all lessons with a for loop and adding them to the database.

3.2 User Interface

3.2.1 Admin Interface

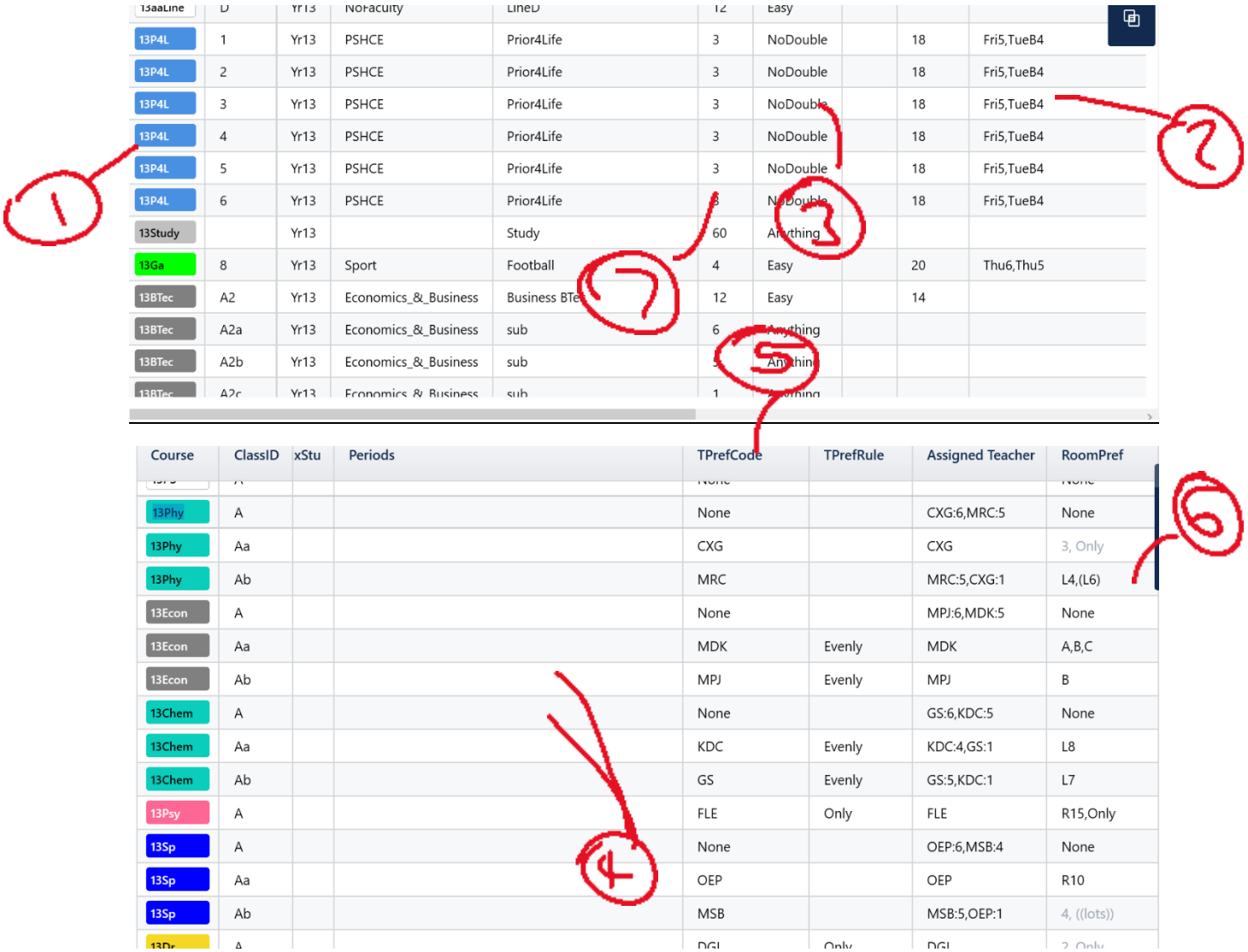
3.2.1.1 Investigating Existing Solutions - Edval

As the software will be used by people that are not necessarily familiar with IT, the user interface must be as simple to use as possible.

To learn more about creating a great user experience, a case study will be conducted on a competing product, Edval, to find out how it creates a good user experience. Edval is used in many schools and delivers a user interface that is simple to use even for those with little experience with IT.

3.2.1.1.1 List of Classes

First of all, there has to be an interface for the head of timetabling to easily manage the lessons, including adding and removing them. This is best done in a column format so it is quick and easy to get specific information about lessons.



13aaLine	D	Yr13	NoFaculty	LineD	12	Easy			
13P4L	1	Yr13	PSHCE	Prior4Life	3	NoDouble		18	Fri5,TueB4
13P4L	2	Yr13	PSHCE	Prior4Life	3	NoDouble		18	Fri5,TueB4
13P4L	3	Yr13	PSHCE	Prior4Life	3	NoDouble		18	Fri5,TueB4
13P4L	4	Yr13	PSHCE	Prior4Life	3	NoDouble		18	Fri5,TueB4
13P4L	5	Yr13	PSHCE	Prior4Life	3	NoDouble		18	Fri5,TueB4
13P4L	6	Yr13	PSHCE	Prior4Life	3	NoDouble		18	Fri5,TueB4
13Study		Yr13		Study	60	Anything			
13Ga	8	Yr13	Sport	Football	4	Easy		20	Thu6,Thu5
13BTec	A2	Yr13	Economics_&_Business	Business BTec	12	Easy		14	
13BTec	A2a	Yr13	Economics_&_Business	sub	6	Anything			
13BTec	A2b	Yr13	Economics_&_Business	sub	3	Anything			
13BTec	A2c	Yr13	Economics_&_Business	sub	1	Anything			

Course	ClassID	xStu	Periods	TPrefCode	TPrefRule	Assigned Teacher	RoomPref
13Phy	A			None		CXG:6,MRC:5	None
13Phy	Aa			CXG		CXG	3, Only
13Phy	Ab			MRC		MRC:5,CXG:1	L4,(L6)
13Econ	A			None		MPJ:6,MDK:5	None
13Econ	Aa			MDK	Evenly	MDK	A,B,C
13Econ	Ab			MPJ	Evenly	MPJ	B
13Chem	A			None		GS:6,KDC:5	None
13Chem	Aa			KDC	Evenly	KDC:4,GS:1	L8
13Chem	Ab			GS	Evenly	GS:5,KDC:1	L7
13Psy	A			FLE	Only	FLE	R15,Only
13Sp	A			None		OEP:6,MSB:4	None
13Sp	Aa			OEP		OEP	R10
13Sp	Ab			MSB		MSB:5,OEP:1	4, ((lots))

[1] Lessons are colour coded, making it very easy to spot similar lessons or look for a particular subject if you know the colour coding. These do appear to be reused which may make it worse, although it is better to reuse distinct colours than to have many colours that are hard to discern.

[2] This is a great format as it is really compact but also communicates very clearly exactly when the lessons are scheduled.

[3] Using words here (rather than numbers or confusing letter codes) reminds the user of what each one means which speeds up timetabling.

[4] The table alternates between two slightly different shades which greatly improves readability along the table - it's much clearer which items are in which row.

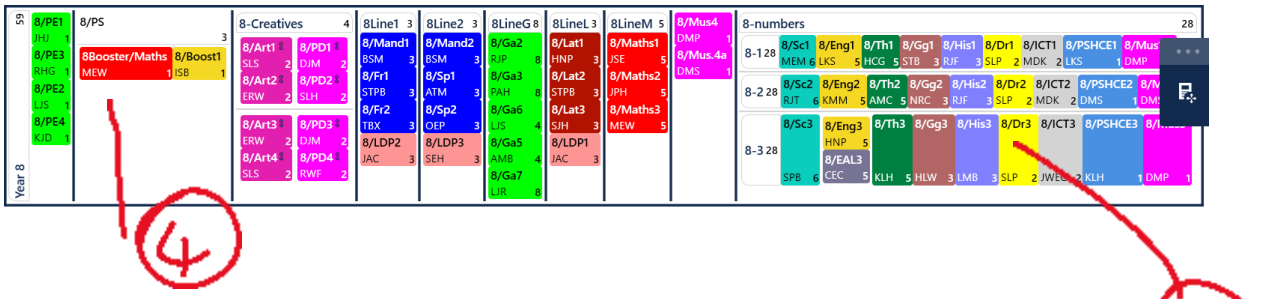
[5] This allows specifying teacher preferences, so that teachers can teach what they prefer teaching

[6] Specifying a room preference also allows the timetable to be as good as possible as teachers are able to stay in the same room

[7] This column allocates how many lessons per fortnight for each subject, although the column is not clearly named. While this is probably the fault of the user, the software should more clearly name things to make it easier to use.

3.2.1.1.2 Blocks

This interface organises the information clearly. However, this would not be appropriate for the new method of timetabling which does not use a traditional block method. Instead, a chronological view could be more appropriate that shows the lessons in a better format.



[1] These subjects are timetabled as one subject, as students are in the same class for all of these subjects. This gives much more flexibility while timetabling. This is very clearly indicated on the user interface.

[2] These blocks are timetabled differently because everyone does these at the same time and has many options to choose from

[3] Classes are represented by a year group, a block, and an abbreviation for the name of the subject. This uniquely represents any class throughout any year group or block, and is very effective

[4] Teacher names are abbreviated to initials to save space onscreen.

3.2.1.1.3 Teacher Timetables / Workload

Name	MaxLoad	LdLeft	Score	13	12	11	10	9	8
[redacted]	44	0	6	6 13C1b/Gg	6 12Da/Gg	5 11/Gg1	6 (10/Gg2)	4 9/Gg1 4 9/Gg4	3 8/G
[redacted]	44	0	0	5 13D1a/Gg 5 13C1a/Gg 4 13/Ga.10		5 11/Gg2	6 10/Gg3	4 9/Gg3	3 8/Gg2
[redacted]	44	2	0	6 13/Ga1 6 13D1b/Gg	5 12Db/Gg	5 11/Gg3	6 10/Gg1	4 9/Gg2	3 8/Gg1
Unassigned	0								
Total	132	2	6	Periods 100	Allowances 20	Max duty 0	Load margin 9.1%	From other faculties Gives 10, takes 0 periods	

Another important bit of timetabling is balancing the load for each teacher. This should be handled by the timetabling algorithm, however it may still be necessary to have a screen to monitor the effectiveness of this. This interface could be added should there be sufficient time to do so.

[This feature was not pursued further beyond this point]

3.2.1.2 Requirements for the Admin Interface

The admin interface is essential to serve as a backup option in case things go wrong. This should give the head of timetabling full control, including all functionality offered by the teacher and student interface.

It must:

- Allow access to the student interface on behalf of any student
- Allow access to the teacher interface on behalf of any teacher
- Display a summary view of the timetable

It should:

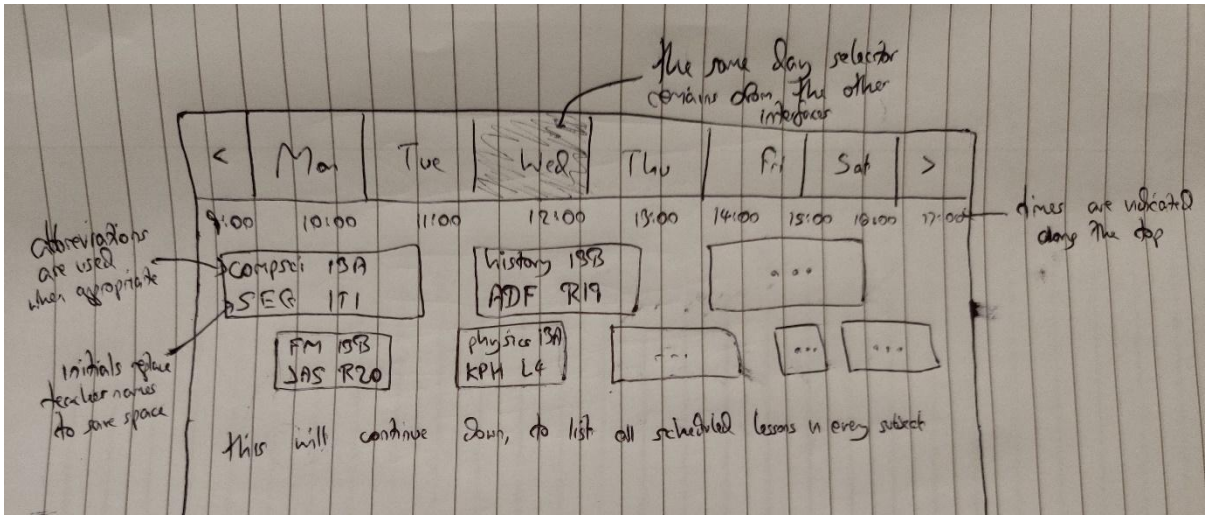
- Allow any decision made by the algorithm to be overridden

As the student and teacher interfaces can be reused, only the summary timetable view is necessary. This will allow them to check that everything is working.

As with the teachers, the head of timetabling will be primarily using a computer and so a mobile interface is unnecessary. Nonetheless, a functional mobile interface is a benefit.

3.2.1.3 The Admin Interface

The admin interface will be as follows:



As annotated, this will provide more dense information therefore some abbreviations have been used where appropriate.

This interface is excellent because it clearly shows lots of information in as little space as possible. It allows clashes to be visualised easily and gaps are very evident.

Some of the usability features present are as follows:

- Abbreviations are used
    - This increases the information density that can be displayed
- Buttons are used to select days
    - This is done in a way that is consistent with other interfaces, so it is easy to understand and use the interface
- Lessons are arranged clearly in chronological order
    - This makes it quick to find specific lessons or identify issues with the timetable

3.2.2 Student Interface

3.2.2.1 Requirements for the Student Interface

The student interface must:

- Allow students to view their timetable

The student interface could include:

- Account registration

The following activities are outside of the scope of the solution:

- Students choosing subjects
- Asking for help / raising a support ticket if there are any issues

This interface will be accessible by a website on a mobile device, to allow students to easily access their timetable.

3.2.2.2 Investigating Existing Solutions

3.2.2.2.1 SchoolBase – Desktop

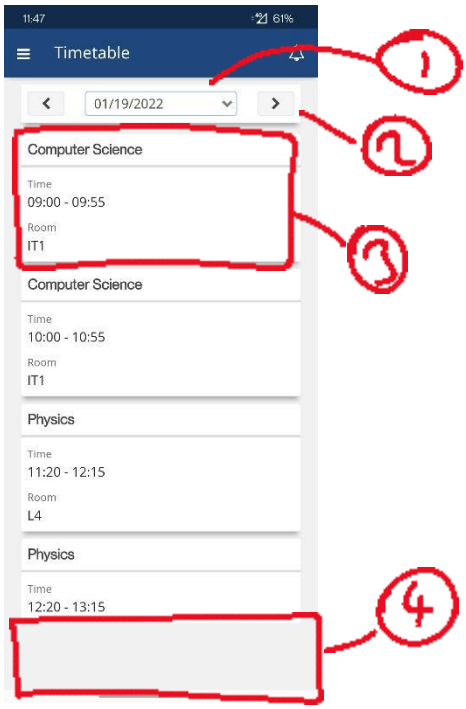
SchoolBase is a widely used school management software, including support for students to view their timetable online.

The interface as viewed on desktop (teacher names redacted) is:

This interface is good because it shows all lessons in a logical format, with appropriately sized gaps between them. However, this is the interface as viewed on desktop, which is inconvenient to check quickly. Therefore, the main focus should be a mobile interface, while a desktop interface could follow subject to sufficient time.

3.2.2.2.2 SchoolBase – Mobile

The mobile interface is significantly worse:



[1] This clearly indicates the current date, however it does not specify the day of the week. Searching for a specific day, in the current week or another week, is difficult without looking up the target date or calculating when it would be. It is also clickable to allow selecting a date far in the future, however this will not be necessary for the proposed solution as the timetable will not be finalised far into the future.

[2] These buttons allow switching between days. While these are poorly implemented into this app, they are a good idea to allow switching forwards and backwards a few days

[3] This box for the lesson is far bigger than it needs to be. As a result, not even the whole timetable for one day manages to fit on a large phone screen (only about 3-4 lessons can fit on the screen at once). This is entirely unnecessary and could easily be fixed to give more space. Also, while taking up lots of space, this interface simultaneously omits key information such as the teacher who is teaching each lesson. Additionally, it is very difficult to visualise how much free time is available as the information is not spaced out accordingly.

[4] This space is completely empty for no apparent reason (there are more lessons below this)

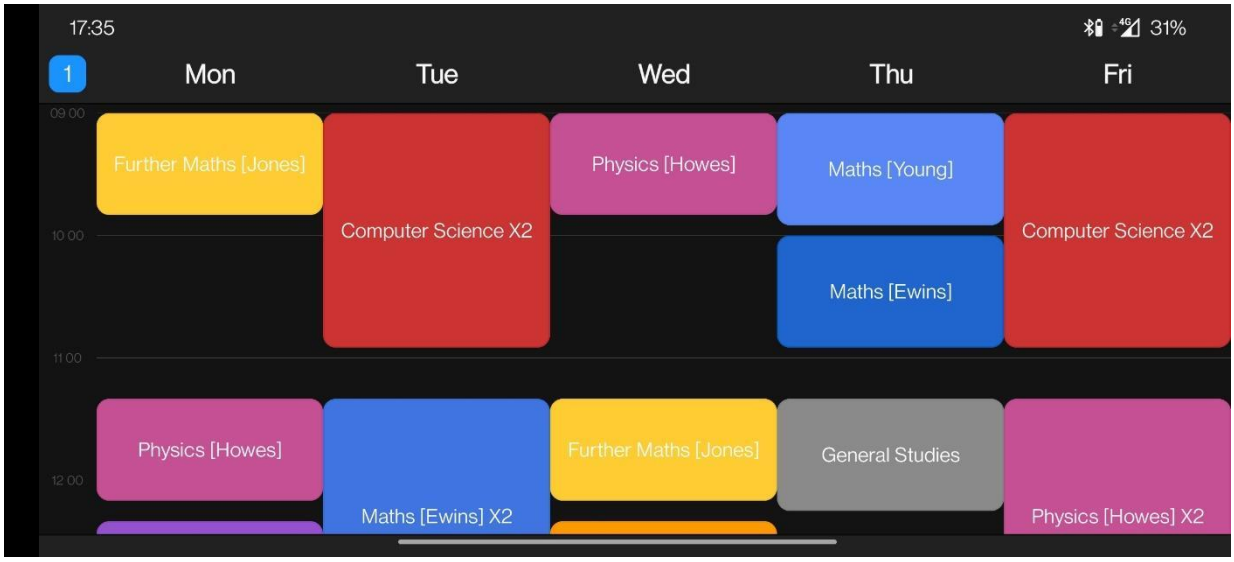
Clearly, this interface is significantly worse than the desktop interface.

3.2.2.2.3 A Different Timetabling App

An example of a much better timetabling interface is as follows:

This interface is always able to fit all the lessons onscreen as it is more compact and makes it much easier to select the days of the week. Colour is used to allow fast identification of lessons, and teachers are displayed (even if these had to be manually entered into the subject name field). The lessons can be clicked on to display more information. It is still not particularly easy to visualise free time, as it is on the desktop version.

However, this interface has a big advantage in terms of what happens when it is rotated sideways:



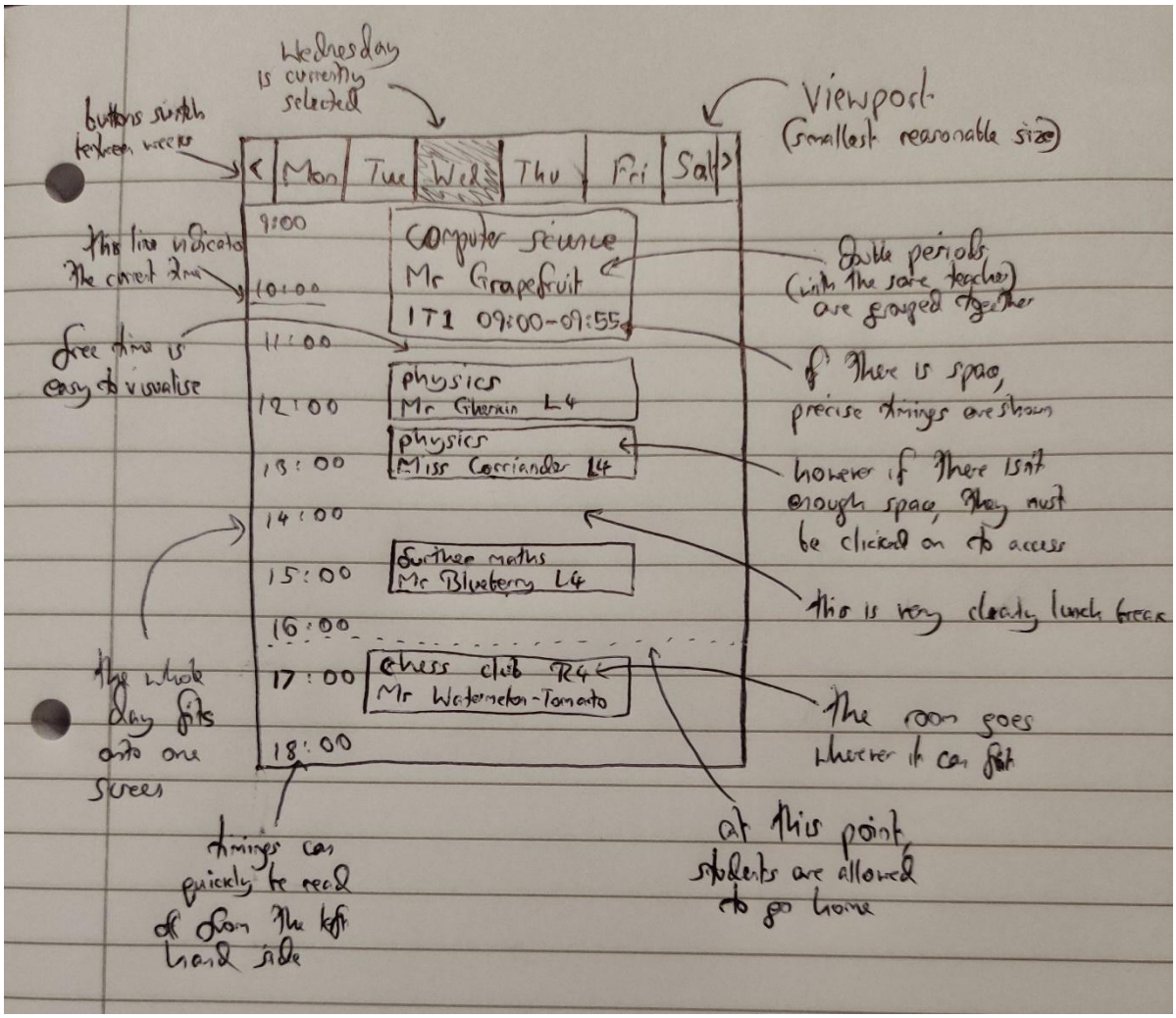
This makes it very easy to visualise a whole week, including easily visualising gaps. It still requires scrolling to view the whole day, however this is much more similar to the desktop interface (which is great). It could be argued this is an improvement over the desktop interface as it is more simple and uses colour effectively, however there isn't as much information on display here.

This is a third-party application, which requires students to type in their full timetable into the app beforehand. Clearly, this is not possible with a varying timetable, so it is important that the official solution matches the quality of these third-party solutions so that anyone currently using a third party solution will be satisfied with the new timetable interface.

While it would be ideal to implement a different interface for landscape, it will be subject to time limitations. As it may only be possible to implement one, the portrait one would be more sensible to prioritise because it will be used far more often. Equally, while it would be ideal to have both a desktop and mobile version of the interface, there is only sufficient time for one of these. As the priority is to allow students to easily access their timetable, the mobile version will be developed first.

3.2.2.3 The Student Interface

The current design for the mobile version is as follows:



This is an abstraction of the proposed solution. Abstraction helps in planning the solution by hiding unnecessary detail so that it is possible to focus on what matters. However, the abstraction has hidden a few details, namely:

- Colour: Each lesson will be represented by a different colour in the proposed solution
- All of the dimensions are done approximately
- Font sizes differ in this picture, however there are no plans for these to differ in the final solution
- This picture does not convey how it will vary on devices of different sizes. It is planned that it is stretched according to the screen size, such that the whole day always fits onscreen, however this will depend on the results from testing to inform development.
- The teacher names have been redacted and replaced with placeholder names of similar length. The specific teacher names are not important to the solution.

This is an ideal user interface because:

- No space is wasted – every bit of available screen space is used effectively
    - In addition to this, the whole timetable fits on one screen so no scrolling is required. Scrolling wastes the user's time and makes it much more difficult to look at the whole day at a glance.
- It is very easy to glance at the timetable and notice how long lessons are and where there are gaps. This makes the precise timings mostly unimportant, however these can always be accessed with a single tap if necessary
- All important information is either onscreen or only a single tap away
- Viewing the timetable for different days in the current week or the following week is very intuitively laid out and requires at most two button presses.
- The interface is clear and easy to understand, such that even those not as familiar with technology will be able to use it with ease.

3.2.3 Teacher Interface

3.2.3.1 Requirements for the Teacher Interface

The teacher interface will facilitate teachers being able to view their timetable as well as schedule lessons for the future.

It must:

- Allow teachers to view their timetable
- Present a form in which teachers can add more lessons to be scheduled

It should:

- Allow teachers to manage the lessons they have already put to be scheduled (provided they have not already been scheduled for the next two weeks)

It could:

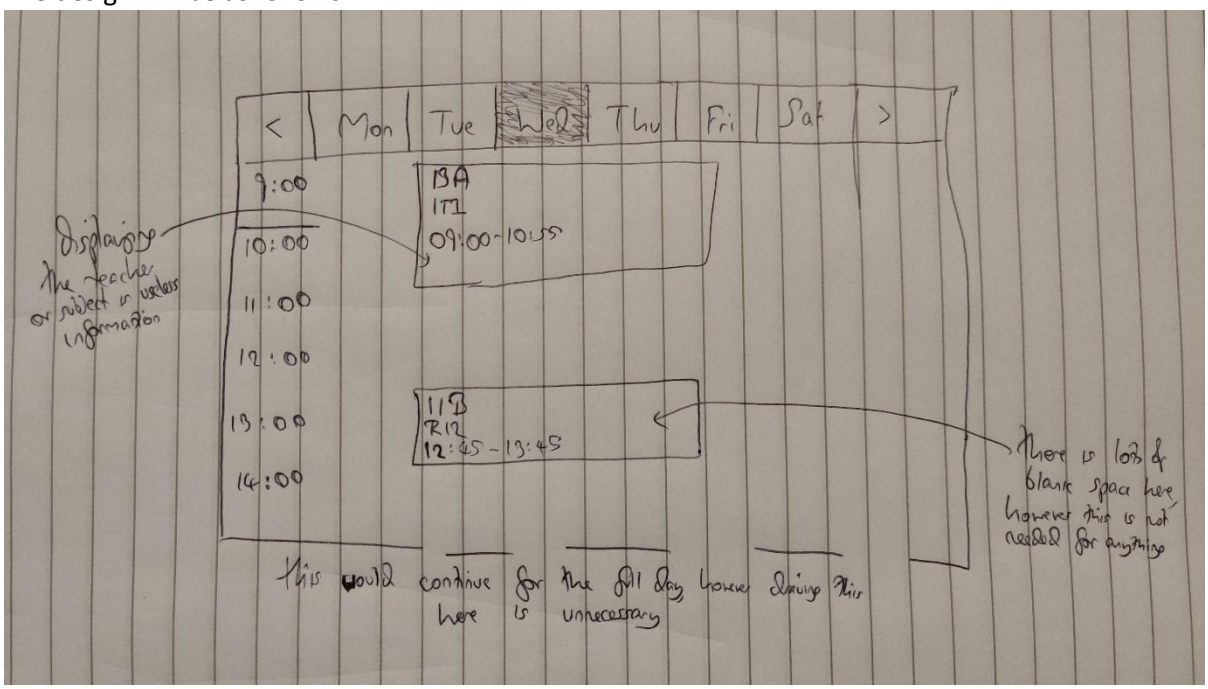
- Allow access to the student interface on behalf of any student

In contrast to the student interface, the teachers are almost exclusively working on laptops so it would be convenient to have a decent desktop interface available. However, teachers are still able to check their timetable on their phones, which may often be more convenient. The interface must adapt depending on the type of device being used.

3.2.3.2 Viewing Timetable

The first part of the interface, viewing the timetable, is rather similar to the student interface, such that this could almost be copied.

The design will be as follows:



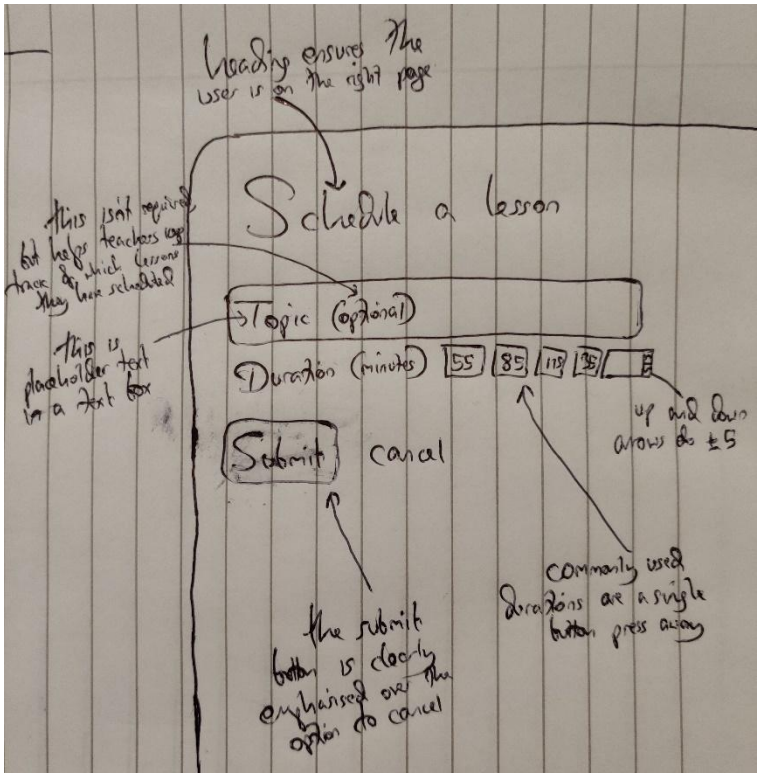
Clearly, this design is almost identical to the student interface. Consequently, this design shares in all of the benefits as the student interface, while also reusing components where possible. Reusable components improve maintainability and reduce the development time of the project. They also make the user interface more consistent so that it is easier for users to understand.

The key change from the student interface is simply what is displayed for each lesson – while the student interface had to display subject and teacher name, this is completely meaningless information to the teacher themselves. This information has been replaced by which class the teacher will be teaching. This interface may also be able to display what the teacher is planning to teach the lesson on, if they have inputted this when scheduling lessons.

3.2.3.3 Scheduling Lessons

The second part of the teacher interface must allow teachers to schedule their lessons, choosing how long they want for each lesson. This should be very simple, fast and easy-to-understand to avoid teachers refusing to schedule their lessons. To avoid overcomplicating the process, there will not be any additional unnecessary features other than the minimum that teachers need to complete before they can continue with other work.

The proposed design for scheduling lessons is:



The goal with this interface was to ensure ultimate simplicity so it is as quick and easy to schedule lessons as possible.

Usability features are clearly annotated on the image.

3.3 Allocating Classes

3.3.1 The Need for Class Allocation

Class allocation refers to the process of assigning one teacher to a group of students which can be taught together.

Class allocation is requirement **17**, under 'Potential Extra Features' (requirements specification: **2.7.3.3**). While it is possible to use other software to allocate classes, it is worth considering as a possibility as:

- Schools would be able to only use one piece of software, rather than two
- This algorithm could be optimised to give a result more suited to the timetabling algorithm

It will be explored further in this section.

3.3.2 Description

Due to the way schools work, classes must be decided at the start of the year and cannot be changed after the school year begins. This means it's critical to create an algorithm that makes it as

easy as possible to schedule lessons throughout the school year, otherwise the timetabling algorithm may fail to find solutions without many clashes. If the class allocation is poor, then timetabling will not work.

While the traditional block method works well with traditional timetabling, it has one main weakness which is that some subject combinations are simply not possible. Every year, a few students will be told that they are unable to do their desired subject combination. This is not good and is something that should be addressed with the new timetabling system, by allowing every single combination of the offered subjects. This may require a few clashes to facilitate but is overall far more beneficial than a student not being allowed to take a given subject. Crucially, students will be informed if they may experience clashes (in the old system, this would be equivalent to being told that they cannot choose that subject) and if clashes occur only those students will be affected. This means that students will only experience clashes if they choose unusual subject choices and will be given an opportunity to change if they would like to avoid infrequent clashes.

3.3.3 Desired Outcome

This will be facilitated by an algorithm which produces the following variant on the traditional block system:

- N blocks which contain classes with roughly the same number in each block with absolutely no conflicts between any other block. These will be referred to as the "primary" blocks
- Additional blocks which contain any classes that could not be allocated into the primary blocks without conflict. These will have a value corresponding to the number of conflicts when timetabled with every other block. Usually, this value will be very high for N-2 blocks but small for two blocks so that it can be scheduled alongside those blocks with few clashes. These will be referred to as "secondary", "overflow" or "clash" blocks

NOTE: In most schools, N = 4

The function will be given an input corresponding to all students' subject choices, as well as predefined constants corresponding to the maximum and minimum class sizes. It will output the classes/blocks to be used while timetabling.

3.3.4 Draft 1

Problem decomposition was used to break the problem down into smaller sections which can be worked on separately.

The first draft for the algorithm is as follows:

1. Combine subjects with zero correlation
    a. This is to simplify the problem
    b. These subjects can then be timetabled together, as they do not clash at all
2. For every class above the maximum class size, add an additional class for that subject to a new block
3. Repeat step 1, allocating students between the classes in a way that remains no conflicts

    a. This must give the solution with the fewest number of overall blocks. Classes should be distributed unevenly between blocks, so that as many students are in as few blocks as possible. Class sizes will always be between the set minimum and maximum but will aim to be distributed evenly between classes.
    b. As this is only executed once over the course of a whole year, an algorithm with a high complexity may still remain feasible, if necessary to generate an optimal solution.
4. Designate the four largest blocks 'main blocks'. For every conflict:
    a. If above the minimum class size, add a new class in a different block to resolve the conflict
    b. If below the minimum class size, move affected classes to an overflow block. Store the number of conflicts between this block and all main blocks
5. Output the blocks and which student is in each class within those blocks

The first two steps can be represented with the following pseudocode:

```
// in this code, 'dict' represents a structure with key-value pairs, where
my_dict['key'] would access the value corresponding to 'key'

const MAX_SIZE = 24
const MIN_SIZE = 4

function allocate_classes(dict students):
   // students should be given in the form {student: [subject]}, where
student is any unique identifier for that student and all values are
strings

   // TODO: INPUT VALIDATION HERE

    // STEP 0: SETUP KEY VARIABLES

   correlation = {} // a 2 dimensional dict in the form {subject1:
{subject2: conflicts}}, where conflicts is an integer representing how many
people have chosen both subject1 and subject2
   // by definition, correlation[subject1][subject2] =
correlation[subject2][subject1]
   subjects = {} // in the form {subject: [student]}
   for student in students: // this iterates over the keys in the
'students' dict
      for subject1 in students[student]:
          subjects[subject1].push(student)
          if subject1 not in correlation:
              correlation[subject1] = {}
          for subject2 in students[student]:
              if subject2 == subject1:
                  continue
              correlation[subject1][subject2] += 1

    //  STEP 1: COMBINE ZERO CORRELATION

    // using the correlation matrix, pair up subjects with zero correlation
    blocks = []
```

```
    allocated = [] // contains which subjects have already been allocated
    for subject1 in correlation:
        for subject2 in correlation[subject1]:
            if correlation[subject1][subject2] == 0:
                if subject1 not in allocated and subject2 not in allocated
then
                    blocks.push([subject1, subject2])
                else if subject1 in allocated:
                    // add subject2 to the same block as subject1
                else if subject2 in allocated:
                    // add subject1 to the same block as subject2
                endif // else, both are already allocated
            else
                if subject1 not in allocated:
                    blocks.push([subject1])
                if subject2 not in allocated:
                    blocks.push([subject2])
            endif

    // STEP 2: DIVIDE CLASSES THAT ARE TOO LARGE
    for subject in subjects:
        classes = math.ceil(subjects[subject].length / MAX_SIZE)
        if classes > 1:
            for x in range(classes-1):
                blocks.push([subject])
```

However, once step 3 was reached, it became clear that this algorithm would not be the best solution to the problem. The main problem with this algorithm is that it is quite inefficient by having many separate steps when it would be more efficient to combine these steps together. Additionally, the best solution from step 3 may not always result in the best solution to step 4, so more possibilities will have to be tested.

3.3.5 Draft 2

Instead, it may be more appropriate to use a recursive algorithm to try all possibilities in one go. A recursive algorithm is necessary as an optimal solution (or at least close to an optimal solution) is required. As these classes will be used for the whole year, it is vital they are as good as possible to facilitate timetabling throughout the year. A significant computational cost, provided it remains feasible, will be insignificant in comparison to the length of time it will be used for.

Draft 2 of the class allocation algorithm:

- For each subject:
    - For each main block:
        - Consider allocating the subject to that block
            - If there are conflicts, the following possibilities should be considered:
                - Splitting up the conflicting classes in the block
                    - Some students may need to be fixed in specific classes

- Splitting up the class to be allocated to the block
    - Some students may need to be allocated to specific classes
- Not resolving the conflict
    - These will be moved to an overflow block later
    - Only if below a certain threshold of conflicts

This algorithm will clearly have a high complexity, as each subject gives four blocks to be explored, which may each have additional possibilities to consider. However, this is necessary to achieve the desired optimal solution.

3.3.6 Rethinking Class Allocation

While class allocation would have benefits, it became more apparent as development continued that the class allocation algorithm was very similar to many algorithms already available on the market. Aside from some minor variations, it is essentially allocating classes to blocks as with most other software. As this would not offer anything unique in the project, there is little reason to focus on this when more time could be allocated to the unique parts of the project such as the timetabling algorithm. The goal of the project is to create a unique timetabling system – duplicating functionality already in existing solutions is not a good use of time.

It is for this reason that the class allocation algorithm will not be pursued further in this project.

3.4 Timetabling Algorithm

3.4.1 Exploring Different Approaches

Timetabling is made very difficult as there are a significant number of possibilities to be checked. The problem of finding the optimal solution is NP-complete, meaning it is simply infeasible to determine the optimal solution. This means a heuristic must be used to achieve a solution that is good enough in a reasonable amount of time.

There are a few different approaches commonly used for nonlinear discrete optimisation problems like scheduling, which will be explored below.

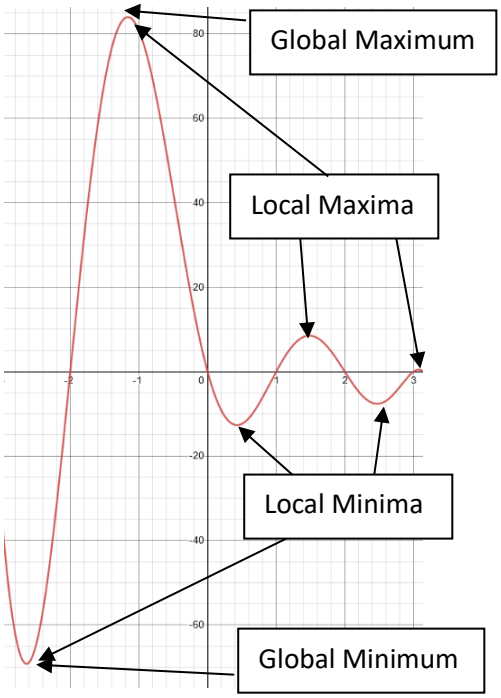
3.4.1.1 Hill Climbing

A basic hill climbing algorithm could be as follows:

1) Start with a random configuration of lessons
2) Using the derivative of the objective function, modify the input parameters to increase the value of the objective function
3) Repeat step **2** until the value converges
4) After this value converges on a local maximum, output the timetable

NOTE: Instead of using the derivative, it would be possible to simply choose random points and test each one, however this is significantly slower, especially with higher-dimensional objective functions as what will be the case with timetabling.

While this algorithm seems to work and finish in a reasonable amount of time, it only finds a local maximum. While changing any aspect of this solution leads to a worse solution, a different initial state could lead to an entirely different solution, which could be even better.



In this diagram, the single decision variable was plotted on the x axis and the value of the objective function was plotted on the y axis.

If this algorithm began at x = 1, it would likely converge to the local maxima of 8.5, whereas if it had begun at x = -1 it would find the actual global maximum of 84.

Clearly, a more sophisticated algorithm will have to be used that explores more of the domain to find the global maximum, or at least a good local maximum.

Advantages	Disadvantages
Fast	Is often trapped in a local optimum
	Rarely gives the optimal solution
	Requires the derivative of the objective function

3.4.1.2 Simulated Annealing

Simulated annealing is an approach that simulates the slow cooling of a metal to obtain an optimal solution. It functions in a similar way to the basic algorithm discussed earlier however initially searches more broadly for solutions before 'cooling down' and executing smaller changes to the domain.

While the temperature *T* is large, the algorithm is more likely to explore points that may decrease the value of the objective function in the short term. As it cools, the algorithm will begin to search for points that are closer to the current solution, until it reaches minimum temperature. If cooled too quickly, it is likely to reach a local maximum that is not the global maximum (as in the hill climbing algorithm). If cooled too slowly, the algorithm will not complete in a reasonable amount of time.

The algorithm can be defined more formally as:

1) Start with a random solution
2) Generate a new solution
    a. If it increases the value of the objective function, always accept the new solution
    b. If not, use the current temperature value to determine whether to accept or reject the new solution
3) Repeat from step **2** until the predefined number of iterations is reached / the solution is better than a predefined threshold
4) Output the solution

Advantages	Disadvantages
Avoids being trapped in local minima.	Can be very slow, especially if objective function is expensive
Works well with nonlinear problems involving lots of constraints	Cannot tell whether the solution is optimal
Allows customising the execution time at the expense of the quality of solutions	Requires predefining the cooling rate which has to be chosen carefully before starting

3.4.1.3 Particle Swarm Optimisation

This method takes influence from biology, simulating the idea that a flock of birds work together to get towards an optimal solution.

The algorithm is as follows:

- Begin with a number of random points (particles)
- For each particle:
    - Use a heuristic to determine the direction to move in
    - Also move towards other particles (birds)
        - This allows them all to work together to locate a good solution
- After a predefined number of iterations, the best solution found so far is deemed the final solution

Advantages	Disadvantages
Can stop at any time and obtain good solution	Higher memory usage as many particles have to be simulated at once
Avoids being trapped in a single local minimum	Cannot tell whether the solution is optimal
Works well with nonlinear problems involving lots of constraints	
Does not require obtaining the derivative of the objective function	

3.4.1.4 Tabu Search

Tabu search explores many different points, exploring the local optima for each, while creating the tabu list to prevent selection of solutions which have already been visited.

An abstract version of the algorithm can expressed as:

- Generate a set of candidate solutions. For each solution:
    - Locate the local optima using the hill climbing method
    - Add to the tabu list to prevent it being visited twice
- Choose the best solution and use this to generate the next set of solutions
- Once a certain number of iterations has been reached, output the solution

Advantages	Disadvantages
Can stop at any time to obtain the current solution	Requires the derivative of the objective function
Avoids being trapped in a single local optimum	
Tabu list prevents searching already visited solutions	

3.4.1.5 Genetic Algorithm

Genetic algorithms work by simulating evolution by natural selection.

A simplified version of the algorithm is as followed:

1) Select a population of random points
2) Select the best to become parents in the mating pool
3) These parents then mutate, producing offspring
    a. Mutation – some values are randomly changed
    b. Crossover – the genes that come from each parent are randomly determined
4) Repeat from step **2** with the new population until the desired stopping condition is reached. The stopping condition could be:
    a. The algorithm converges on a solution / difference between values in each iteration is small
    b. Predefined number of iterations is reached / time elapsed
    c. Value of objective function increases past a threshold
5) Output the solution

Advantages	Disadvantages
Can be parallelised	Input parameters such as rate of mutation, population size must be carefully chosen or the algorithm may not converge.
Can deal with very complex problems	
Does not require the derivative of the function	

3.4.1.6 Choosing a solution

Out of all available solutions, the hill climbing algorithm is the worst as it is unlikely to find a good solution and often gets trapped in local optima. Tabu search requires the derivative of the objective function which will be very difficult to compute with a discrete highly nonlinear problem such as timetabling. While the other three algorithms would be suitable, genetic algorithms have been shown to provide a solution to similar problems in a fast time. A genetic algorithm will be pursued further to solve the timetabling problem.

More details surrounding its implementation are decided during development.

3.4.2 Defining the Objective Function

Before optimising for a given objective function, the objective function itself must be decided on. This is crucial to achieving a solution that satisfies the following requirement from the requirements specification: “the timetabling algorithm should deliver a balanced timetable that is conducive to productive work, where lessons are spread out appropriately according to sensible criteria”.

3.4.2.1 Initial Requirements

From highest priority to lowest priority, an initial set of requirements will be as follows:

#	Constraint	Justification
1	Teacher clashes	If a teacher is unable to teach a lesson, then the lesson cannot happen.
2	Student clashes	Students cannot go to multiple lessons at once, so they will miss out on their education
3	Even allocation of lesson time across subjects	Teachers must be able to rely on a certain amount of lesson time over the course of a year
4	Consecutive lessons must be separated by a gap of at least 5 minutes	This allows time for students and teachers to move between classrooms.
5	Variety of subjects throughout the day / week	Having one subject lots of times in a row and then not having it for a week is not conducive to productivity
6	Maximum lesson time on a day	Students / teachers will be tired if they have too many lessons in a single day. Appropriate class allocation along with constraint **3** will ensure a sensible minimum amount of lesson time is achieved
7	Gaps should be either: - 5 minutes (so lessons are back-to-back) - Long (20+ minutes) to allow time for study	Short 10-minute breaks do not offer enough time to do work, so this time will likely be wasted as people are left waiting for their next lesson
8	Early finish time	People like finishing early

3.4.2.2 Forming a Cost Function

The cost function is implemented in **4.2.1.3**.

To formalise this into an objective function, each should be assigned a score based on how bad the issue is. Then, the algorithm could minimise the value of the objective / cost function.

The value of the cost function will not be restricted to integer values and will be stored as a floating point number.

Constraints marked ***** have been explored further below

#	Constraint	Cost Value + Formal Definition
1	Teacher clashes	**Summary** Any arbitrarily large number +10,000 for each teacher clash **Formal Definition** n = number of teacher clashes k = points per teacher clash = 10000 $C_1(n) = kn$
2	Student clashes	**Summary** +100 for each student clash **Formal Definition** n = number of student clashes k = points per student clash = 100 $C_2(n) = kn$
3*	Even allocation of lesson times	**Summary** Increased weighting later in the year. Every 1% off the desired allocation, +1 point **Formal Definition** This is explained in **3.4.2.2.1** x = $x_1, x_2, \ldots, x_n$ = lesson time allocated to each class so far s = $s_1, s_2, \ldots, s_n$ = lesson time intended for each class n = number of classes t = school days elapsed since start of year a = arbitrary constant = 0.4 b = arbitrary constant = 8 **NOTE**: $\frac{b}{a}$ gives the value of t where $w(t) = 0.5$ k = weighting of a 100% difference = 100 $C_3(x,s,t) = \sum_{i=0}^{n} k\, f(x_i, s_i)\, w(t)$ $f(x,s) = \frac{\lvert x-s \rvert}{s}$ $w(t) = sig(a+bt)$
4	Consecutive lessons gap	**Summary** +10 points for each occurrence **Formal Definition** n = number of occurrences k = points per occurrence = 10 $C_4(n) = kn$ **This is now being computed in constraint 7**
5*	Variety of subjects	**Summary** If there are two lessons on the same day, add a fixed number of points for each occurrence If lessons are too far apart, points scale exponentially with the time between the lessons

		Formal Definition This is explored in **3.4.2.2.2** $n_{i,t}$ = number of lessons with class *i* on day *t*; N = number of classes t_0 = start day; T = number of days $d_{i,t}$ = days between the lesson on day t and the next lesson a = arbitrary constant $$C_5(\dots) = \sum_{t=t_0}^{T+t_0} \sum_{i=0}^{N-1} (f(n_{i,t}) + g(d_{i,t}))$$ $$f(n) = max\{k(n-1), 0\}$$ $$g(t) = ka^t$$
6	Maximum daily workload	**Summary** 0 for all values up until max load, at which point it scales exponentially with the teaching time required **Formal Definition** x = x_0, x_1, x_2, …, x_{n-1}, x_n = workload of users x_i = daily workload in hours for user *i* n = total number of users k = arbitrary constant = 5 c = arbitrary constant (determines max load) = 4 **NOTE**: max workload = $clnc$ $$C_6(x, n) = \sum_{i=0}^{n-1} f(x_i)$$ $$f(x) = k\ max\{e^{\frac{x}{c}} - c, 0\}$$
7*	Gaps	**Summary** See definition of f(x) for points of each gap length **Formal Definition** Explored in **3.4.2.2.3** $x_{i,t}$ = $x_{i,t,0}$, $x_{i,t,1}$, …, $x_{i,t,n}$ = length of each gap for student *i* on day *t* k = arbitrary constant t_0 = start day; T = number of days n = number of students; $N_{i,t}$ = number of gaps student *i* day *t* a,b,c,d = arbitrary constants; a = 10; b = 5; c = 2; d = 1 $$C_7(x) = \frac{1}{k} \sum_{t=t_0}^{T+t_0} \sum_{i=0}^{n} \sum_{j=0}^{N_{i,t}} f(x_{i,t,j})$$ $$f(x) = \{a, \quad x = 0 \quad b, \quad x = 10{,}15\ c, \quad x = 20 \quad d, \quad x = 25\ 0, \quad otherwise$$
8	Early Finish	**Summary** +1 point for every hour after the start time that each individual has to remain in school, divided by some constant k **Formal Definition** t_i = number of hours user *i* has to be in school. (earliest finish time - latest start time); n = number of users k = arbitrary constant = 10 $$C_8(t) = \frac{1}{k} \sum_{i=0}^{n-1} t_i$$

It is extremely unlikely that the constant values suggested here will be the best possible values to produce the best timetable. It will be important to test the output of the algorithm throughout development and tweak these values as appropriate. All arbitrary constants involved have been identified as such and while a value has been suggested, these will need to be tweaked significantly to achieve good results. The initial value only provides a starting point for further testing.

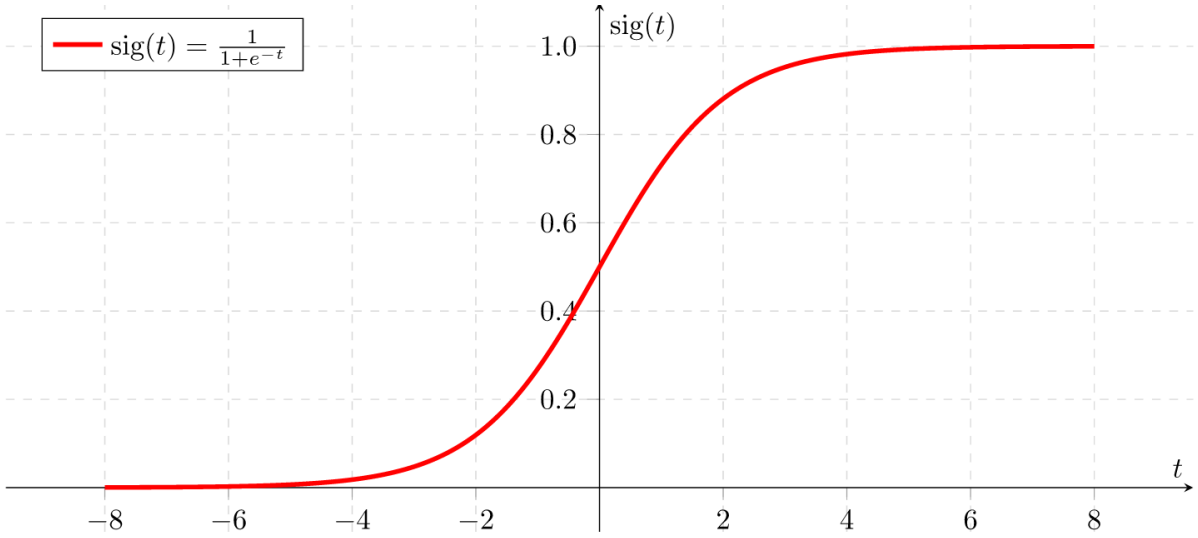
The final cost function can be evaluated by summing all of the individual cost functions, i.e.

$$C(\dots) = \sum_{i=1}^{8} C_i(\dots)$$

3.4.2.2.1 Constraint 3 – Even allocation of lesson times

This constraint will vary depending on the position through the year. At the beginning of the year, this constraint is effectively irrelevant and relying on it will lead to very poor timetables (as the percentage difference in allocation will be very large). However, as the year progresses, it will become more significant as there has been more time available for each subject to accumulate time.

To weight the value of this, a sigmoid function would be appropriate as it starts small, gradually increasing until reaching a maximum and remaining at that for the remainder of the year. A graph of a sigmoid function is below:



This function is easily differentiable and could be approximated with a Taylor series should this make computation faster.

The function could be transformed so that it is almost 0 for the first few weeks of the year, then slowly increases to maximum by the end of the first term. The transformed graph will have general equation $y = sig(at + b)$, where a and b are constants which can be varied to adjust the position of the varying region.

If t is measured in teaching days elapsed since the start of the year, it would be good if the point at t=0 corresponded to the -8 on the graph, staying low for the first few weeks until reaching 0.5 at

about t=20. Then it would be close to 1 after 6 weeks of school, which is desired. This corresponds to $b = 8$ and $a = 0.4$. These constants must be tested and improved.

As for the value itself, an initial value could be that every 1% off from the desired number of hours at this point in the year, an extra point is added.

3.4.2.2.2 Constraint 5 – Variety of subjects

Examples of undesirable situations are as follows:

- Having two lessons in one day with the same class at different times
- A given class not having any lessons for a period of time

Out of these, the key factor to minimise is the space between each lesson, as the overall maximum amount of each subject will be handled by constraint 3. Minimising the space between each lesson while also minimising the occurrences on the same day will lead to a well balanced timetable.

This could be calculated as follows:

- For each class:
    - Add some constant k for every occurrence on the same day above 1
        - This can be represented by $max\ \{k(n-1), 0\}$, where k is some constant and n is the number of occurrences on the day
        - A linear model would be appropriate to make it unlikely that there are many lessons on the same day. However, this could potentially leave scenarios where the same subject is featured three times on the same day so the model should be reviewed in testing.
    - Compute the time between this lesson and the next lesson in days. Time of day does not matter. Add on a value that scales exponentially with the time
        - This can be represented by ka^t, where k and a are constants
        - An exponential has been used as this constraint should be insignificant for small t however scale very quickly if there has been a long period of time without any lesson as this would lead to students forgetting information.

3.4.2.2.3 Constraint 7 – Gaps

The total weight will be made up by summing the score from each user in a given day.

This function f(x) must output:

- 0 at x=5
- A large value at x=10,15
- A medium/low value at x=20,25, with f(25) < f(20)
- 0 for x >= 30

NOTE: Time is discrete in multiples of 5 minutes, i.e., $x = 5m, m \in N$

To achieve this 0 cutoff for x >= 30, a max function could be used to change any negative values to 0. The function could also include checking constraint **4** (consecutive lessons), which can replace the separate function defined there.

While exploring possible functions, it became clear that as the values are discrete and there are very few possibilities, there is no need for a single formula to underpin this constraint. The values could simply be manually configured for each possibility.

3.4.3 A Complete Solution

This algorithm forms a complete solution to the problem because it solves the timetabling problem entirely, taking in a list of students and groups and outputting the timetable.

NOTE: This does not form a complete solution to the whole problem, as it is not the only algorithm present. Other algorithms, such as to implement the user interface, will not be detailed here as many specifics regarding the implementation remain unknown. Designing these algorithms in this section rather than in development would not be a valuable use of time.

3.5 Designing the Database Structure

3.5.1 Data to be Stored

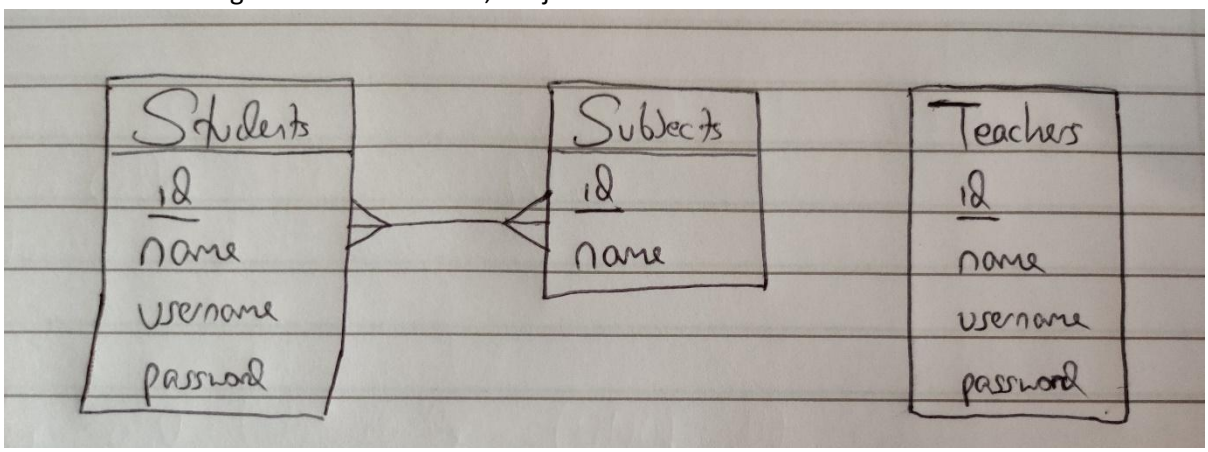
The database will need to store:

- The students
    - Name, login details etc.
    - Which subjects they've chosen
    - Which classes they've been allocated to
- The teachers
    - Name, login details etc.
    - Their classes
- The current timetable
    - Searchable by teacher and by student
- Admin login details

As this is a lot of information, a relational database will be required to store this data in a feasible way.

3.5.2 Initial Draft

An initial draft using a table for Students, Subjects and Teachers is as follows

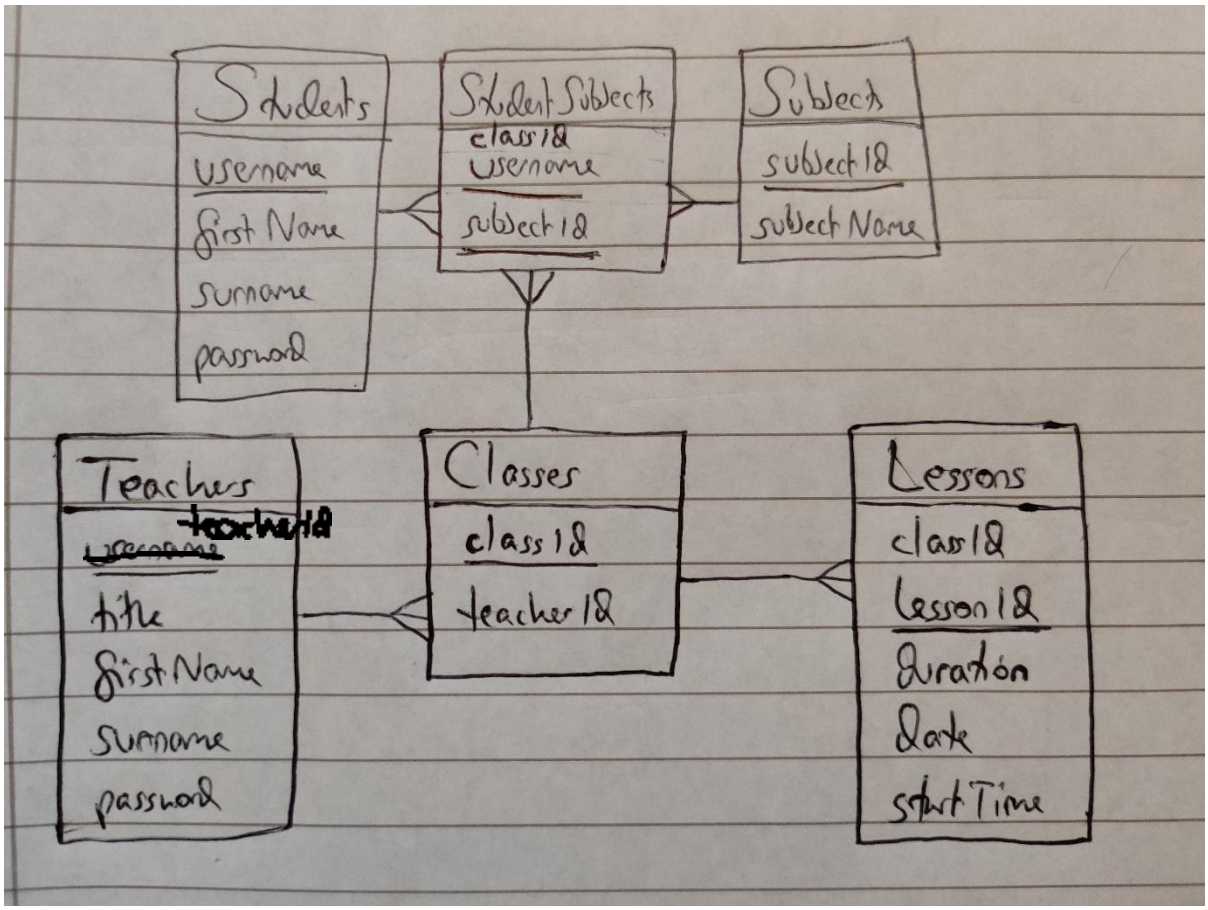


When attempting to join the Teachers table with the others, it did not work. There are other issues however with this data structure that prevent it from being in 3NF, such as:

- The ‘name’ field is not atomic – it must be split up into first name and surname
- There is a many-to-many relationship. This should be fixed by using a linking table

3.5.3 Second Draft

Having fixed these issues, the new database structure is:



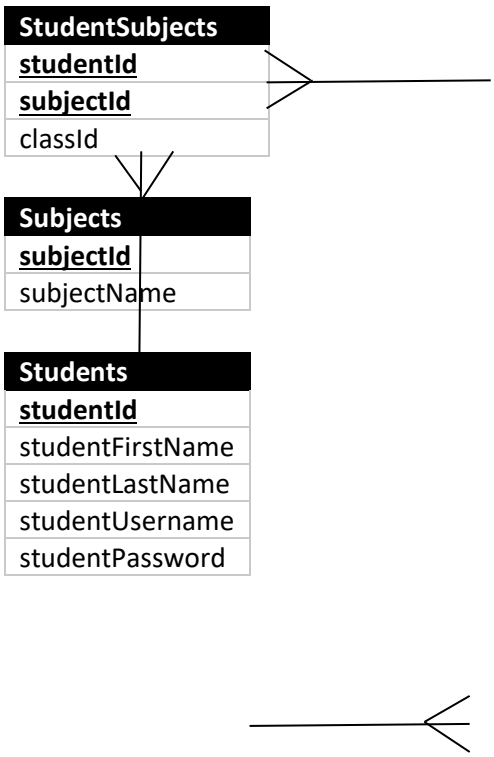
This is better, however does not yet include all necessary fields.

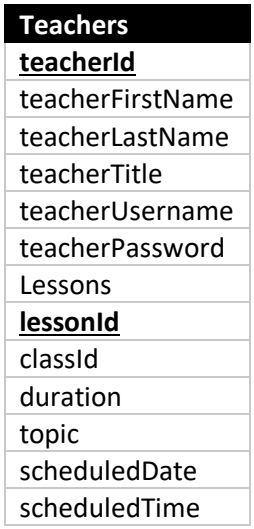
[A page break has been inserted to ensure the next page displays properly]

3.5.4 The Database Structure

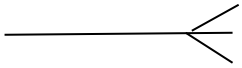
*[NOTE: As with many parts of the project, the database structure is changed throughout. The next iteration can be found at **4.1.3.3**]*

Adding in all necessary fields gives the following database structure





classId
teacherId



This database structure is better because it stores all necessary data (as is known at this point in the project) and satisfies 3NF.

3.6 Key Classes

3.6.1 Draft 1

3.6.1.1 Timetabler Class

[NOTE: The following was written before it was decided that a genetic algorithm would be used]

Multiple different classes are needed to store different pieces of data relevant to the code. Classes are the best way to store this information as it allows binding attributes and methods to the class.

The class to encapsulate all attributes will be known as `Timetabler`. A class is appropriate to hold all attributes as it can easily be passed between methods in a way that avoids global variables. Global variables should be avoided as they are confusing and more difficult to follow in the code.

This class will have the following attributes to store information:

- `classes` [`Classes`]
- `blocks` [`Blocks`]
- `timetable` [`LiveTimetable`]

The class will also have methods to update this information:

- `update()` – Updates the timetable for the next two weeks, using all available data to find the best solution. Timetable for the next two weeks remains unchanged. This should be called daily.
    - A single update function keeps the program simple and helps to ensure it is all kept up-to-date
    - This function will not assume that it has been called on a regular basis, and should continually check to make sure that nothing has been missed.
- `loop(interval: float)` – A blocking call that repeatedly calls `update()` every `interval` seconds
    - This makes it very easy to start a program to keep the timetable up-to-date
- `setup(subjects)` – Allocates classes given a list of student subject choices

`Classes` will be used to represent the various classes and which students fall within the classes. It will be of the format `{subject: {id: [student]}}`, of type `{str: {str: [str]}}`

- `to_dict()` – Outputs a `dict` representing the contents of the class

`Blocks` will provide a quick way for the timetabler to recognise which classes conflict with each other and which do not. It will be stored as a `dict` in the form `{block: {'classes': [id], 'conflicts': {block: conflicts}}}` and of the type `{str: {str: [str], str: {str: int}}`

- `to_dict()` – Outputs a `dict` representing the contents of the class

`LiveTimetable` will represent the current timetable for the next two weeks, and possibly more (note that anything beyond two weeks from the last `update()` call is subject to change). It will be stored as a list of classes of the form: `[{'start': time, 'end': time, 'class': id}]` and of the type `[{str: float, str: float, str: str}]`, where time is in UNIX. A UNIX timestamp allows easy computation of the time difference between different lessons and helps to avoid issues with timezones by using a single number.

- `to_dict()` – Outputs a `dict` representing the contents of the class

3.6.1.2 This has since been changed

None of the above is present in the final solution.

This structure is not ideal because:

- `Classes` and `Blocks` store the same information twice
- The structure is confusing – `LiveTimetable` will need to access `Classes`, which is an attribute of the superclass. This is not good practice.
- It does not have any direct integration with the database or frontend

3.6.2 Final Solution

3.6.2.1 Classes Needed

For a genetic algorithm, it would be appropriate to have two primary classes:

- A class to represent a population of solutions
  - It will include methods relating to the genetic algorithm
- A class representing each individual solution
  - Including methods to operate on a single solution, such as the cost function
  - Many instances of this class will be stored in one instance of the first class

3.6.2.2 Representing a Population

The class Population must have the following methods which will be interacted with publicly:

- A constructor

    - Defines all key attributes and initialises the class
- Start
    - Begins the timetabling process with all initialised attributes

The class will have the following methods which are called as part of the timetabling process:

- Iterate
    - Performs one iteration / generation of the algorithm
- Select best solution
    - Returns the current best solution in the population
- Choose new population
    - Given a list of potential individuals, chooses `self.popsize` of the best individuals
- Evaluate all costs
    - Re-evaluates the cost function for every individual in the population
    - *NOTE: This is not a part of the final solution*
- Choose parents
    - Chooses the best `self.num_parents` individuals from the population
- Generate offspring
    - Generates offspring, using the methods crossover and mutate
- Crossover
    - Returns one offspring, which contains half information from either parent
- Mutate
    - Iterates over all offspring, calling the `mutate()` method bound to each of them

While these methods could be made private, they have been kept public as it allows for them to be used from outside of the class, should the timetabling process want to be modified.

A few of the attributes required are as follows:

- `self.popsize`: size of population
- `self.num_parents`: number of parents to select each iteration
- `self.first_day`: first day to be scheduled
- `self.days`: number of days to schedule for (starting at `first_day`)

Some validation will be needed for the input data:

- Ensure the number of parents is not greater than the size of the population, as this could cause an error
- Ensure the population size is an integer greater than zero. The program will not work if given a negative or non-integer value (so it is important that this issue is identified before timetabling begins)
- The number of days must also be a positive integer, otherwise the program will not work
- The `first_day` must be of an appropriate time, likely a Python `datetime.datetime`. If the type is incorrect, the program will not work.

Other details will be determined during implementation.

3.6.2.3 Representing a Timetable

The class Timetable must have these methods which will need to be public:

- Constructor
- Random
    - Generates a random solution
    - This is separate from the constructor so that it is possible to generate a specific solution instead of just generating random ones
- Cost
    - Retrieves the cost function
    - *NOTE: This was renamed to `get_cost()` during development*
- Mutate
    - Mutates the timetable
- Add
    - Adds the solution to the database

The class also ended up with the following methods:

- get_teacher
    - Gets the teacher that teaches a given lesson, caching the result
- get_gaps
    - Gets a list of the gaps in a day, for use in the cost function
- get_gap_cost
    - Gets the cost value associated with a given gap length

The following attributes are vital:

- `self.unscheduled_lessons`: Stores the list of lessons that are available to be scheduled
    - This should be computed in the constructor if not provided
- `self.lessons`: A `dict` in the form `{day: lesson}`
    - A given timetable is defined by the lessons it contains. These need to be stored in a good format
    - Start time of lessons only includes the start time, not the date, so they need to be stored separately for separate days
    - Lessons cannot stretch over multiple days, so it is not appropriate to store them in the same location
- `first_day`, `time_per_day`, `seconds_per_unit_time`: These all define important facts to relate the abstracted model used for timetabling to the real world

3.6.2.4 Other Classes

Other classes will be used and interacted with as part of the project, however none are as significant as the classes detailed above.

For example, classes are used to define the database structure.

NOTE: The class `PotentiallyScheduledLesson` was added during development as a way to store abstracted start times.

3.7 Validation required

There are two main types of validation:

- User input should be validated to prevent security issues
- Data into function calls should be validated to aid debugging and prevent issues from affecting more parts of the solution. This is defensive programming.

While some validation has been detailed in this section, there is much more validation detailed throughout the project. As much of the project is still undecided, it does not make sense to decide all validation at this stage. Much of the validation will be decided during the development phase. This is detailed where appropriate.

Examples of validation which was decided during the development phase are as follows:

- A lesson's topic field is restricted to a maximum length of 128 characters so that it fits on the interface and is not too large to be stored.
- Lesson durations are restricted to multiples of 5 minutes as this is the smallest increment that the timetabling algorithm will allow.
- User credentials are always verified on all restricted pages.

3.8 Test data

3.8.1 The User Interface

The User Interface does not require specific test data as such, as many of the interactions should be tested simply by navigating the interface. Identifying specific test data is much more relevant for the two algorithms: Timetabling and Class Allocation

3.8.2 Class Allocation

3.8.2.1 This is no longer relevant

The following test data was produced before it was decided that the class allocation algorithm would not be included in the final product. The data has been included here, however none of the following content is present in the final solution.

3.8.2.2 Introduction

In all of the following examples, a table is used to represent the key-value structure sent in to the class allocation algorithm corresponding to the students' subject choices.

3.8.2.3 Simple Cases

This is the simplest possible example and should be used first to check that the algorithm works at all.

Mike	Biology

Expected output:

Block 1	Biology 1

This test will check that the algorithm is able to spread classes over the different main blocks, even when there are no conflicts between them

Mike	Biology
Sam	Geography
Luke	Computer Science
Charlie	Maths

Expected output (in any order):

Block 1	Biology 1
Block 2	Geography 1
Block 3	Computer Science 1
Block 4	Maths 1

The next test will check the algorithm is able to combine together a few subjects into a single block given there are no conflicts, and spread them evenly across all blocks

3.8.2.4 Representing Input

For the purposes of representing the input here, an abstracted format will be most effective as it hides unnecessary detail that will not affect the result of the algorithm. This saves space and helps focusing on the important details of the test data.

Students will simply be represented by numbers, representing how many students have chosen that subject. If a student picks more than one subject (all students will choose 3 or 4 subjects), they will be represented with a letter followed by a number corresponding to the number of students who also picked that subject. Subjects can be represented by letters as these are also unimportant in the context of the problem

For example, the following:

A	13[A1B4D1]
B	4[A1D1]
C	16[B4C2]
D	12[C2A1]

Could be represented as:

Mike	Art, Biology, DT
[4 people]	Art, Chemistry
Dylan	Art, Biology
[7 people]	Art
[2 people]	Biology
[2 people]	Chemistry, DT
[10 people]	Chemistry

[9 people]	DT

(Even this format was simplified because it would take up a significant amount of space to represent the full list of people)

The first format makes it much easier to focus on the conflicts which is the main challenge for the algorithm.

Abstraction has been used to:

- Hide the names of people
- Hide the names of the subjects
- Emphasise the conflicts between subjects
- Organise the list by subject as it is more intuitive

3.8.2.5 Handling Conflicts

The next test will assess the algorithm's ability to handle a single conflict

A	1
B	1[A1]
C	1
D	1[A1]

Expected output (in any order):

1	A
2	B
3	C
4	D

However, even if the algorithm doesn't handle conflicts, it should output this result given it is managing spread across blocks. The following test will ensure that managing spread is not enough to guard against conflicts

A	10[B1]
B	2[A1B1]
C	2[A1B1]
D	2[A1B1]
E	10[A1]

Expected output (in any order):

1	B
2	C
3	D
4	EA

Clearly this output does not manage spread efficiently, however it is the only valid solution that has no conflicts

This is very similar to the problem above, however there is another conflict which will make it impossible to allocate into 4 blocks. As the classes are small, this should form an overflow block

A	1[B1]
B	2[A1B1]
C	2[A1B1]
D	2[A1B1]
E	1[A1]
F	2[A1B1]

Note that this input represents the following:

Alice	EBCDF
Bob	ABCDF

This example is clearly unrealistic as it only includes two students who each pick 5 subjects, however it is a very simple case to test the algorithm's ability to handle overflow blocks.

Expected output (in any order):

M1	B
M2	C
M3	D
M4	EA
O1	F

3.8.2.6 Other Tests

The following tests, and others, would have been included if the class allocation algorithm was due to be present in the final solution. There is no value in creating the test data for these now as it will have no relevance to the final solution.

- Testing spread
- Testing with more than four subjects
- Dividing students between classes (if more students want to do a subject than the maximum class size)

3.8.3 The Timetabling Algorithm

The requirements for the timetabling algorithm are detailed in **3.4.2.1**. These tests have been designed to test if the algorithm fulfils these requirements.

For the purposes of this test data, it can be assumed that the algorithm is timetabling for one day at a time, given a plentiful list of lessons to be scheduled.

3.8.3.1 Base Case

The first test should always consist of the simplest possible scenario. This means that all errors and critical issues can be fixed before testing under more complex circumstances.

The base case is simply an empty school – no lessons to be scheduled.

The expected output is that no lessons will be scheduled.

3.8.3.2 A Simple Case

The next most simple scenario will involve a single lesson. This lesson will belong to a single class, which is taught by a single student and a single teacher.

The expected result is that it will be timetabled at any point in the day. While there is a requirement that should make it more likely to be scheduled earlier on, this is not required for the test to pass.

3.8.3.3 Handing Simple Clashes

Next, it should be tested that the algorithm is able to handle simple clashes. The simplest possible way to test if clashes occur is to schedule two lessons. These lessons will be from the same class (therefore same student and teacher) so that they clash.

The expected output is that the two lessons will be scheduled at a time where they do not clash.

However, there is a high probability that this test will simply pass by random chance. Therefore, it must pass on **10** consecutive attempts to be considered passed.

3.8.3.4 A Full School

3.8.3.4.1 Introduction

Once the simple scenarios have been addressed, the testing process could either:

- Include more specific tests that each test specific parts of the functionality (unit tests)
- Progress to one overall test that will test all features at once (black-box testing)

A black-box test will be required at some point as it tests how the algorithm performs as a whole. Ideally, a combination of both would be used to thoroughly identify all issues present, however this is more time consuming and does not fall within the time constraints of the project.

It was not feasible to obtain data from the school during the project. Producing test data that is at the scale required for a full school environment is time consuming. Post-development, some real-world test data will have to be obtained to fully evaluate the efficacy of the algorithm.

3.8.3.4.2 Shortening the School Day

As it is difficult to create enough data to test for a full school day, it should be possible to evaluate the performance of the algorithm on small test cases and then extrapolate these results to a full school environment. A way in which this could be achieved is by making the school day shorter. By making the school day unrealistically short, fewer lessons can be scheduled before the algorithm reaches a difficult situation that is comparable to the complexity of a full school day.

NOTE: This was eventually changed as it turned out there was sufficient test data to test a full school day

3.8.3.4.3 Success Criteria

The school day can be shortened to 4 hours long. Note that this does make timetabling more difficult as lessons cannot span multiple days (at the boundary between days, lessons cannot be timetabled

as usual). If this was any shorter, the impact from the day boundaries would become too noticeable. A longer day requires more lessons to be scheduled to properly test the capability of the algorithm.

The success criteria states that **44** time units of lessons should be scheduled in a **96** hour day.

Based on the estimation above, an average of at least 1.7 hours of lessons a day should be scheduled per student. This is slightly lower than the estimate based on an actual school due to the added complexity created by the shorter school day. Minimum lesson time per teacher is not a metric that is indicative of the performance of the school so will not be measured.

More precisely, out of ten attempts, the average student lesson time must be at least **20** time units. The day is **48** time units long.

3.8.3.4.4 The Test Data

There will be 11 different groups, 10 of which have at least one lesson to be scheduled. There will be 30 students, where each student does exactly 3 subjects. There will be 8 teachers – two teachers teach multiple classes. Every attempt will be made to make this representative of a typical school; however this cannot be done perfectly without real-world data from a school.

3.8.3.4.4.1 Groups / Teachers

Group ID	Group Name	Subject	Teacher
1	Ma1	Maths	teacher
2	Ma2	Maths	teacher
3	N/A	N/A	[none]
4	Ph1	Physics	alberteinstein
5	Ph2	Physics	mariecurie
6	Econ1	Economics	adamsmith
7	Bio1	Biology	charlesdarwin
8	Bio2	Biology	charlesdarwin
9	Chem1	Chemistry	mendeleev
10	CS1	Computer Science	johnvonneumann
11	Geog1	Geography	christophercolumbus

Subject names have been included to aid production of a realistic timetable. They do not affect timetabling, and will likely not be present in the database.

3.8.3.4.4.2 Students

Subject combinations were purposefully picked to be relatively common subject combinations (but were not exclusively common combinations). Particularly common combinations may have been chosen multiple times by students.

Student	Group 1	Group 2	Group 3
1	Chem1	Ma2	Ph2
2	Bio2	Chem1	Ph2
3	Econ1	CS1	Ph2
4	Ma2	Chem1	Bio2
5	Ph1	CS1	Geog1
6	Ma2	Ph2	Chem1
7	Bio1	Geog1	Ma1
8	Econ1	Geog1	Ma1
9	Bio2	Chem1	Ph2
10	Ma1	Ph1	Bio1
11	Bio1	Geog1	Ma1
12	Bio2	Chem1	Ph2
13	Bio2	Chem1	Ma2
14	Ma2	Ph2	Chem1
15	Ma1	Ph1	CS1
16	CS1	Ma1	Econ1
17	CS1	Chem1	Ma2
18	Geog1	Econ1	Bio1
19	Ma1	Ph1	CS1
20	Ph1	Bio1	Geog1
21	Econ1	Geog1	Bio1
22	Bio2	Chem1	Ma2
23	Ma1	Ph1	CS1
24	Ma1	Ph1	Bio1
25	Econ1	Ma2	Ph2
26	Ma2	Chem1	Bio2
27	Bio2	Chem1	Ma2
28	Ph1	CS1	Geog1
29	Ph1	Bio1	Geog1
30	Econ1	Ma2	Ph2

Note that these students have been allocated into groups by observation, which will not produce results as good as a proper class allocation algorithm would. This will need to be factored in when evaluating the performance of the algorithm.

Group	Students
Ma1	9
Ma2	10

Ph1	9
Ph2	8
Econ1	7
Bio1	8
Bio2	8
Chem1	11
CS1	8
Geog1	9

NOTE: It was later determined that this test data was suboptimal, as each student should be a part of two classes (one for each teacher) rather than just one per subject. This is because in a sixth form, students typically have six teachers rather than just three. Re-creating the test data and running the tests again would not be feasible.

3.8.3.4.4.3 Lessons

Lessons will be generated randomly according to a uniform distribution. The details surrounding its implementation will be addressed when necessary.

3.8.4 Post-Development Testing

Post-development testing is outlined thoroughly in the success criteria available in **2.7.3**.

This test data will suffice to evaluate the performance of the algorithm either during development or post-development. There is no additional test data of this scale required specifically for post-development testing.

Examples of post-development testing include:

- The usability of the interface is tested through a specific test that users must complete within a time limit.
- The performance of the algorithm will be tested using the same test data detailed here.
- Function and robustness will be tested through using the product and attempting to break it.

4 Development

4.1 User Interface

The various user interfaces have already been designed (see **3.2**) and need to be implemented. As in the requirements specification, the UI will be web-based as this ensures convenient access for everyone.

4.1.1 Choosing a Web Framework

A web framework will significantly speed up the development process for the website.

The most common Python web frameworks are Django and Flask.

Django	Flask
Includes many pre-built solutions	Lightweight and modular
Built-in support for databases, which are linked to classes	Flexible – allows any database to be used. Many third-party libraries are available
Better for multi-page applications	Better for single-page applications
Developers do not have full control	Full control for developers
Supports dynamic HTML pages	Does not support dynamic HTML pages
No virtual debugging	Built-in debugger and unit testing
Faster to develop in	More flexible to develop in
Easier to learn	More difficult to learn
Open source	Open source

Overall, Django is easier to learn, faster to develop in and contains many built-in solutions, whereas Flask is lightweight, allowing the developer more control. As the proposed solution does not require unusual features and is somewhat time pressured, Django would be the most sensible choice.

4.1.2 Boilerplate Code

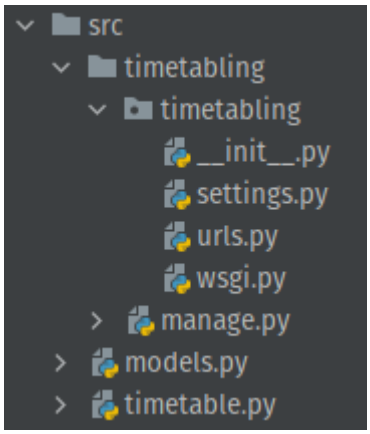
The boilerplate code provides a starting point for further development on the project. This includes all key elements that apply to every Django project.

4.1.2.1 Generating the Basic Code

4.1.2.1.1 Creating the Project

The command `django-admin startproject timetabling` generates this code, including a few basic files with some default settings and other starting code.

The current file structure is as follows:

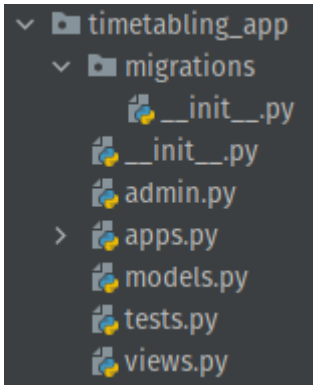


*NOTE: This structure has changed significantly throughout the project. The final structure is available in **6.2***

4.1.2.1.2 Creating the App

Next, an app was created within this project by using the command `python manage.py startapp timetabling_app`. Apps are isolated sections of a Django project, which are often used to separate different areas of the website (e.g. a blog page, a status page and a store). For this project, only one app will be required.

This created the new folder `timetabling_app`:



4.1.2.1.3 Choosing the Webpages

It is critical to choose which webpages are required for the project before beginning to implement them.

Initially, the following pages will be used:

- Homepage
- Student:
    - Student login
    - Student timetable
- Teacher:
    - Teacher login
    - Teacher timetable
    - Teacher scheduler

- Admin:
    - Admin login
    - Admin timetable

Other pages may be deemed necessary for the project as it continues development

These could then be put into Django, using placeholder functions to display the webpages:

```
urlpatterns = [
    path('', views.home, name='timetable-home'),

    path('student/login/', views.student_login, name='timetable-student-login'),
    path('student/', views.student_timetable, name='timetable-student'),

    path('teacher/login/', views.teacher_login, name='timetable-teacher-login'),
    path('teacher/', views.teacher_timetable, name='timetable-teacher'),
    path('teacher/schedule', views.teacher_scheduler, name='timetable-teacher-schedule'),

    path('admin/login/', views.admin_login, name='timetable-admin-login'),
    path('admin/', views.admin_timetable, name='timetable-admin'),
]
```

```
def home(request):
    return HttpResponse("Here will be a basic homepage giving links to the other sites")


def student_login(request):
    return HttpResponse("This will process student logins")


def student_timetable(request):
    return HttpResponse("This will display student timetables")


def teacher_login(request):
    return HttpResponse("This will process teacher logins")


def teacher_timetable(request):
    return HttpResponse("This will display teacher timetables")


def teacher_scheduler(request):
    return HttpResponse("This will allow teachers to schedule lessons")


def admin_login(request):
    return HttpResponse("This will process admin logins")


def admin_timetable(request):
    return HttpResponse("This will display the overall timetable")
```

Placeholder functions were used initially as they allow for the overall structure of the code to be tested before proceeding.

4.1.2.2 Review

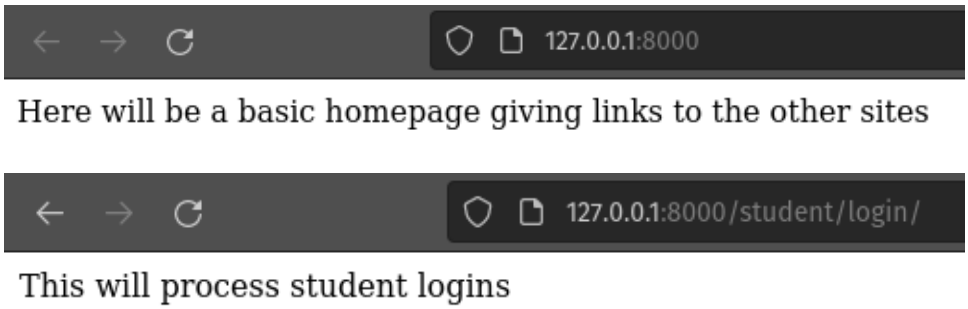
While this code is fully functional, the naming scheme is very confusing.

- The name of the project will be renamed from `timetabling` to `django_project`
- The name of the app will be renamed from `timetabling_app` to `timetable`

These names are much clearer and easier to work with.

4.1.2.3 Testing

The project can be started by simply running `python manage.py runserver`. The server is then running on `localhost:8000`



The website worked as intended without any further changes.

4.1.3 Implementing the Database

4.1.3.1 Choosing a Database

Django supports many databases, including SQLite (default), PostgreSQL and others.

For this project, the database must be:

- Fast
- Reliable

It does not have to be particularly:

- Scalable (not much data is being stored)

These criteria are satisfied by most databases, including many offered by Django. The default option, SQLite, will suffice for the needs of the project.

The database structure was first designed in **3.5**. Implementing this database structure was trivial when using Django's database support. Using Django's database support has a number of advantages, including:

- It can be interacted with through natural Python code rather than SQL
- Django will automatically deal with current security threats such as SQL injection, and will be updated to deal with any potential new security threats in future

4.1.3.2 Adding the Models

In Django, a table is represented by a class which inherits from `models.Model`, and is called a model. Records are represented by instances of these models (objects), and fields are represented with attributes of these objects, which can be queried and modified as with other Python attributes.

`models.py`

```
from django.db import models

class Students(models.Model):
    first_name = models.CharField(max_length=16)
    last_name = models.CharField(max_length=16)
```

```
    username = models.CharField(max_length=32)
    password_hash = models.CharField(max_length=32)

class Subjects(models.Model):
    name = models.CharField(max_length=16)

class StudentSubjects(models.Model):
    student = models.ForeignKey(
        'Students',
        on_delete=models.CASCADE,
    )
    subject = models.ForeignKey(
        'Subjects',
        on_delete=models.CASCADE
    )
    group = models.ForeignKey(
        'Groups',
        on_delete=models.SET_NULL
    )

class Groups(models.Model):
    teacher = models.ForeignKey(
        'Teachers',
        on_delete=models.SET_NULL
    )

class Teachers(models.Model):
    first_name = models.CharField(max_length=16)
    last_name = models.CharField(max_length=16)
    title = models.CharField(max_length=8)
    username = models.CharField(max_length=32)
    password_hash = models.CharField(max_length=32)

class Lessons(models.Model):
    group = models.ForeignKey(
        'Groups',
        on_delete=models.CASCADE
    )
    duration = models.DurationField()
    topic = models.CharField(max_length=128)
    scheduled_datetime = models.DateTimeField()
```

To enforce referential integrity, each foreign key has a field `on_delete` – this informs the database what to do when a record is deleted. The two used in the code were:

- `models.CASCADE`: Delete the record
    - This was used when it made no sense to keep the record without the other record.
- `models.SET_NULL`: Set the value for the foreign key to null
    - In these situations, deletion should not be done. However, if it is done for whatever reason, it would make sense to keep the record in case there was useful information stored in it.

Length limits were used where appropriate to prevent long data from appearing in the database, which may then not display properly in the user interface.

4.1.3.3 Changes to the Database

*[Previous Iteration: **3.5.4** Next Iteration: **4.1.6.1.3**]*

Some changes were made to the database, such as:

- Django does not support composite primary keys, so the `StudentSubjects` table will have a different auto-generated primary key
- Many fields were renamed
- 'Classes' will now be described as 'Groups' to avoid confusion with Python classes.
    - In this project, they may be described as either 'classes' or 'groups'. Both of these terms have the same meaning.

A new structure diagram was not produced for this iteration of the database structure.

4.1.4 Implementing the Templates

While the website will a strong programmed component, many of the webpages have many static elements that can be developed in HTML and CSS first. These are registered in Django as a 'template' - they contain static code as well as sections that can be dynamically modified before sending to the user. These are a combination of standard HTML elements as well as a custom Django language that resembles Python, allowing for selection and iteration.

The pages that will be required were determined in **4.1.2.1.3**.

Fortunately, as many of these are very similar, they can be combined into just a few main templates:

- Login (student, teacher, admin)
- Timetable (student, teacher)
- Scheduler (teacher)
- Admin (admin timetable)

Django also allows modular design of templates, so a fifth base template could be used which contains all the essential code that features on all webpages. This would be very beneficial so if it needs changing on one page it can be changed on all pages at once. Each of these templates would then inherit from the base template.

Other pages (e.g. student timetable, teacher timetable) can inherit from these templates as appropriate, with minor modifications.

4.1.4.1 Base Template

This template should contain all code that is present in every webpage.

Currently, this only includes the basic HTML code required.

```
{% extends "timetable/base.html" %}
<!DOCTYPE html>
```

```
<html>

<head>
    <title>Timetabling</title>
</head>

<body>
    {% block content %}
    {% endblock %}
</body>

</html>
```

Using a block for the content makes it very clear where the code should go. In other files, HTML can be placed inside of the content block and it will appear in this location. For example:

```
{% extends "timetable/base.html" %}
{% block content %}
    CODE HERE
{% endblock %}
```

4.1.4.2 Home Page

The home page was implemented as follows:

```
{% extends "timetable/base.html" %}
{% block content %}
    <h1>Timetabling</h1>
    <p>Welcome to the timetabling website</p>
    <p>Are you a:</p>
    <a href="/student">Student</a>
    <a href="/teacher">Teacher</a>
{% endblock %}
```

4.1.4.3 Login Page

The login page can be very simplistic. It must take in a username and password and send it to the backend for processing. Then, the user is either redirected to the relevant site or an error message is displayed.

```
{% extends "timetable/base.html" %}
{% block content %}
    <h1>Login as {{user}}</h1>
    <form>
        <label>
            <input placeholder="username">
        </label>
        <label>
            <input type="password" placeholder="password">
        </label>
        <input type="submit">
    </form>
{% endblock %}
```

If passed with the variable `user` which takes the value of '`Student`', '`Teacher`' or '`Admin`', this HTML will form a basic login page.

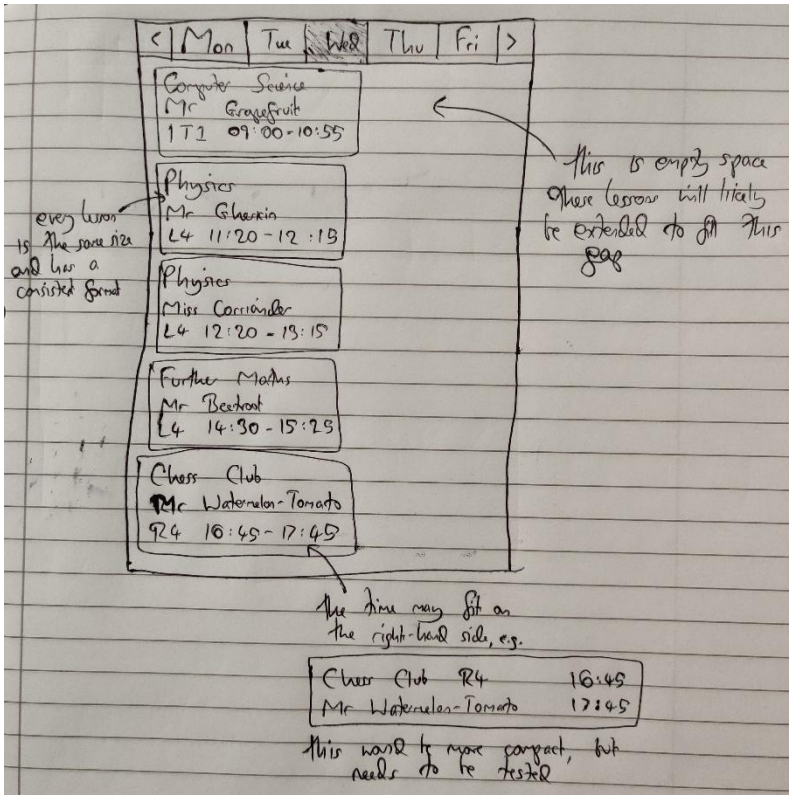
4.1.4.4 Timetable

The design for this page can be found in **3.2.2.3** and **3.2.3.2**. The template `timetable.html` should only include features that are the same in both designs.

Unfortunately, upon doing some basic testing on a mobile device, it was clear that it was not feasible to fit the whole day onto the screen with gaps between lessons. The gaps use up too much of the vertical space available such that it is not possible to fit the whole day onto the screen.

Having the whole day on one screen is valuable as it avoids having to scroll unnecessarily and allows users to quickly glance at their timetable. This is more important than using gaps to indicate free time.

The interface will be changed to no longer include gaps to allow the content to fit onscreen.



This does have some advantages. As there is more space to fit information on each lesson, students will not have to click on the lesson to find out any information that couldn't fit. A consistent size for the boxes also simplifies the implementation, allowing for more time to be allocated to other areas of the project.

For the initial implementation, the placeholder data will be used. Then, in **4.1.7**, it can be programmed to gather the relevant data from the database.

First, a block was created for the header which can be stored in a different file. This can then be re-used in this interface as well as the admin interface.

`header.html`

```
{% extends "timetable/timetable.html" %}
{% block header %}
    <div>
        <div><</div>
        <div>Mon</div>
        <div>Tue</div>
        <div>Wed</div>
        <div>Thu</div>
        <div>Fri</div>
        <div>></div>
    </div>
{% endblock %}
```

Next, the elements corresponding to the example lessons can be created. A class was given to each timetabled lesson. Other classes will be assigned when doing the CSS in **4.1.5**.

Note that the final version will not include this much repeating code – this is only to test the functionality and appearance of the interface until it is programmed in Django.

`timetable.html`

```
{% extends "timetable/base.html" %}
{% block content %}
    {% block header %}
    {% endblock %}

    <div>
        <div class="timetabled-lesson">
            <p>Computer Science</p>
            <p>Mr Grapefruit</p>
            <p>IT1</p>
            <p>09:00 - 10:55</p>
        </div>
        <div class="timetabled-lesson">
            <p>Physics</p>
            <p>Mr Gherkin</p>
            <p>IT1</p>
            <p>11:20 - 12:15</p>
        </div>
        <div class="timetabled-lesson">
            <p>Physics</p>
            <p>Miss Coriander</p>
            <p>IT1</p>
            <p>12:20 - 13:15</p>
        </div>
        <div class="timetabled-lesson">
            <p>Further Maths</p>
            <p>Mr Beetroot</p>
```

```
            <p>IT1</p>
            <p>14:30 - 15:25</p>
        </div>
        <div class="timetabled-lesson">
            <p>Chess Club</p>
            <p>Mr Watermelon-Tomato</p>
            <p>R4</p>
            <p>16:45 - 17:45</p>
        </div>
    </div>
{% endblock %}
```

4.1.4.5 Scheduler

This section of the site will need to be two pages:

- A form to schedule the lessons
- A list of scheduled lessons

4.1.4.5.1 Scheduling Lessons

This was designed in **3.2.3.3.**

Implementation of this is very similar to the form created for the login page.

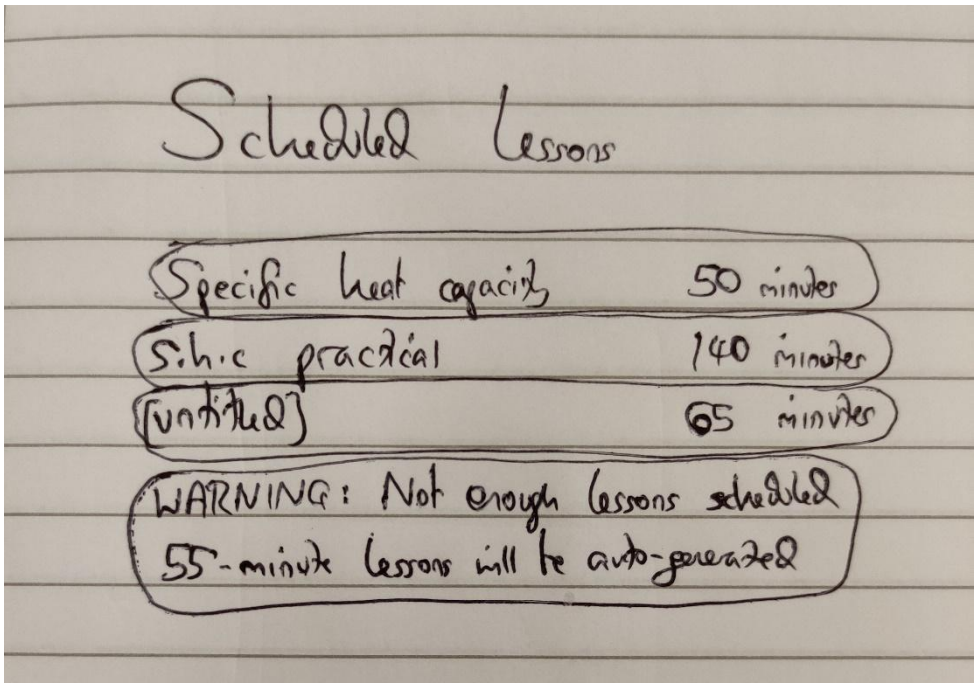
`schedule.html`

```
{% extends "timetable/base.html" %}
{% block content %}
    <h1>Schedule a Lesson</h1>
    <form>
        <label>
            <input placeholder="Topic (optional)">
        </label>
        <label>Duration (minutes)
            <button>55</button>
            <button>85</button>
            <button>115</button>
            <button>135</button>
            <input type="number">
        </label>
        <input type="submit">
    </form>
{% endblock %}
```

IDs and classes have not yet been assigned – these will be assigned while creating the style for the website.

4.1.4.5.2 List of Scheduled Lessons

The design for the interface will be the following:



This interface is great because it shows only the required information in a compact way. If a teacher has many scheduled lessons, lots of them will be able to fit on the screen while a scroll bar can be used to access the rest.

It may also be appropriate to colour in lessons that have already been scheduled so that it is clear which lessons are scheduled and how many are ready to be scheduled.

`scheduled_list.html` [this could have been named scheduled.html, but this could potentially be confused with schedule.html]

```
{% extends "timetable/base.html" %}
{% block content %}
    <h1>Scheduled Lessons</h1>
    <div>
        <p>Specific Heat Capacity</p>
        <p>50 minutes</p>
    </div>
    <div>
        <p>s.h.c. practical</p>
        <p>140 minutes</p>
    </div>
    <div>
        <p>[untitled]</p>
        <p>65 minutes</p>
    </div>
    <div>
        <p>WARNING: Not enough lessons scheduled</p>
        <p>55-minute lessons will be auto-generated</p>
    </div>
{% endblock %}
```

[Again, this repeating code will not be present in the final solution. This is only placeholder data]

4.1.4.6 Admin

The admin interface was designed in **3.2.1.3**.

This interface can make use of the header used for the student/teacher timetable.

First, a new block had to be created to contain all the times along the top.

`horizontal_times.html`

```
{% extends "timetable/admin_timetable.html"}
{% block horizontal_times %}
    <p>9:00</p>
    <p>10:00</p>
    <p>11:00</p>
    <p>12:00</p>
    <p>13:00</p>
    <p>14:00</p>
    <p>15:00</p>
    <p>16:00</p>
    <p>17:00</p>
    <p>18:00</p>
{% endblock %}
```

Then, a few placeholder lessons were included. Lessons are wrapped in `<div>` tags so they are contained and can be rearranged easily.

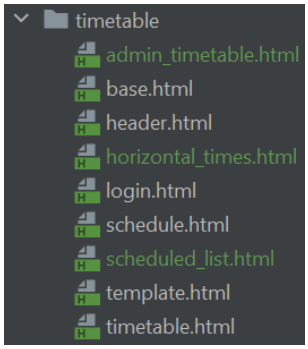
`admin_timetable.html`

```
{% extends "timetable/base.html" %}
{% block content %}
    {% block header %}
    {% endblock %}

    {% block horizontal_times %}
    {% endblock

    <div>
        <div class="admin-timetabled-lesson">
            <p>CompSci</p>
            <p>13A</p>
            <p>SEG</p>
            <p>IT1</p>
        </div>
        <div class="admin-timetabled-lesson">
            <p>FM</p>
            <p>13B</p>
            <p>JAS</p>
            <p>R20</p>
        </div>
        <div class="admin-timetabled-lesson">
            <p>History</p>
            <p>13B</p>
```

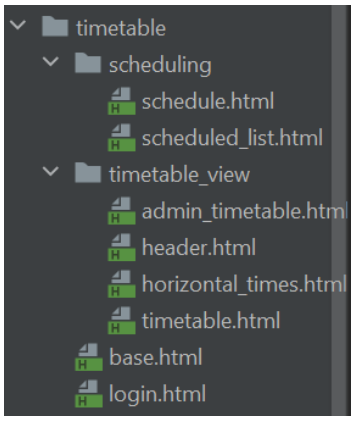
```
            <p>ADF</p>
            <p>R19</p>
        </div>
    </div>
{% endblock %}
```

4.1.4.7 Reviewing the Templates (pre-CSS)

The current directory for the templates is as follows:



This contains too many files which are not clearly organised. To fix this, the file structure has been changed. The new structure is as follows:



The relevant parts of the files including `{% extends … %}` have been renamed.

4.1.4.8 Testing the Templates

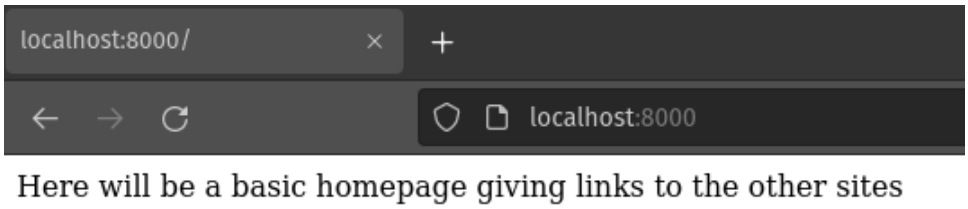
4.1.4.8.1 Starting the server

Starting the Django server worked as expected:

```
josh@pop-desktop:~/Documents/coding/a-level-timetabling/src/django_project$ python3 manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
February 02, 2022 - 19:35:43
Django version 2.2.24, using settings 'django_project.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

4.1.4.8.2 Accessing the Homepage

However, upon accessing the website, it was clear that these new templates were not being used by the website

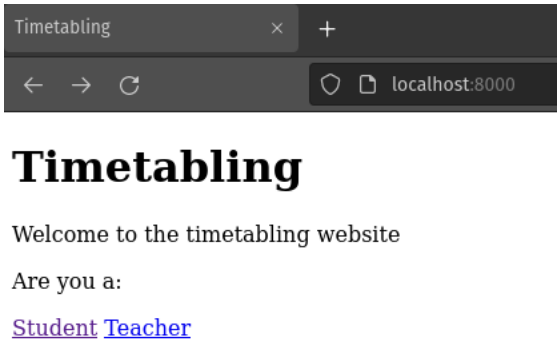


The Django code had not been updated to use the templates

This can be addressed by updating the `views.py` file as shown:

```
from django.shortcuts import render
from django.http import HttpResponse


def home(request):
    return render(request, 'timetable/home.html')


def student_login(request):
    return render(request, 'timetable/login.html', {'user': 'Student'})


def student_timetable(request):
    return render(request, 'timetable/timetable_view/timetable.html')


def teacher_login(request):
    return render(request, 'timetable/login.html')


def teacher_timetable(request):
    return render(request, 'timetable/timetable_view/timetable.html')


def teacher_scheduler(request):
    return render(request, 'timetable/scheduling/schedule.html')


def teacher_scheduled(request):
    return render(request, 'timetable/scheduling/scheduled_list.html')


def admin_login(request):
    return render(request, 'timetable/login.html')


def admin_timetable(request):
    return render(request, 'timetable/timetable_view/admin_timetable.html')
```

This gives the page:



However, ‘`Student`’ and ‘`Teacher`’ should not be next to each other. This can be addressed by creating a bullet point list with `<ul>`.

(extract from) `home.html`

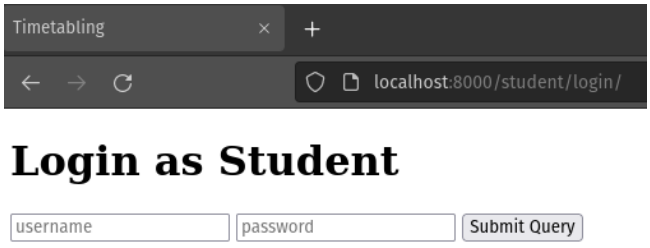
```
<ul>
    <li><a href="/student">Student</a></li>
    <li><a href="/teacher">Teacher</a></li>
</ul>
```

4.1.4.8.3 Login Page

The following behaviour should be noticed on the login pages:

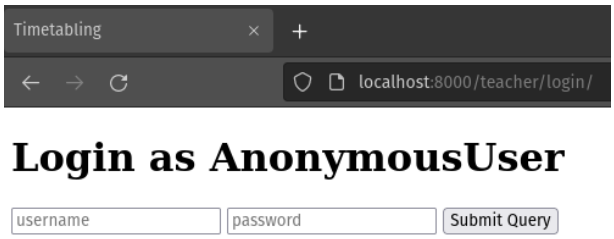
- The elements on the page should be as expected
- The ‘Login as...’ will be followed by either ‘Student’, ‘Teacher’ or ‘Admin’ as appropriate
- The password field should only display dots while typing

The site appeared entirely as expected:



The password field also functioned as expected:

Unfortunately, accessing **/teacher/login** did not function correctly:

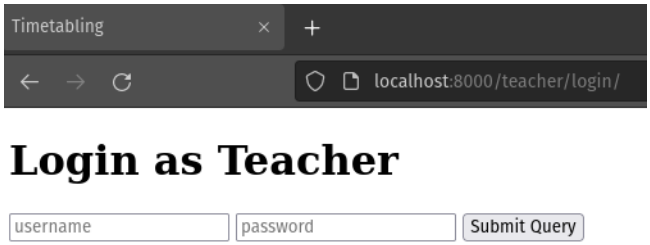


However, this can easily be fixed by updating the context variable sent while rendering the template

```
return render(request, 'timetable/login.html', {'user': 'Teacher'})
```

The same was done for the admin login

After these changes, it functioned entirely as expected:



However, another issue was encountered when attempting to access `/admin/login`. This redirected to the Django admin page instead of the admin timetable.

4.1.4.8.3.1 Reconsidering the Admin Page

Django has a built-in admin page which allows for easy maintenance of the server and direct database access. This page may even offer enough functionality to replace a custom interface.

Django Interface	Custom Implementation
Offers full database access	User interface could be customised further
Tested and maintained by the Django team	Additional features not currently required may be added in future
Ready-to-go	Requires development time to create

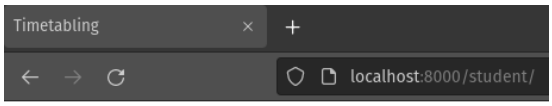
While a custom solution would give more flexibility to add new functionality, this is significantly outweighed by the benefits of Django's admin page.

It is for this reason that a custom implementation of the admin interface will not form a part of the final solution.

The Django interface will still be modified appropriately to contain all important functionality as required by the requirements specification.

4.1.4.8.4 Timetable

Accessing `/student` produced the following:



Computer Science

Mr Grapefruit

IT1

09:00 - 10:55

Physics

Mr Gherkin

IT1

11:20 - 12:15

Physics

Miss Corriander

IT1

12:20 - 13:15

Further Maths

Mr Beetroot

IT1

14:30 - 15:25

Chess Club

Mr Watermelon-Tomato

R4

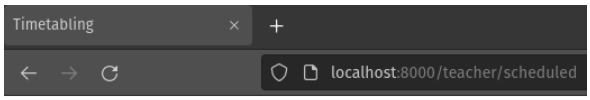
16:45 - 17:45

While this is clearly unsuitable for use, this was the expected output before CSS is done.

Accessing `/teacher` produced exactly the same result, as expected, because the same placeholder data is being used for both pages.

4.1.4.8.5 Scheduling

`/teacher/scheduled`



Scheduled Lessons

Specific Heat Capacity

50 minutes

s.h.c. practical

140 minutes

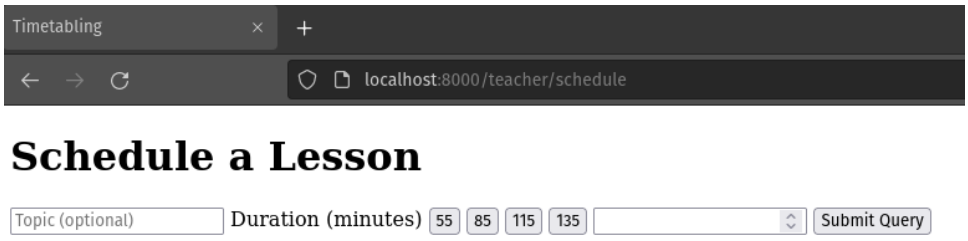
[untitled]

65 minutes

WARNING: Not enough lessons scheduled

55-minute lessons will be auto-generated

`/teacher/schedule`



[NOTE: While it was mentioned earlier that it was easy to confuse 'schedule' and 'scheduled', the teachers are not expected to be modifying these URLs, and they are not used in programming outside of the `urls.py` *file. If this becomes an issue, the URL paths can very easily be changed by just changing the urls.py file.]*

Before CSS, these webpages are exactly as expected.

4.1.4.9 Addressing Security Concerns

The project is open source and stored on GitHub. This means it is important that the secret key is not visible publicly. Unfortunately, this key was in the code on GitHub available publicly. As the website was not publicly accessible, this was not a problem initially. However, this issue needed to be addressed should the webpage become publicly accessible.

As the secret has already been public, it must be regenerated.

```
from django.core.management.utils import get_random_secret_key
get_random_secret_key()
```

This returned a key which will not be put here for security reasons.

Next, a `credentials.json` file was created in the same directory to include information that should not be on GitHub. This had the secret key stored under `django_secret`

```
with open('credentials.json') as credentials_bytes:
    credentials_text = credentials_bytes.read()
    credentials = json.loads(credentials_text)
```

Then, the secret key could be replaced by a reference to this file

```
SECRET_KEY = credentials['django_secret']
```

Finally, `credentials.json` could be added to `.gitignore` to prevent it from being pushed to GitHub

A quick test ensured that the site still worked and that pushing to GitHub no longer revealed the secret.

4.1.5 Style

Of course, these templates are completely unsuitable for use without any style. Without style, the user interface is more difficult to use and does not look professional.

To style Django websites, Bootstrap is widely used. Bootstrap simplifies the process by providing many pre-made classes with accompanying CSS and JavaScript to implement commonly used web structures.

4.1.5.1 Installing Bootstrap

4.1.5.1.1 Installation

The first step was to install Bootstrap. There are two alternative ways of installation:

Downloading the files	Using Bootstrap CDN
Page will load faster	Will automatically update as changes are pushed to Bootstrap
Version can be controlled to ensure it always works	Does not use up as much storage space

The file `style.css` was created and linked to `base.html` with a `<link>` tag. This will be included on every webpage.

Some other changes were made to the file at the same time:

- If the `title` variable is passed in, it will be displayed in the `<title>` of the webpage
- Some basic `<meta>` tags to help formatting and aid consistency across browsers

`base.html`

```
{% load static %}
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, height=device-
height">
    <meta name="keywords" content="timetabling, timetable, scheduler">
    <meta name="author" content="Joshua Humphriss">

    <title>
        {% if title %}{{title}} - {% endif %}Timetabling
    </title>

    <link rel="stylesheet" type="text/css" href="{% static
'timetable/style.css' %}">
</head>

<body>
```

```
    {% block content %}
    {% endblock %}
</body>

</html>
```

4.1.5.1.2 Testing

First, it was important to check that these changes had not stopped the site from working. This was not the case, and the site continued to work as usual.

However, when attempting to use CSS, Bootstrap (and the `style.css` file) were clearly not active.

One issue was that Django had not been properly configured to recognise where this CSS code is. This could be fixed by adding a variable `STATICFILES_DIRS` in `settings.py` as shown

```
STATICFILES_DIRS = (
    '/django_project/timetable/static/',
)
```

This made the `style.css` work but did not facilitate Bootstrap working. This is because `base.html` did not include a reference to the Bootstrap code, so it was not applied on the webpage. This could be easily fixed by adding the following lines to `base.html`

```
<link rel="stylesheet" type="text/css" href="{% static
'timetable/css/bootstrap.css' %}">
<script src="{% static 'timetable/js/bootstrap.js' %}"></script>
```

Now, Bootstrap was active along with `style.css` to allow for easy and flexible styling.

4.1.5.2 Home Page

Only basic styles were needed for the homepage, which could be done entirely with Bootstrap.

```
{% extends "timetable/base.html" %}
{% block content %}
    <div class="container">
        <h1 class="text-center mt-4">Timetabling</h1>
        <p class="text-center mt-3 mb-4">Choose one of the options to
proceed to the appropriate interface</p>
        <div class="text-center">
            <a href="/student"><button class="btn btn-
primary">Student</button></a>
            <a href="/teacher"><button class="btn btn-
primary">Teacher</button></a>
        </div>
    </div>
{% endblock %}
```

The following changes were made:

- Everything was centre-aligned
    - This helps the content be closer to the centre of the screen which is closer to where the user will naturally be looking and is easier to use
- The list was converted to side-by-side buttons, which were made to look more professional
    - This is easier and more intuitive to interact with
- The text was changed to be more informative
- Spacing was added to aid readability

4.1.5.3 Login Page

4.1.5.3.1 Development

The login page was significantly improved, with both aesthetics and QoL improved.

- Everything was centre-aligned
- The input boxes were made wider to fill the relevant screen space. This gives plenty of space for the user's input, without it occupying so much of the screen to be overwhelming.
- A new link was added to quickly switch between teacher and student login. This will be helpful in case the user accidentally clicks on the wrong link.
- The login button had its text changed, and was made green to indicate success
- A new link was added to quickly navigate back to the home page, in case the user gets lost while navigating the website.

```
{% extends "timetable/base.html" %}
{% block content %}
    <div class="text-center login-width m-auto">
        <h1 class="my-4">Login as {{user_type|title}}</h1>
        <form class="text-left">
            <label class="w-100">
                <input class="w-100" placeholder="Username">
            </label>
            <label class="w-100">
                <input class="w-100" type="password"
placeholder="Password">
            </label>
            <a href="/{{user_opposite}}/login">Switch to {{user_opposite}}
login</a><br>
            <input type="submit" class="btn btn-success mt-2"
value="Login">
            <a href="/" class="text-danger">Back to home</a>
        </form>
    </div>
{% endblock %}
```

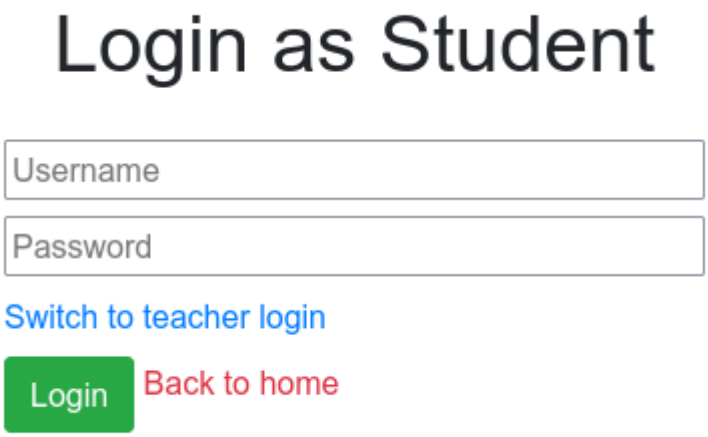
Both the teacher page and the user page are being done through this single template. This is excellent because the styling is consistent on both pages and it means that if they both need to be updated, it can easily be done at the same time while ensuring consistency.

`views.py` was changed to pass in the user as lowercase, while also passing the opposite to allow for easy navigation between the two pages.

```
def student_login(request):
    return render(request, 'timetable/login.html', {'user_type': 'student',
'user_opposite': 'teacher'})

def teacher_login(request):
    return render(request, 'timetable/login.html', {'user_type': 'teacher',
'user_opposite': 'student'})
```

4.1.5.3.2 Testing (on desktop)

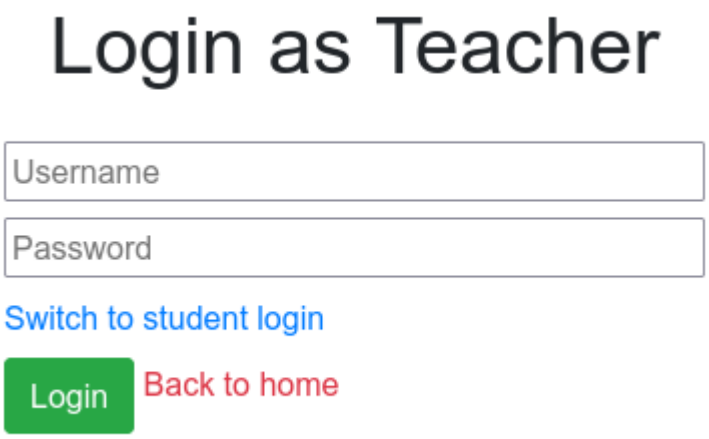
Testing was done throughout the development process for this page. The final result, on desktop, is as follows:



This interface is very intuitive and contains many usability features that help navigation around the site, such as:

- The 'Switch to teacher login' link
- 'Back to home' link
- Colour coding helps users quickly identify the relevant button
- Large text for the heading helps the user ensure they are on the correct page

Clicking 'Switch to teacher login' navigates to `/teacher/login` as expected

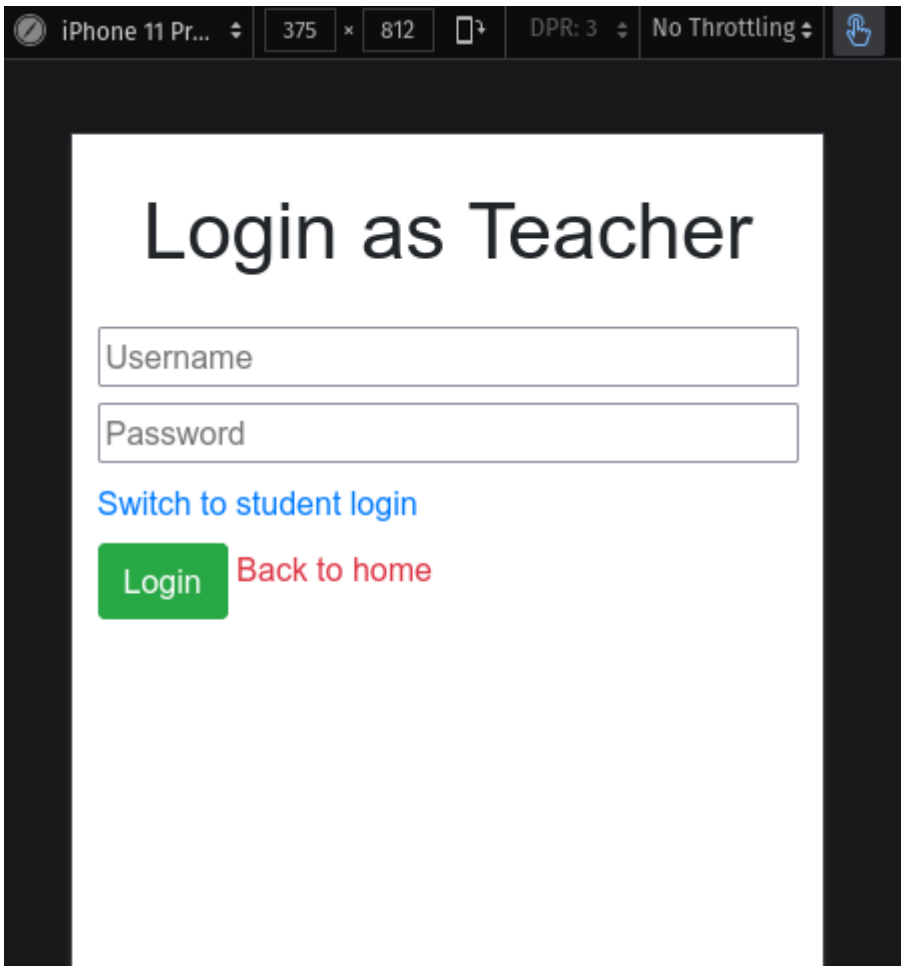


This interface displays entirely as expected

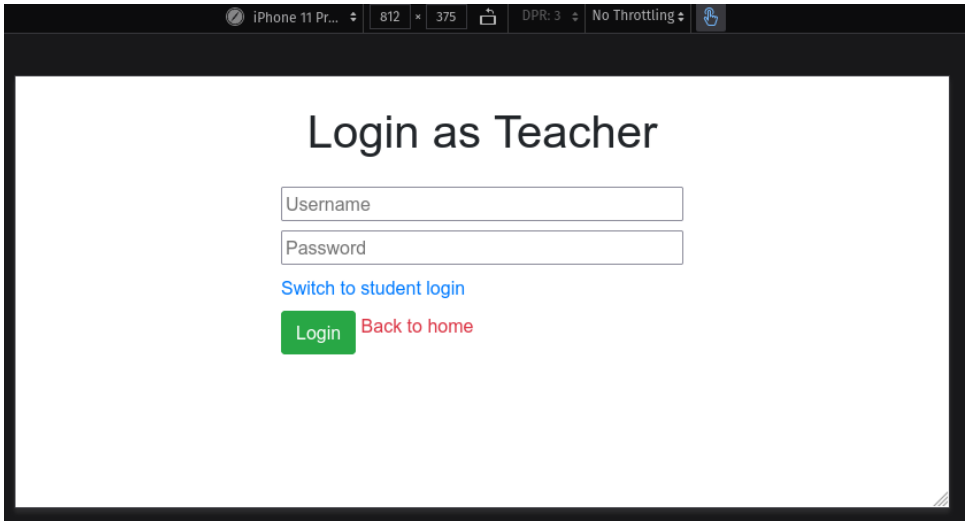
4.1.5.3.3 Testing (on mobile)

While testing on an actual mobile device is not possible when running via localhost, Firefox has built-in options to allow testing on simulated mobile environments.

The webpage displayed as expected on the simulated iPhone 11 Pro environment.



It also worked in landscape mode, however did not look as good. This is not an issue as it would be unusual to be browsing in landscape mode.



Other devices (screen sizes) were also tested, and no issues were found.

4.1.5.4 Timetable

4.1.5.4.1 Development

The timetable interface is the most important interface. It must quickly convey the user's timetable as quickly as possible. A design very close to the intended design was achieved.

- The header was done by using Bootstrap classes such as `nav`, `nav-tabs` and `nav-item`. `nav-justified` means that each element has equal width.
- Lessons were given classes to highlight them with different colours. These colours were carefully chosen such that two similar colours are not adjacent to each other.
    - The `!important` CSS tag had to be added to override the Bootstrap defaults.

Note that:

- The links are still dead links (#).
- There is lots of repeating code.
- The lessons are only placeholder data, and will need to be fetched from the database
    - This data is the same as in the UI design

All of the above are addressed later in the section.

`timetable.html`

```
{% extends "timetable/base.html" %}
{% block content %}
    <ul class="nav nav-tabs nav-justified">
        <li class="nav-item">
            <a href="#" class="nav-link px-0"><</a>
        </li>
        <li class="nav-item">
            <a href="#" class="nav-link px-0">Mon</a>
```

```
        </li>
        <li class="nav-item mx-0">
            <a href="#" class="nav-link px-0">Tue</a>
        </li>
        <li class="nav-item active">
            <a href="#" class="nav-link px-0">Wed</a>
        </li>
        <li class="nav-item">
            <a href="#" class="nav-link px-0">Thu</a>
        </li>
        <li class="nav-item">
            <a href="#" class="nav-link px-0">Fri</a>
        </li>
        <li class="nav-item">
            <a href="#" class="nav-link px-0">></a>
        </li>
    </ul>

    <div>
        <div class="timetabled-lesson subject-1 card">
            <div class="card-body">
                <h5 class="card-title">Computer Science</h5>
                <p class="card-text mb-1">Mr Grapefruit</p>
                <p class="card-text mb-1">IT1</p>
                <p class="card-text">09:00 - 10:55</p>
            </div>
        </div>
        <div class="timetabled-lesson subject-2 card">
            <div class="card-body">
                <h5 class="card-title">Physics</h5>
                <p class="card-text mb-1">Mr Gherkin</p>
                <p class="card-text mb-1">L4</p>
                <p class="card-text">11:20 - 12:15</p>
            </div>
        </div>
        <div class="timetabled-lesson subject-2 card">
            <div class="card-body">
                <h5 class="card-title">Physics</h5>
                <p class="card-text mb-1">Miss Coriander</p>
                <p class="card-text mb-1">L4</p>
                <p class="card-text">12:20 - 13:15</p>
            </div>
        </div>
        <div class="timetabled-lesson subject-3 card">
            <div class="card-body">
                <h5 class="card-title">Further Maths</h5>
                <p class="card-text mb-1">Mr Beetroot</p>
                <p class="card-text mb-1">R20</p>
                <p class="card-text">14:30 - 15:25</p>
            </div>
        </div>
        <div class="timetabled-lesson subject-4 card">
            <div class="card-body">
                <h5 class="card-title">Chess Club</h5>
```

```
                <p class="card-text mb-1">Mr Watermelon-Tomato</p>
                <p class="card-text mb-1">R4</p>
                <p class="card-text">16:45 - 17:45</p>
            </div>
        </div>
    </div>
{% endblock %}
```

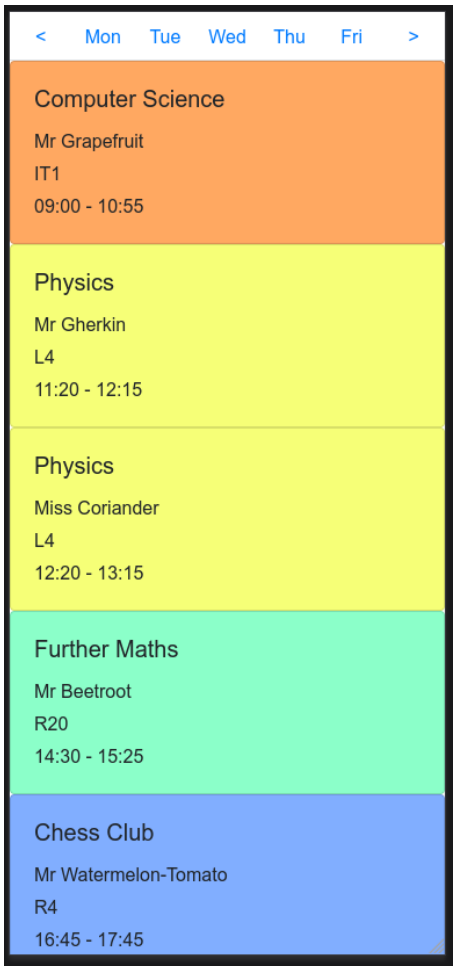
style.css

```
.login-width {
    width: 350px;
}

.subject-1 {
    background-color: #ffa861 !important;
}
.subject-4 {
    background-color: #ffabac !important;
}
.subject-2 {
    background-color: #f6ff77 !important;
}
.subject-5 {
    background-color: #a4ff77 !important;
}
.subject-3 {
    background-color: #8bffc9 !important;
}
.subject-6 {
    background-color: #87eefc !important;
}
.subject-4 {
    background-color: #81aeff !important;
}
.subject-7 {
    background-color: #b49afa !important;
}
.subject-9 {
    background-color: #ffa7f7 !important;
}
```

The teacher interface would be the same as the student interface with different placeholder data. Therefore, it would not be a good use of time to create separate placeholder data and also test this interface.

4.1.5.4.2 Testing (on mobile)

Testing was conducted throughout development. On the iPhone 11 Pro environment used earlier, the interface looks excellent.



4.1.5.4.3 Testing (on desktop)

It was established earlier that the mobile interface takes precedence over the desktop interface. This is because it is far more convenient to view the timetable on a mobile device than a desktop. However, some users may find a desktop more convenient or may not have access to a mobile device. Therefore, it is important that the interface works on desktop, even if suboptimal.

Clearly, this interface is not as good on desktop as it is on mobile. A better interface to make good use of the space available would display all days at once so that the whole week can be viewed on

one screen. However, this will depend on the time available for the project. The interface is entirely functional on desktop and can be used if a mobile device is not available.

4.1.5.5 Scheduled Lessons List

This was designed in **4.1.4.5.2**.

4.1.5.5.1 Development

Various Bootstrap classes were given to space out content appropriately, such as:

- `m-4` gives a large margin in all directions
    - This makes the header stand out
- `my-2` gives a small margin in the vertical direction
    - This gives a small gap between each lesson so that they are not too close together
- `mb-0` removes the bottom margin from the element
    - This removes unnecessary whitespace so that more content can be fit on the page

The colour for lessons that have already been scheduled had to be defined manually by creating a class `scheduled` and giving it an appropriate colour (a dark grey). This grey must be chosen such that the contrast with the text meets accessibility standards.

Note that placeholder data is still being used and this code will not be representative of the final solution (which will query the database). Placeholder data is used to focus on the design of the interface without having to implement the database access yet.

`scheduled_list.html`

```
{% extends "timetable/base.html" %}
{% block content %}
    <h1 class="m-4">Scheduled Lessons</h1>
    <div class="card scheduled my-2 mx-4">
        <div class="card-body">
            <p class="float-left mb-0">Specific Heat Capacity</p>
            <p class="float-right mb-0">50 minutes</p>
        </div>
    </div>
    <div class="card my-2 mx-4 bg-light">
        <div class="card-body">
            <p class="float-left mb-0">s.h.c practical</p>
            <p class="float-right mb-0">140 minutes</p>
        </div>
    </div>
    <div class="card my-2 mx-4 bg-light">
        <div class="card-body">
            <p class="float-left mb-0">[untitled]</p>
            <p class="float-right mb-0">65 minutes</p>
        </div>
    </div>
    <div class="card my-2 mx-4 bg-warning">
        <h5 class="mx-2 mt-2">Not enough lessons scheduled</h5>
        <p class="mx-2">Please ensure you always have enough lessons
scheduled in advance. If you do not schedule enough lessons, 55-minute
```

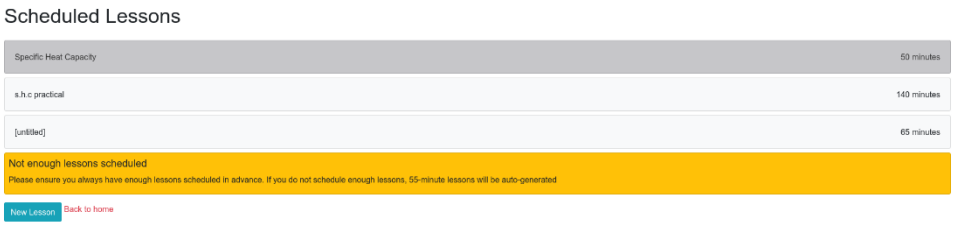
```
lessons will be auto-generated</p>
    </div>
    <a href="/teacher/schedule"><button class="btn btn-info ml-4 mt-2">New
Lesson</button></a>
    <a href="/" class="text-danger">Back to home</a>
{% endblock %}
```

(extract from) `style.css`

```
.scheduled {
    background-color: #c6c6c9 !important;
    color: #3b3b3b !important;
}
```

4.1.5.5.2 Testing (on desktop)

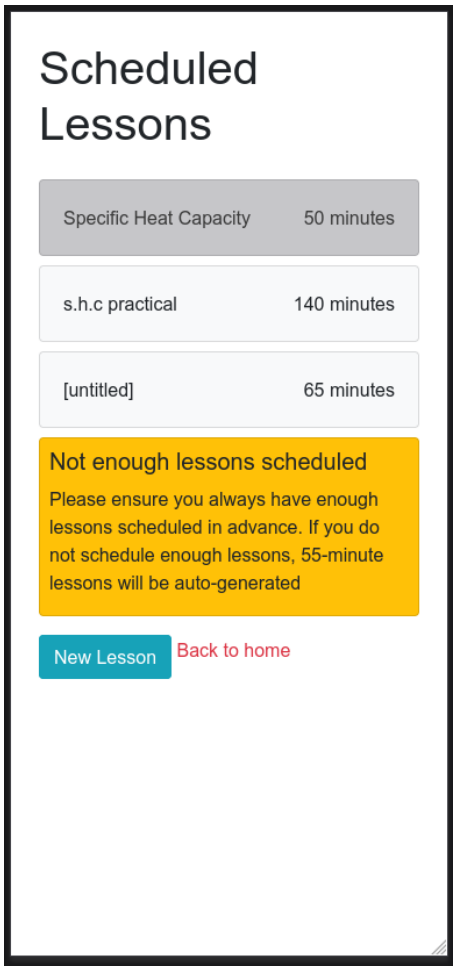
As usual, testing was done throughout the development. The final interface is as shown:



This interface is good, however does have a large amount of unused space. This won't be an issue, but is worth noting that this screen space is available for potential functionality in the future.

4.1.5.5.3 Testing (on mobile)

It is expected that teachers will be using desktop while doing this task, however this interface looks and functions just as well, if not better, on mobile.



Teachers are able to quickly and easily schedule their lessons from any device. This helps the process of scheduling lessons be as effortless as possible. This exceeds the requirements initially set.

4.1.5.6 Schedule Lesson Form

4.1.5.6.1 Development

Some of the style can be taken directly from the login form, as these two interfaces are similar.

The predefined options for the duration needed to be designed such that they would sit inline with the other elements and still allow enough space for the text box. This was possible, and the text box could be configured to occupy the remaining space.

`w-100` is a tag that makes an element occupy 100% of the width of the parent element. This was useful so that the overall width can be configured once, then all other elements can be set to equal width.

`schedule.html`

```
{% extends "timetable/base.html" %}
{% block content %}
    <div class="text-center login-width m-auto">
        <h1 class="my-4">Schedule a Lesson</h1>
        <form class="text-left">
```

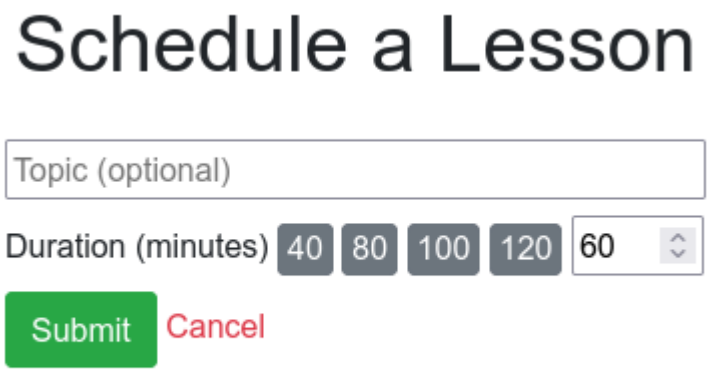
```
            <label class="w-100">
                <input class="w-100" placeholder="Topic (optional)">
            </label>
            <label class="w-100" for="duration-input-number">Duration
(minutes)
                <button class="btn-secondary btn px-1 py-0">40</button>
                <button class="btn-secondary btn px-1 py-0">80</button>
                <button class="btn-secondary btn px-1 py-0">100</button>
                <button class="btn-secondary btn px-1 py-0">120</button>
                <input id="duration-input-number" type="number" step="5"
value="60">
            </label>
            <input type="submit" value="Submit" class="btn btn-success">
            <a href="/teacher/scheduled" class="text-danger">Cancel</a>
        </form>
    </div>
{% endblock %}
```

`style.css` (extract)

```
#duration-input-number {
    width: 66px;
}
```

4.1.5.6.2 Testing (on desktop)

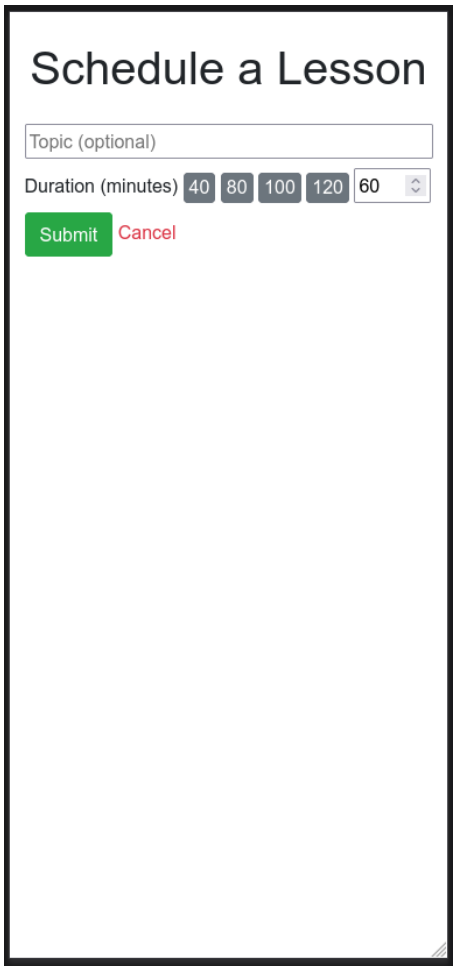
The final interface is as follows:



The up and down arrows did go up and down by 5 each time, as designed (the timetabling algorithm will use time increments of 5 minutes).

4.1.5.6.3 Testing (on mobile)

The interface also functions as expected on mobile, ensuring teachers are able to schedule lessons at any time that is convenient.



The user interface appears concentrated at the top of the page and is not using up all of the space available. In practice, this is not an issue while using the interface. Occupying more space with no useful content would not improve the product in any way and would only overwhelm the user with unnecessary information.

4.1.5.7 Reviewing the User Experience

Overall, the user experience is great. It is very intuitive to navigate through the website as both a student and a teacher (the admin interface will be addressed later).

However, there was no link to access the teacher scheduler, only the teacher timetable. This would be very confusing for teachers who want to schedule lessons and cannot find a link to do so. Therefore, the initial flow through the website was changed for teachers.

Flow for students: Home (choose user type) -> Login -> Timetable

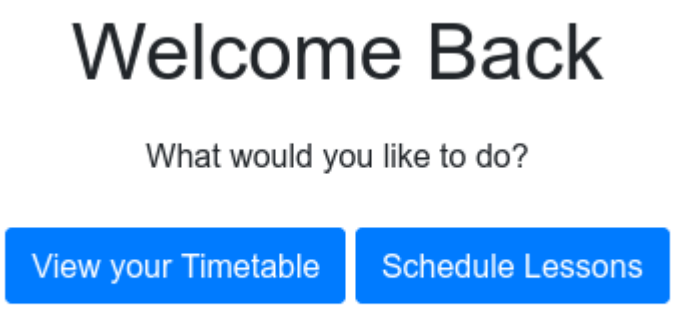
Flow for teachers: Home (choose user type) -> Login -> Teacher -> Timetable / Scheduler

However, there is another possibility for the teacher flow. It could be more efficient to divide the 'Teacher' button into two separate buttons which each redirect directly to the relevant interface, avoiding the need for a teacher webpage.

Using two separate buttons, teachers would choose their desired interface directly from the homepage, saving a click. However, this is also presenting more information to the user at once which could risk them being confused. Students, which make up the majority of the users, also do not need to see this option so this could potentially confuse them. It is also easier for teachers to navigate between parts of the teacher interface without a student login button present.

It was determined that separating the buttons was too confusing and had a detrimental impact on the student experience. Therefore, an entirely new page has been created. Some URLs have been changed.

Clicking on `Teacher` on the homepage leads to `/teacher` which is as follows:



The teacher timetable has been moved to `/teacher/timetable`. For consistency, the student timetable is also accessible under `/student/timetable`, as well as the original `/student`. The redirections in the interface should favour `/student/timetable` for consistency.

This interface is very similar to the homepage, which adds to the simplicity of the design and reduces unnecessary development time.

4.1.6 User Account Management

4.1.6.1 Exploring Possibilities

4.1.6.1.1 Exploring how user accounts should be stored

The first step in programming the interface will be to explore how user accounts should be stored.

In the planned data structure student accounts will be stored in a `Students` table and teacher accounts will be stored in a `Teachers` table. However, it was not known at this point that Django has support for user account management. Using Django's option will have many benefits, such as:

- It will save development time, allowing for efforts to be focused elsewhere in the project (such as improving the core timetabling algorithm)
- Django's account management will not only be developed to incorporate all industry standard security features, but will be regularly updated to patch new issues. This is a key feature that helps the maintenance of the product as some security patches can be done

automatically. Therefore, maintenance time can be focused on developing new functionality rather than addressing security concerns. These features include:
    - Storing passwords as a salted hash
    - Checking strength of users' passwords
    - Encryption
- Custom fields can be added to the table to store custom information
- More flexible permissions can be allocated to users, rather than just binary student/teacher
- The admin account to login to the admin portal will use the same credentials as the other login page

There are no clear drawbacks of using Django's user account management.

4.1.6.1.2 Redesigning the User Flow

A key change as a result of this will involve storing teachers and students in the same database table. This has many advantages, notably facilitating a unified login page for all user types. This also means that the home page is redundant, as it becomes immediately clear whether a user is a student or a teacher when they login. The new user flow is as follows:

Students: Login -> Timetable

Teachers: Login -> Teacher -> Timetable / Schedule

Clearly, this is shorter than what was designed in **4.1.5.7**. This helps to prevent unnecessary steps and avoid wasting user time.

This will involve the removal of the home page, and a redesign of the login page to facilitate logging in for any user.

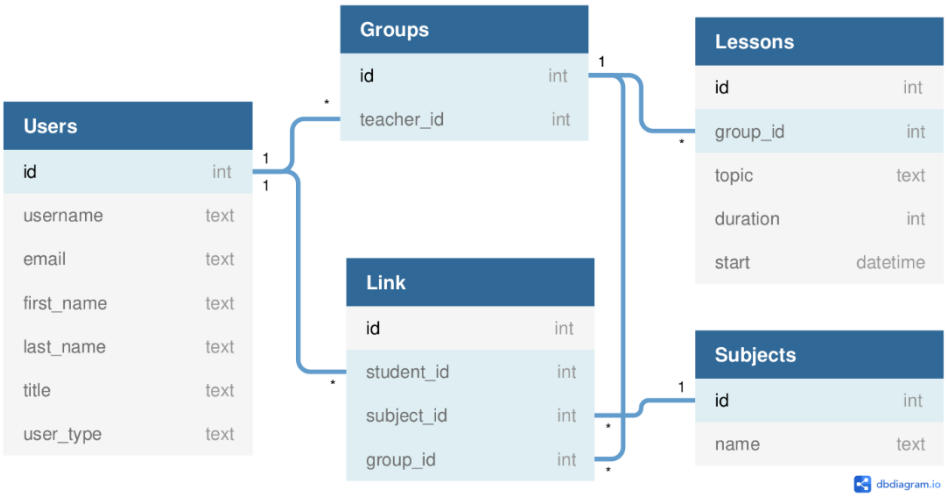
The admin flow would also be as follows:

Login -> Admin Page

4.1.6.1.3 Redesigning the Database Structure

*[Previous Iteration: **4.1.3.3** Next Iteration: **4.1.7.1**]*

Students and teachers are now being stored under a single unified Users table. This affects the data structure as shown:



NOTE: One-to-many relationships are indicated by a `1` *and an* `*` *at the relevant ends.*

4.1.6.2 Setting up the Database Structure

4.1.6.2.1 Implementing the Database Structure

First, `models.py` was changed to include a new `User` class to replace the previous `Student` and `Teacher` classes. Note that the class inherits from `AbstractUser` which contains the basic code to handle users in Django. Character limits were added to ensure data is not too long and that it displays appropriately on the interface.

```
class User(AbstractUser):
    title = models.CharField(max_length=8, default='')
    first_name = models.CharField(max_length=16)
    last_name = models.CharField(max_length=16)
    user_type = models.CharField(max_length=8, default='admin')


class Subject(models.Model):
    name = models.CharField(max_length=16)


class Link(models.Model):
    student_id = models.ForeignKey(
        settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE,
    )
    subject_id = models.ForeignKey(
        'Subject',
        on_delete=models.CASCADE
    )
    group_id = models.ForeignKey(
        'Group',
        on_delete=models.CASCADE
    )
```

```
class Group(models.Model):
    teacher_id = models.ForeignKey(
        'User',
        on_delete=models.SET_NULL,
        null=True
    )


class Lesson(models.Model):
    group = models.ForeignKey(
        'Group',
        on_delete=models.CASCADE
    )
    duration = models.DurationField()
    topic = models.CharField(max_length=128)
    start = models.DateTimeField()
```

Next, `admin.py` was updated to contain a link to this new model

```
admin.site.register(User, UserAdmin)
```

Finally, `settings.py` was modified to indicate that this class was the class to use for storing user accounts.

```
AUTH_USER_MODEL = User
```

4.1.6.2.2 Testing the Database Structure

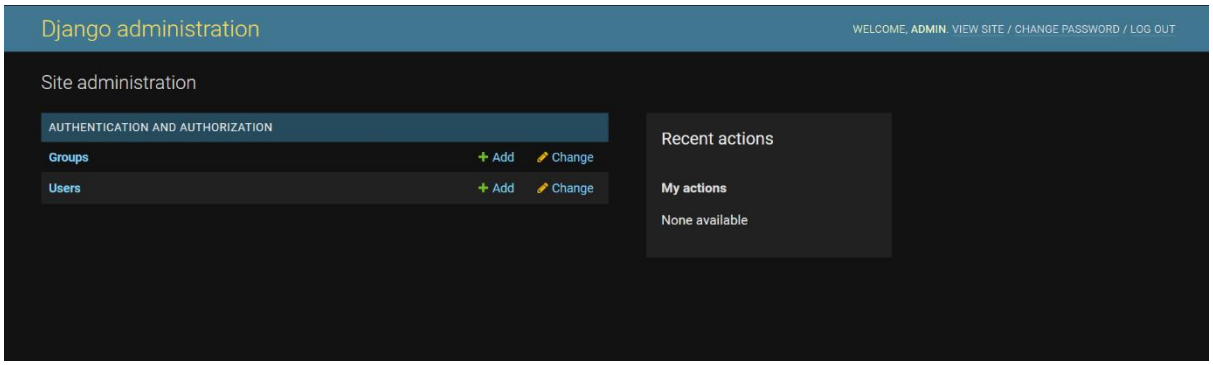
After fixing a few minor issues, the website was functioning again with the new database structure. This did require deleting and re-creating the database (as significant changes were made to the structure), however this was not an issue as there is currently not any data stored in it.

4.1.6.3 Creating Users on the Admin Page

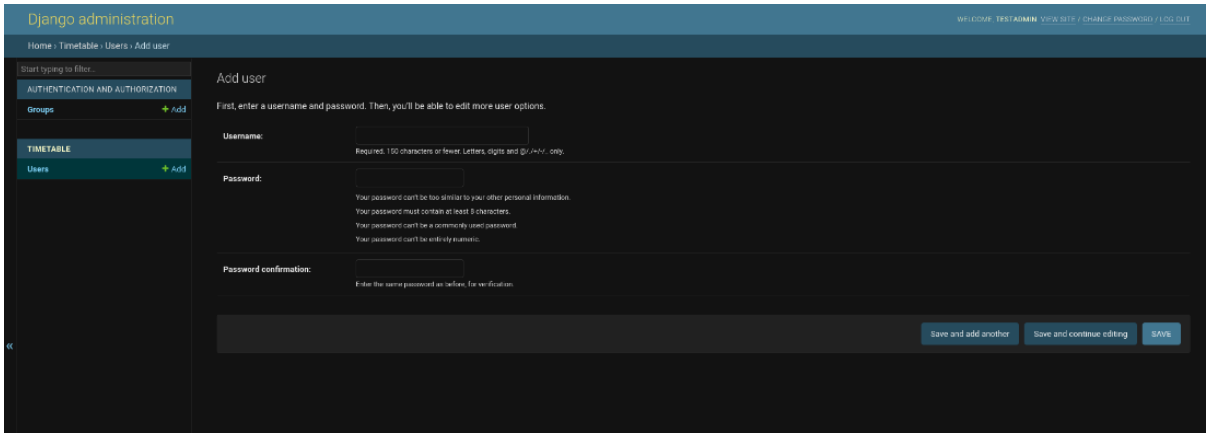
4.1.6.3.1 Creating the Admin User

To access the admin page, an admin user (known as a superuser) must be created. This was done with the command `python manage.py createsuperuser`.

Navigating to `/admin` and entering these credentials yields the following page:



The admin could easily add a new user by clicking Add.



However the next screen showed that there were not any options to specify the custom fields.

4.1.6.3.2 Creating Users

The `User` class inherits from the Django base class `AbstractUser`. A few fields are created to offer basic functionality inherent to users, such as storing full name and which type of user they are. `user_type` will store either `'student'` or `'teacher'` (or `'admin'`).

Forms must then be created to allow the admin to create and edit user information. Again, this can inherit from the `UserCreationForm` class which includes some fields already configured, such as password handling.

The title and user types were implemented in a drop-down field as there are a fixed number of valid options. These were stored in constant variables so they can easily be changed later. In Python, the naming convention is to use all uppercase characters when representing a constant. Naming conventions aid readability and maintainability of the code.

The fields for modifying the user were organised into sections: authentication, details, permissions, dates. This makes the interface clearer, so it is quicker for the admin to find the information they need and change it.

`models.py`

```
class User(AbstractUser):
    title = models.CharField(max_length=8, default='')
    first_name = models.CharField(max_length=16)
```

```
    last_name = models.CharField(max_length=16)
    user_type = models.CharField(max_length=8, default='admin')
```

admin.py

```
TITLE_CHOICES = [('', "It doesn't matter"),
                 ('mr', 'Mr'),
                 ('mrs', 'Mrs'),
                 ('miss', 'Miss'),
                 ('ms', 'Ms'),
                 ('dr', 'Dr')]
USER_TYPES = [('student', 'Student'),
              ('teacher', 'Teacher')]


class CustomUserCreationForm(UserCreationForm):
    title = forms.ChoiceField(choices=TITLE_CHOICES)
    first_name = forms.CharField()
    last_name = forms.CharField()
    user_type = forms.ChoiceField(choices=USER_TYPES)
    email = forms.EmailField()

    class Meta(UserCreationForm.Meta):
        model = User
        fields = ('username', 'email', 'first_name', 'last_name',
'password1', 'password2', 'user_type',)


class CustomUserChangeForm(UserChangeForm):
    title = forms.ChoiceField(choices=TITLE_CHOICES)
    first_name = forms.CharField()
    last_name = forms.CharField()
    user_type = forms.ChoiceField(choices=USER_TYPES)
    email = forms.EmailField()

    class Meta(UserChangeForm.Meta):
        model = User
        fields = ('username', 'email', 'first_name', 'last_name',
'password')


class CustomUserAdmin(UserAdmin):
    add_form = CustomUserCreationForm
    form = CustomUserChangeForm
    add_fieldsets = (
        (None, {
            'classes': ('wide',),
            'fields': ('username', 'email', 'title', 'first_name',
'last_name', 'user_type', 'password1', 'password2'),
        }),
    )
    fieldsets = (
        (_('Authentication'), {'fields': ('username', 'email',
```

```
'password')}),
        (_('Details'), {
            'fields': ('first_name', 'last_name', 'title', 'user_type'),
        }),
        (_('Permissions'), {
            'fields': ('is_active', 'is_staff', 'is_superuser', 'groups',
'user_permissions'),
        }),
        (_('Important dates'), {'fields': ('last_login', 'date_joined')}),
    )


admin.site.register(User, CustomUserAdmin)
```

4.1.6.4 The Login Page

In the same way as the forms for creating/editing users were created, the login form was created as a class inheriting from `AutheticationForm`. The two fields could then be created as they appear on the HTML, with appropriate classes, IDs and placeholders.

`forms.py`

```
class LoginForm(AuthenticationForm):
    username = UsernameField(widget=forms.TextInput(
        attrs={'class': 'w-100 mb-3',
               'placeholder': 'Username',
               'id': 'username'}),
        label='')
    password = forms.CharField(widget=forms.PasswordInput(
        attrs={
            'class': 'w-100 mb-2',
            'placeholder': 'Password',
            'id': 'password',
        }),
        label='')
```

Next, this form must be displayed on the login page. The old elements for the input fields were replaced with this new form, and the form was passed in from `views.py`. Note that the submit button still needs to be added separately. The CSRF token was added to prevent CSRF attacks.

`login.html`

```
{% extends "timetable/base.html" %}
{% block content %}
    <div class="text-center login-width m-auto">
        <h1 class="my-4">Log in</h1>
        <form method="POST" class="text-left">
            {% csrf_token %}
            {{form}}
            <input type="submit" class="btn btn-success mt-2"
value="Login">
        </form>
```

```
    </div>
{% endblock %}
```

The logout page was a very simple page to simply inform the user that they have logged out. A link was added in case they want to log in again, making the website easier to navigate.

`logout.html`

```
{% extends "timetable/base.html" %}
{% block content %}
    <div class="text-center login-width m-auto">
        <h1 class="my-4">Logged out</h1>
        <p>You have successfully logged out</p>
        <a href="/">Log in again</a>
    </div>
{% endblock %}
```

The login redirect function was created to redirect the user to the correct webpage based on the logged in user type.

- If not logged in:
    - Redirect to `/login/`
- If logged in as student:
    - Redirect to `/student/`
- If logged in as teacher:
    - Redirect to `/teacher/`
- If logged in as admin (or erroneous data given):
    - Redirect to `/admin/`
    - If the user type is incorrect and it redirects to `/admin/`, this user will not be able to do anything on the admin site and will simply be prompted to log in with a staff account.

Other functions were modified, as shown below:

`views.py`

```
def login_redirect(request):
    # redirect appropriately depending on user type
    user = request.user
    if isinstance(user, AnonymousUser):
        return redirect('/login/')
    else:
        if user.user_type == 'student':
            return redirect('/student/')
        elif user.user_type == 'teacher':
            return redirect('/teacher/')
        else:
            return redirect('/admin/')


@login_required
```

```
def student_timetable(request):
    return render(request, 'timetable/timetable.html')


@login_required
def teacher(request):
    return render(request, 'timetable/teacher.html')


@login_required
def teacher_timetable(request):
    return render(request, 'timetable/timetable.html')


@login_required
def teacher_scheduler(request):
    return render(request, 'timetable/scheduling/schedule.html')


@login_required
def teacher_scheduled(request):
    return render(request, 'timetable/scheduling/scheduled_list.html')
```

urls.py

```
urlpatterns = [
    path('login/',
django_views.LoginView.as_view(template_name='timetable/login.html',
authentication_form=LoginForm),
         name='timetable-login'),
    path('logout/',
django_views.LogoutView.as_view(template_name='timetable/logout.html'),
name='timetable-logout'),

    path('', views.login_redirect, name='timetable-login-redirect'),

    path('student/', views.student_timetable, name='timetable-student'),
    path('student/timetable', views.student_timetable, name='timetable-
student'),

    path('teacher/', views.teacher, name='timetable-teacher'),
    path('teacher/timetable', views.teacher_timetable, name='timetable-
teacher'),
    path('teacher/scheduled', views.teacher_scheduled, name='timetable-
teacher-scheduled'),
    path('teacher/schedule', views.teacher_scheduler, name='timetable-
teacher-schedule'),
]
```

4.1.7 Displaying the Timetable

Now that the user interface is functional and users are able to log in, it is necessary to format the timetable based on the information stored in the database. This will require some additional processing as the timetable is not directly stored under each user, but instead in a `Lessons` table.

The best approach would be to retrieve all relevant lessons in the function in `views.py` to generate an array of lessons, which can then be iterated over to format the lessons within the template itself. While templates allow for iteration, they are not designed for complex database queries. This breaks the problem down into two sub-problems:

1) Query database to get array of lessons
2) Iterate within template and format

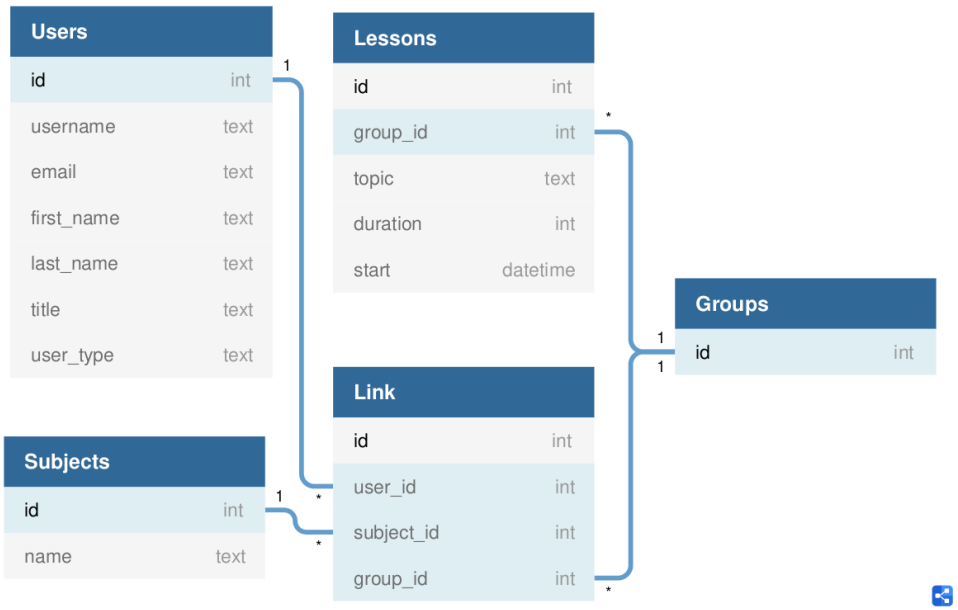
4.1.7.1 Restructuring the Database

*[Previous Iteration: **4.1.6.1.3** Next Iteration: **4.1.8.1**]*

Upon inspecting the existing database structure, it became clear that it would be inconvenient to traverse two different routes depending on if the user is a student or a teacher. To fix this issue, teachers will now have the same relationships as students. This has a few benefits:

- Teachers are stored along with their subject(s)
- Database traversal is easy
- Some bits of code will no longer have to distinguish between teachers and students as both can be handled in exactly the same way.

The new database structure is as follows:



This was implemented by removing the field `teacher_id` from the class `Groups` and renaming `student_id` to `user_id` in `Link`.

4.1.7.2 Retrieving the Lessons

4.1.7.2.1 Implementation

The first step is to find all lessons that involve a specific user in a specific day. Note that with the data structure it does not matter whether this user is a student or a teacher – the database can be traversed in the same way. However, they will need to be formatted slightly differently.

The filter queries were complex and many consisted of multiple steps, so these were only finished while testing in **4.1.7.2.3**.

The code is available in **4.1.7.2.3**.

4.1.7.2.2 Test Data

This could not be tested without some test data being created that can be formatted.

To make all tables accessible from the admin page, they need to be registered in `admin.py`

```
admin.site.register(User, CustomUserAdmin)
admin.site.register(Lesson)
admin.site.register(Group)
admin.site.register(Link)
admin.site.register(Subject)
```

This allows the administrator to easily add all subjects, users and groups, and then allow the algorithm to create records in `Link` and lessons.

In this situation, all records will be generated manually to test this specific functionality. Only one subject and one group are necessary, while two records in `Link` were created to have the test user and test teacher both part of the class.

The subject had name `Maths` and abbreviation `Ma`. The group had name `Ma1`. The teacher had name `Mr Hamilton`. The test lesson had duration `115` minutes, start time `9:00`, room `'Room'` and topic `'This is a test lesson'`.

4.1.7.2.3 Testing

After the filter queries were completed correctly (many of these have multiple steps), the code worked and printed out the correct lists when logged in as a student and a teacher.

The final code, after testing was completed, is as follows:

`views.py`

```
user = request.user
lessons = []
for lesson in
Lesson.objects.filter(group_id__link__user_id__username__exact=user.usernam
e):
    lesson_data = []
    if user.user_type == 'student':

lesson_data.append(Subject.objects.filter(link__group_id__lesson__id__exact
=lesson.id)[:1].get().name)
```

```
        teacher =
User.objects.filter(link__group_id__lesson__id__exact=lesson.id,
user_type='teacher')[:1].get()
        if teacher.title:
            lesson_data.append(teacher.title.title() + ' ' +
teacher.last_name.title())
        else:
            lesson_data.append(teacher.first_name.title() + ' ' +
teacher.last_name.title())
    else:

lesson_data.append(Group.objects.filter(lesson__id__exact=lesson.id)[:1].ge
t().name)
        topic = lesson.topic
        if len(topic) > 32:
            topic = topic[:29]+'...'
        lesson_data.append(topic)
    lesson_data.append('Room')  # Room allocation will be completed later
    start = lesson.start
    end = lesson.start + lesson.duration
    lesson_data.append(start.strftime('%H:%M') + ' - ' +
end.strftime('%H:%M'))
    lessons.append(lesson_data)
print(lessons)
return render(request, 'timetable/timetable.html', context={'lessons':
lessons})
```

A print statement was added to test the functionality of this code in isolation.

When logged in as a student, the program printed `[['Maths', 'Mr Hamilton', 'Room', '09:00 - 10:55']]`

When logged in as a teacher, the program printed `[['Ma1', 'This is a test lesson', 'Room', '09:00 - 10:55']]`

The print statement can now be removed, and the template can be adjusted to format this data on the webpage.

4.1.7.3 Formatting the Lessons

Regardless of whether the user is a student or a teacher, the data passed in will be a 2D array of lessons, each containing 4 pieces of information. The first piece of information should be displayed with a larger font size.

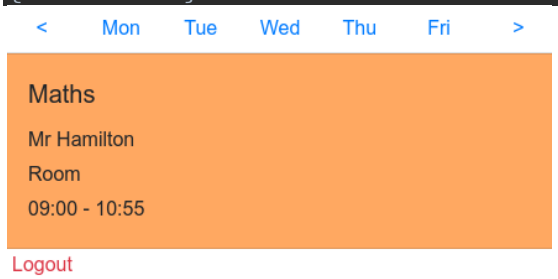
Adjusting the template to format the lessons was relatively easy.

Note that `forloop.counter0` is zero-indexed and `forloop.counter` begins at 1.

`timetable.html`

```
{% for lesson in lessons %}
<div class="timetabled-lesson subject-{{forloop.counter}} card">
    <div class="card-body">
```

```
        <h5 class="card-title">{{lesson.0}}</h5>
        <p class="card-text mb-1">{{lesson.1}}</p>
        <p class="card-text mb-1">{{lesson.2}}</p>
        <p class="card-text">{{lesson.3}}</p>
    </div>
</div>
{% endfor %}
```



4.1.7.4 Choosing the Date

4.1.7.4.1 Filtering by Date

While the above format works, instead of finding only lessons on a specific date, it finds all lessons. This can be fixed by updating the filter query as follows:

```
current_date = datetime.datetime.now()
after = current_date.replace(hour=0, minute=0, second=0)
before = current_date.replace(hour=23, minute=59, second=59)
for lesson in
Lesson.objects.filter(group_id__link__user_id__username__exact=user.usernam
e, start__gte=after, start__lte=before):
```

This now only finds lessons on the given `current_date`, by modifying this object for the start and end of the day and finding records after the start of the day and before the end of the day.

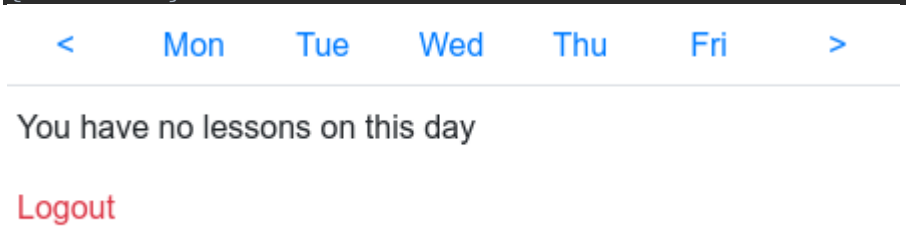
Note that if a lesson was to span multiple days, it would only be displayed on the first day rather than all days. However, it is not expected that there would be any lessons spanning multiple days, and this behaviour is intuitive.

As it is now the day after the test lesson was created, it is expected that it will no longer display on the website. This is exactly what was observed.

However, this interface doesn't look very professional, so an additional note has been added if there are no lessons on the day being viewed.

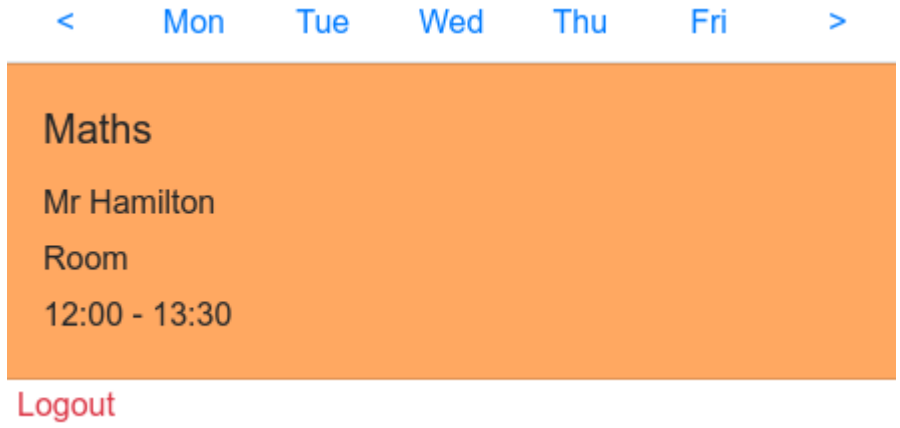
timetable.html

```
{% if not lessons %}
<p class="text ml-2 mt-2">You have no lessons on this day</p>
{% endif %}
```



As expected, manually changing the current date to be yesterday has now displayed the lesson again and the message indicating there are no lessons has been removed

```
current_date = datetime.datetime(2022, 3, 12, 0, 0, 0)
```



However, the timetable doesn't only display lessons for the current date but allows choosing any day of the week (and other weeks) to view lessons for. This will require implementing:

- Choosing the date via the top bar
- Automatically selecting the current date in the top bar
- Showing lessons based on the current state of the top bar

4.1.7.4.2 Automatically selecting the current date

Before the date was automatically selected, the code for the navigation bar was improved to reduce the amount of copied and pasted code. Code that is copied increases file sizes and reduces the maintainability of the product.

A list was created to store the weekdays

```
weekdays = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri']
weekday: int = current_date.weekday()
```

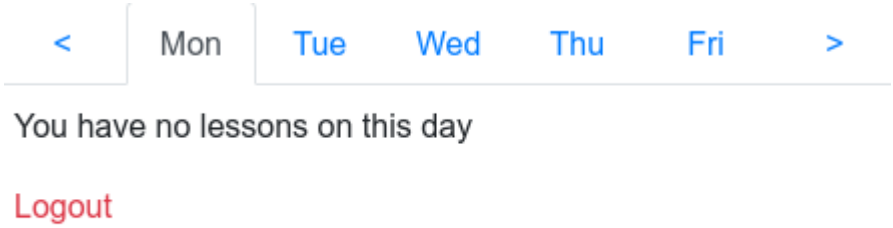
A type hint was added to make it clear that weekday is an integer (0 = Monday) rather than a string.

This was iterated over in the template, showing as active if it is the current date

```
{% for day in weekdays %}
<li class="nav-item">
    <a href="#" class="nav-link px-0{% if weekday == forloop.counter0 %}
active{%endif%}">{{day}}</a>
</li>
{% endfor %}
```

This also makes it easier to add additional days (e.g. Saturday school) should they be required in a future version of the product.

At the time of writing, it is a Sunday so none of the days show as active. Manually modifying `current_date` to another day correctly highlights the day of the week.



4.1.7.4.3 Choosing the Date

Currently, the top bar is not clickable. However, it can easily be configured to contain a link that refreshes the page on a different day.

This can be done by passing in a parameter `day` which contains a UNIX timestamp pointing to the relevant date. Upon loading the page, the links in each button would be updated to point to the relevant days, so that if the user clicks on them they will navigate appropriately.

This could be done by creating two variables to pass into the template:

- `weekday_format`: Represents the weekday in text as it is shown as `'Mon'`, `'Tue'` etc.
- `weekdays_links`: A `dict` of the form `{weekday: link}`, where `weekday` is the exact text to be shown (or >, <) and `link` contains the timestamp to go into the `day` parameter of the URL

The current state of the function `timetable()` in `views.py` (up to the query for lessons) is as follows:

```
user = request.user
day = request.GET.get('day')

if day and day.isdigit():  # user could have modified it to not be an int
    day = int(day)
    current_date = datetime.datetime.utcfromtimestamp(int(day))
else:
    current_date = datetime.datetime.utcnow()
weekday: int = current_date.weekday()
weekstart = current_date - datetime.timedelta(days=weekday)
after = current_date.replace(hour=0, minute=0, second=0)
```

```
before = current_date.replace(hour=23, minute=59, second=59)

weekdays = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri']
weekdays_links = {'<': math.floor((weekstart -
datetime.timedelta(days=3)).timestamp())}
for i in range(5):
    weekdays_links[weekdays[i]] = math.floor((weekstart +
datetime.timedelta(days=i)).timestamp())
weekdays_links['>'] = math.floor((weekstart +
datetime.timedelta(days=7)).timestamp())
if weekday <= 4:
    weekday_format = weekdays[weekday]
else:
    weekday_format = None
```

The navbar in `timetable.html` has also been simplified further:

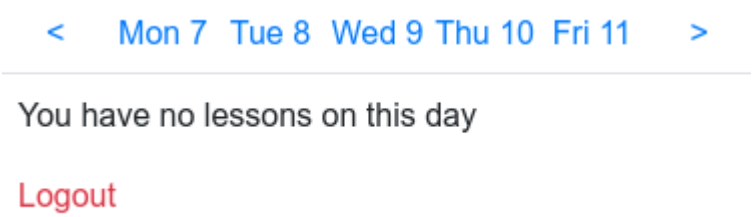
```
<ul class="nav nav-tabs nav-justified">
    {% for day, link in weekdays_links.items %}
    <li class="nav-item">
        <a href="?day={{link}}" class="nav-link px-0{% if weekday_format ==
day %} active{%endif%}">{{day}}</a>
    </li>
    {% endfor %}
</ul>
```

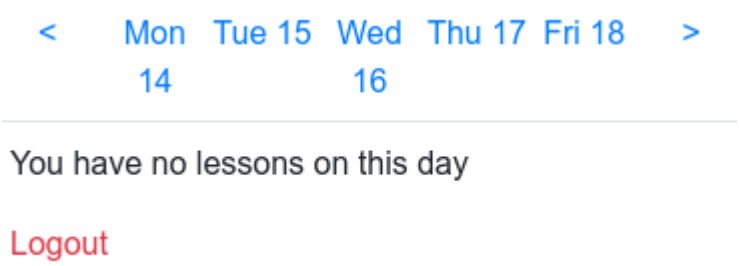
This is significantly shorter and does not involve repeating code.

4.1.7.4.4 Updating the Interface

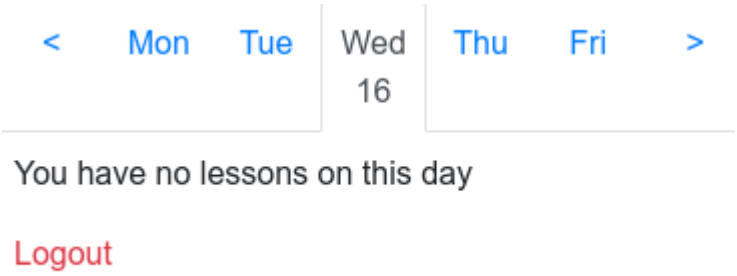
Navigating this interface works perfectly – clicking on the days highlights them, refreshing the lessons, and clicking on the left and right arrows correctly navigates to the next Monday or the previous Friday. However, it is not clear through the interface which week is actually being viewed.

Putting the date in the top of all days was tried, however was ultimately decided against because it occupied too much space on a mobile device and some were displayed across multiple lines.

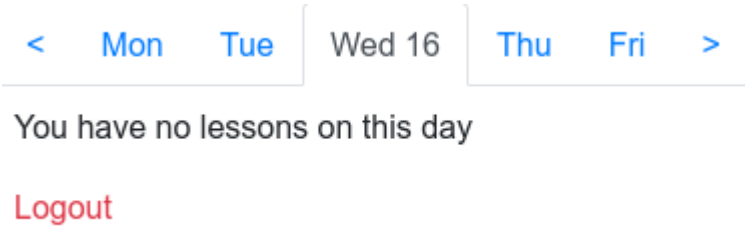




Next, it was tried to only include the date for the lesson the user has selected.



However, this did not solve the issue as the navigation bar was being rendered with equal width to all elements. This was fixed by changing the nav style from `nav-justified` to `nav-fill`.

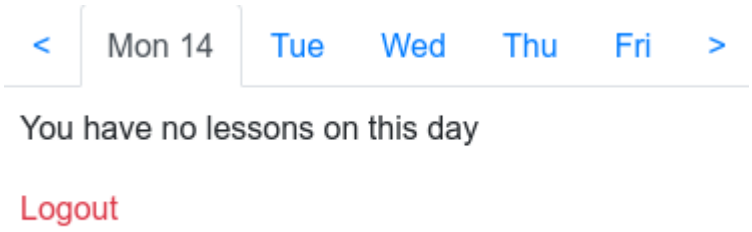


`nav-fill` gives equal padding to all elements, but adjusts the size depending on how much space is needed.

This interface looks excellent. It offers as much information as is necessary in a clear and intuitive format while abstracting away other information such as the month (the timetable is not accessible for more than two weeks in advance).

Additionally, when accessing the timetable on the weekend, it was showing the timetable for the past week. This did not make sense, so instead it has been changed so that if viewed on a Saturday or Sunday, it displays the timetable as if it were already the next Monday.

```
if weekday >= 5:
    current_date = current_date + datetime.timedelta(days=7-weekday)
    weekday = 0
```

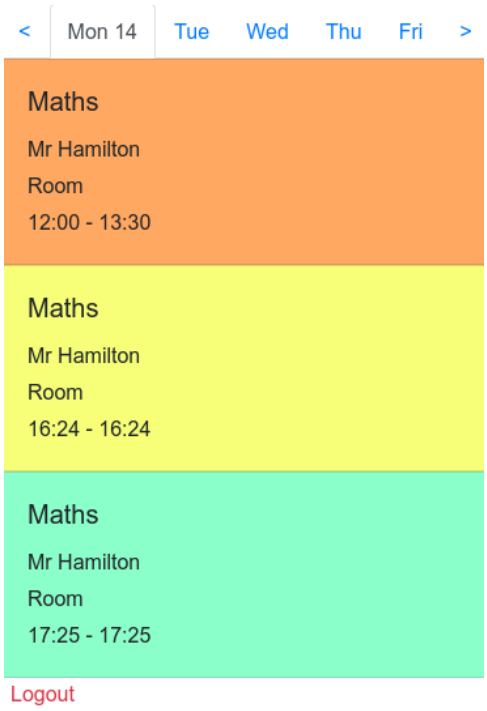


4.1.7.5 Final Testing

The user interface for students and teachers viewing their timetable has now been finished. While it has been tested extensively throughout the development phase, further testing will be conducted in this section and in the evaluation phase to ensure the interface works as expected.

The test lesson was moved to be on a school day. A few new lessons were created, some with different students/teachers to test whether it is being properly filtered in the database.

Logging in as a student (on Sunday) correctly showed the timetable for the upcoming Monday:



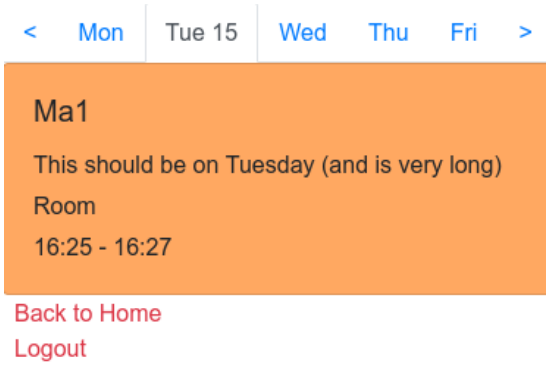
It did not show the lesson which was scheduled on this day but did not involve this student.

Logging in as a teacher also worked perfectly:

All lessons were correctly shown (except for the one that did not involve this teacher, which was correctly hidden). One of the topics was deliberately long and this was appropriately truncated. However, it could be left a bit longer.

This allows for more content without it potentially spanning multiple lines on smaller devices (many different devices were tested, and none were found to be too small).

Navigating to Tuesday correctly showed the lesson, however was meant to be 2 hours and was instead 2 minutes.



This was an issue with data entry, as if 120 is directly entered into the admin panel it is assumed to be 2 minutes rather than 2 hours. This issue can easily be avoided by entering `2:0:0` in the duration field. However, it will never be necessary to manually enter lessons should the scheduling algorithm work properly.

4.1.8 Scheduling Lessons

4.1.8.1 Changing the Database Structure

*[Previous Iteration: **4.1.7.1**]*

Lessons that have not been scheduled could be stored in a separate table, or they could exist in the `Lessons` table with `start` set to `null`. With this approach, a field `fixed` can determine if the lesson can still be rescheduled or if it is now fixed at a specific time.

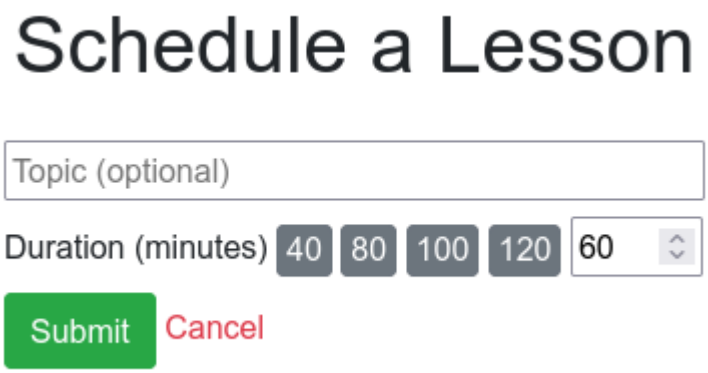
```
start = models.DateTimeField(null=True)
fixed = models.BooleanField()
```

This would be preferable to a separate table as it reduces the number of tables and makes it easier to schedule – instead of deleting the unscheduled lesson and adding a new scheduled one, the start time simply has to be updated.

4.1.8.2 Scheduling Form

4.1.8.2.1 **Current State**

Currently, the interface is as follows:



This needs to be updated to:

- Have the buttons in the duration update the duration field
- Have the submit button send the data to the database
- The following validation should be done both client and server side:
    - Validate the duration to ensure the duration is a multiple of 5 between 30 and 120
    - Truncate the topic field to a maximum length of 128 characters

4.1.8.2.2 Duration Buttons

The following inline JavaScript was added to set the value of the number field

```
<button onclick="document.getElementById('duration-input-
number').value=40;" class="btn-secondary btn px-1 py-0">40</button>
<button onclick="document.getElementById('duration-input-
number').value=80;" class="btn-secondary btn px-1 py-0">80</button>
<button onclick="document.getElementById('duration-input-
number').value=100;" class="btn-secondary btn px-1 py-0">100</button>
<button onclick="document.getElementById('duration-input-
number').value=120;" class="btn-secondary btn px-1 py-0">120</button>
```

Inline JavaScript was used because the code is only one line and would be easier to read than storing in a separate file, even though it does involve some repeating code.

This code worked, however clicking these buttons submitted the form which was not desired (they should only update the value in the box, not submit the form). This was fixed by adding a `type="button"` to the `<button>` elements.

```
<button type="button" onclick="document.getElementById('duration-input-
number').value=40;" class="btn-secondary btn px-1 py-0">40</button>
<button type="button" onclick="document.getElementById('duration-input-
number').value=80;" class="btn-secondary btn px-1 py-0">80</button>
<button type="button" onclick="document.getElementById('duration-input-
number').value=100;" class="btn-secondary btn px-1 py-0">100</button>
<button type="button" onclick="document.getElementById('duration-input-
number').value=120;" class="btn-secondary btn px-1 py-0">120</button>
```

This code works exactly as intended.

4.1.8.2.3 Submitting the Form (client-side)

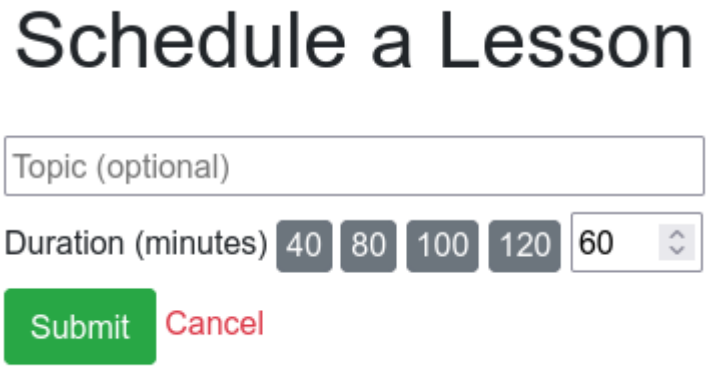
The form can be created in a similar manner to the login form which has already been created. All the elements need to be re-created in a subclass of `Form` which can then be passed in via the view to the template. The `csrf_token` is also passed in to prevent cross-side request forgery attacks.

The buttons could be added to the label by using the function `mark_safe()` to ensure it is rendered properly. A `for` loop was implemented to prevent the repeating code present above and make it easier to change the available durations. All other attributes of the elements were added here until the form looked exactly like it did before.

The validation was also added:

- `max_length` was used to limit the length of the topic field
- the `step` HTML attribute determines the smallest increment allowed for the input

```
class ScheduleForm(forms.Form):
    topic = forms.CharField(widget=forms.TextInput(
        attrs={'class': 'w-100 mb-2',
               'placeholder': 'Topic (optional)',
               'id': 'topic-input'}),
        label='', max_length=128, required=False)
    duration_options = ['40', '80', '100', '120']
    buttons_html = ''
    for duration in duration_options:
        buttons_html += f"""<button type="button"
onclick="document.getElementById('duration-input-
number').value={duration};" class="btn-secondary btn px-1 py-
0">{duration}</button>"""
    duration = forms.CharField(widget=forms.NumberInput(
        attrs={
            'class': 'mb-2',
            'id': 'duration-input-number',
            'step': '5',
            'value': '60'
        }),
        label=mark_safe(f"Duration (minutes) {buttons_html}"))
```



4.1.8.2.4 Processing the Data (server-side)

On submit, this form will send the data in the body of a POST request to the same URL. This must be processed in the view.

As both requests are going to the same URL, they must be processed differently depending on the method. GET requests should load the form, whereas POST requests should process the result from the form. If valid, it should redirect back to the list of lessons.

This can be achieved with the following code:

```
if request.method == 'POST':
    form = ScheduleForm(request.POST)
    if form.is_valid():
        # TODO: Process data
        return redirect('/scheduled')
else:
    form = ScheduleForm(label_suffix='')

return render(request, 'timetable/scheduling/schedule.html', {'form':
form})
```

As validation is already complete, processing only needs to entail the addition of a new record in the database for the newly created lesson.

Upon implementing this, it became apparent that the `group` field was forgotten about in the initial form. This is required to ensure the lesson is scheduled for the correct class.

4.1.8.2.5 Choosing the Group

The best implementation would be having the group selection be a drop-down list containing all the logged in teacher's groups / classes. This would be ideal because:

- There are a limited number of possibilities which would not be overwhelming in a drop-down menu.
- The group names are short and may not otherwise be easy to remember for a text field.
- If adding many lessons, it would be inconvenient to have to type out the group name every time.

Additionally, the selection should:

- Be in a consistent order, so it is easy to add many lessons in a row for one group (instead of finding the correct group in the list each time, it is easy to predict and move the cursor pre-emptively to the correct position)
- Clearly show the year group for each, in the event this is not included in the name of each group (of which a syntax has not yet been decided).
    - It can be assumed that each class will only include one year group. In the event this is not the case, the year group of the first student in the database should be shown.

This will require:

- Querying the database to find a list of the classes that the logged in teacher is in
- Showing these on the drop-down menu

- Validating on submission that the teacher is actually in the class that is input.

4.1.8.2.5.1 Adding the drop-down menu to the form

The drop-down menu can be added as shown:

```
group = forms.CharField(widget=forms.Select(
    choices=group_choices,
    attrs={
        'id': 'group-select'
    }),
    label='Class', required=True)
```

Where `group_choices` is a list of tuples of the form `(group_id, group_name)`.

4.1.8.2.5.2 Finding a list of groups

To allow the value of the form to depend on the logged in user, the form must be modified slightly to accept the request object. This can be done by defining the `__init__()` method, initialising the form and then modifying the field after creation to include the relevant groups.

```
def __init__(self, request, *args, **kwargs): # this is required to receive
the request object
    self.request = request
    # render the form
    super(ScheduleForm, self).__init__(*args, **kwargs)
    # modify the group field to include all the relevant groups
    group_choices = []
    for group_option in
Group.objects.filter(link__user_id__id=request.user.id):
        group_choices.append((group_option.id, group_option.name))
    self.fields['group'].widget.choices = group_choices
```

This code works, however it does not state the year group along with the name of the class.

To add the year group field, this must first be added to the student model.

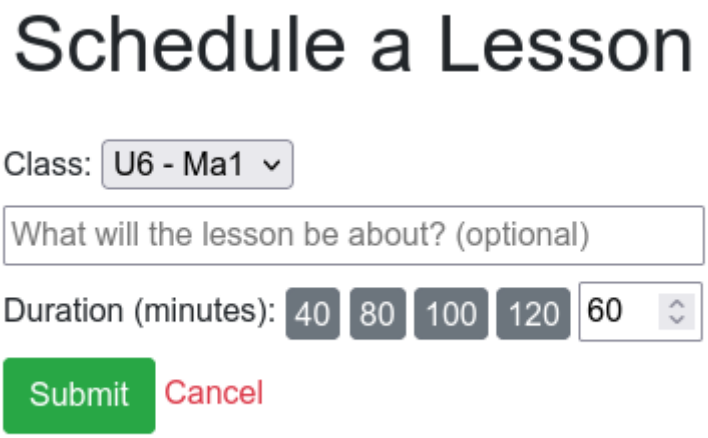
```
year_group = models.CharField(max_length=4, null=True, default=None)
```

This was then added to the user creation / modification forms.

Then, this could be queried within `__init__()`

```
example_student: User =
User.objects.filter(link__user_id__id=request.user.id,
user_type='student')[:1].get()
if example_student:
    text = f"{example_student.year_group} - {group_option.name}"
else:
    text = group_option.name
group_choices.append((group_option.id, text))
```

After assigning a year group to both of the test students, the groups displayed correctly.



Some formatting was also updated to include a colon (and shrink the size of the duration box accordingly) and a clearer note of what the field is for.

4.1.8.2.5.3 Processing the Result

Processing the data was made significantly more difficult by the group dropdown box as the input has to be processed as a group object so that each group could be easily identified. The way fields were processed, sending the primary key, was re-done. The new method:

- Includes server-side validation
- Is easier to read

However, this renders unhelpful options such as `Group object (1)` which give no indication to the user about which class they are actually choosing. To fix this, a subclass of the `ModelChoiceField` had to be created to override the way it was rendered in the dropdown list.

This could then have the function `label_from_instance` which takes in the object and outputs the label for it. Note that year group is only stored alongside students, so must be queried as such. Code was added in case that the class is empty and therefore no year group is present – it will simply display the class name instead of producing an error.

`fields.py`

```
class CustomModelChoiceField(ModelChoiceField):
    def label_from_instance(self, obj):
        example_student =
User.objects.filter(link__group_id__id__exact=obj.id,
user_type='student')[:1]
        if example_student:
            example_student = example_student.get()
        if example_student and example_student.year_group:
            text = f"{example_student.year_group} - {obj.name}"
        else:
            text = obj.name
        return text
```

Then, the group field could be defined as shown:

```
group = CustomModelChoiceField(widget=forms.Select(
    attrs={
        'id': 'group-select'
    }),
    label='Class:', required=True,
    queryset=Group.objects.filter(link__user_id__id__exact=None),
    to_field_name='name', empty_label=None)
```

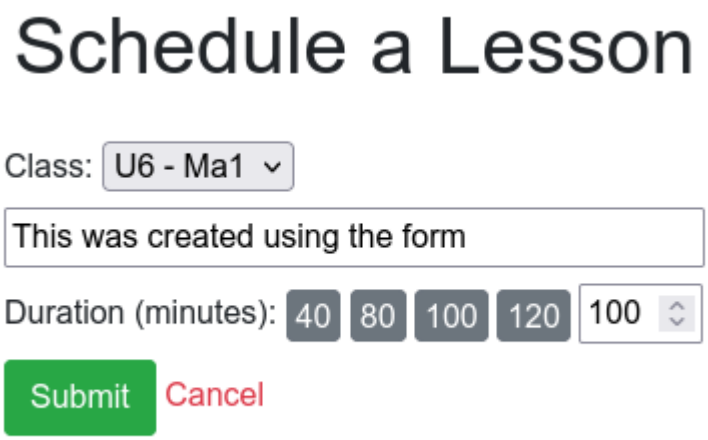
Finally, the `queryset` was updated after initialisation to include the logged in teacher.

```
self.fields['group'].queryset =
Group.objects.filter(link__user_id__id__exact=self.request.user.id)
```

This had to be updated after initialisation because the logged in user is not known beforehand and depends on each form.

4.1.8.2.6 Testing

The following test data was input:



Clicking `Submit` correctly redirected to the list of scheduled lessons (although this has not yet been implemented)

4.1.8.3 Listing Scheduled Lessons

4.1.8.3.1 Breaking down the problem

This should:

- Retrieve the (relevant) lessons from the database and pass them into the template
- The template should then render them on the page
    - Lessons that are fixed should be displayed as such
    - Lessons in the past should not be displayed at all

- A warning should display if not enough lessons are scheduled
    - The minimum number of unscheduled lessons should be implemented as a constant which can be changed later.

4.1.8.3.2 Retrieving the Lessons

The lessons should be retrieved in a similar manner to when viewing the timetable:

(part of) `views.py`

```
@login_required
def teacher_scheduled(request):
    user = request.user

    lessons = []
    unscheduled = 0
    for lesson in
Lesson.objects.filter(group_id__link__user_id__username__exact=user.usernam
e, start__gte=datetime.datetime.now()):

        hours = math.floor(lesson.duration / 3600)
        minutes = math.floor((lesson.duration - hours) / 60)

        if lesson.topic:
            topic = lesson.topic
        else:
            topic = '[untitled]' # this makes more sense than an empty
string

        if not lesson.fixed:
            unscheduled += 1

        lessons.append({'topic': topic, 'duration': str(hours)+'h
'+str(minutes)+'m', 'fixed': lesson.fixed})

    return render(request, 'timetable/scheduling/scheduled_list.html',
{'lessons': lessons, 'enough': unscheduled >= MIN_UNSCHEDULED_LESSONS})
```

4.1.8.3.3 Rendering within the Template

Lessons can be added to the list with a simple `for` loop. As for style, a simple condition checks if it is fixed or not and assign classes appropriately. The warning displays if there are not enough lessons, which is determined in the Python code and passed in via the Boolean variable `enough`.

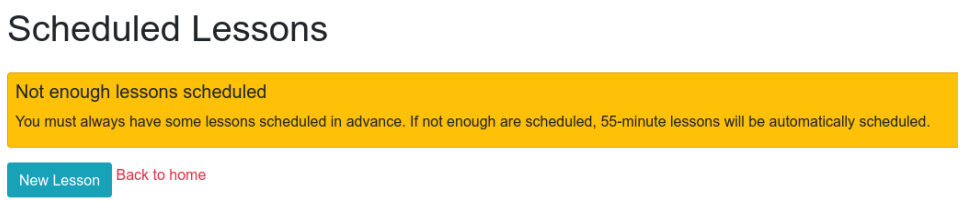
`scheduled_list.html`

```
{% extends "timetable/base.html" %}
{% block content %}
    <h1 class="m-4">Scheduled Lessons</h1>
    {% for lesson in lessons %}
    <div class="card my-2 mx-4 {% if lesson.fixed %}scheduled{% else %}bg-
light{% endif %}">
        <div class="card-body">
```

```
            <p class="float-left mb-0">{{lesson.topic}}</p>
            <p class="float-right mb-0">{{lesson.duration}}</p>
        </div>
    </div>
    {% endfor %}
    {% if not enough %}
    <div class="card my-2 mx-4 bg-warning">
        <h5 class="mx-2 mt-2">Not enough lessons scheduled</h5>
        <p class="mx-2">Please ensure you always have enough lessons
scheduled in advance. If you do not schedule enough lessons, 55-minute
lessons will be automatically generated.</p>
    </div>
    {% endif %}
    <a href="/teacher/schedule"><button class="btn btn-info ml-4 mt-2">New
Lesson</button></a>
    <a href="/" class="text-danger">Back to home</a>
{% endblock %}
```

4.1.8.3.4 Testing

As expected, after correcting a few typos, running the code produced a list of no lessons and a warning that there are not enough. There are lessons in the database but these are in the past, so it was expected that the program did not display these lessons.



To fully test the functionality, some of the lessons which were previously created can have their start date changed so that they appear on this interface, and potentially have their `fixed` field changed.

Running this made apparent that the lesson duration was not of type `float`, but instead of type `datetime.timedelta`. This could easily be converted by accessing the `seconds` attribute

```
hours = math.floor(lesson.duration.seconds / 3600)
minutes = math.floor((lesson.duration.seconds - hours) / 60)
```

It then gave the output:

This result made it apparent that the order of lessons was not properly being maintained. This is very important as teachers may not be able to teach the next concept without the knowledge from the previous one. The durations also appeared wrong, however these may not have been properly input

into the lessons. This also omits lessons which have not yet been allocated a start time, which should not be the case.

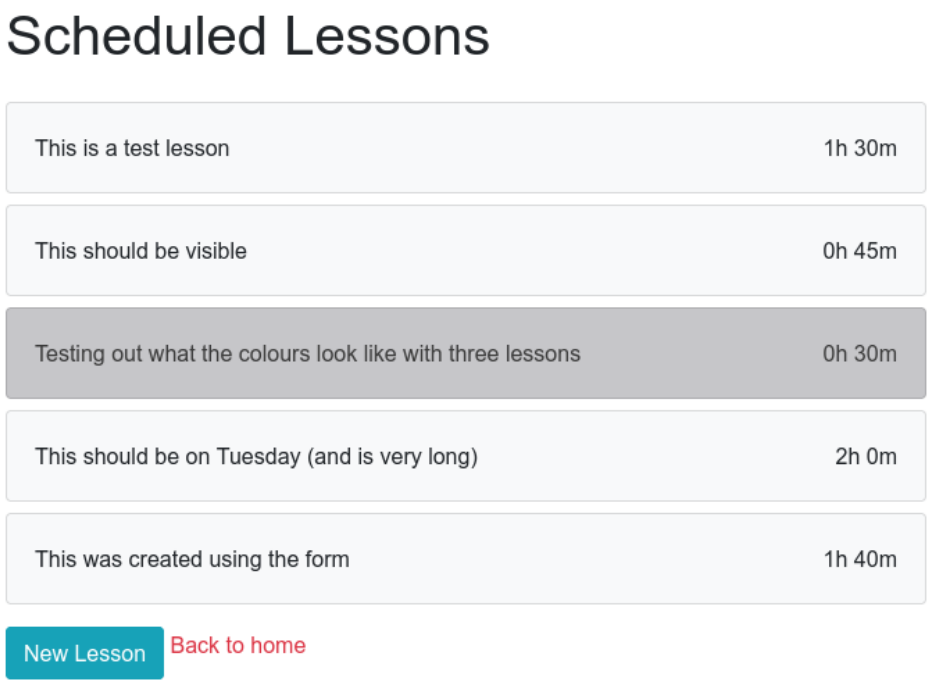
Reversing the query for the start time and instead excluding results made the lesson without a start time appear.

Maintaining the order of the lessons can be done by simply sorting by primary key. This appears to be the order it is returned usually, however this is not guaranteed. The manually input test data was determined to not be valid so this result is actually correct. In a real-world scenario, lessons would be added in the correct order and the fixed / grey lessons would always be at the top.

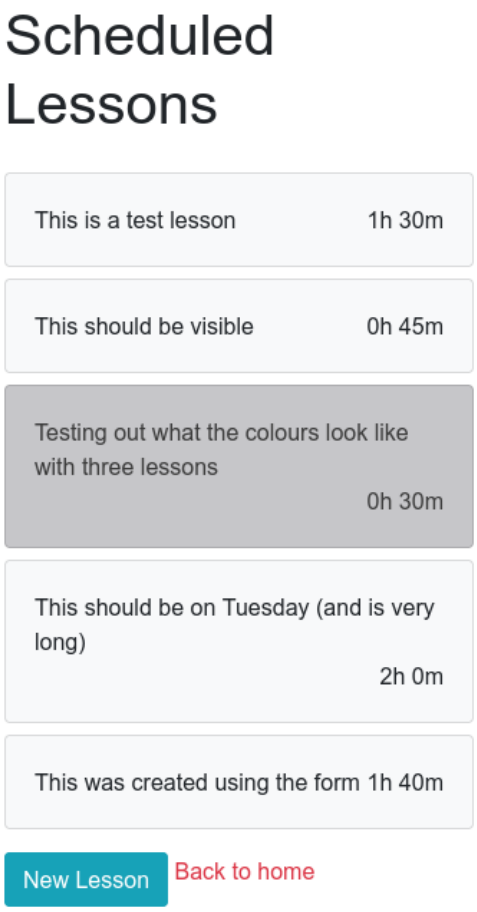
Checking the durations of the lessons shows that this is an issue with how the lessons were input rather than the code itself (some lessons were input as one minute, others 30 or 45 seconds...). Nonetheless, the durations of these lessons were corrected to make the user interface look better. The durations were also formatting wrongly as the 3600 was missing from the line below:

```
minutes = math.floor((lesson.duration.seconds - hours*3600) / 60)
```

These issues have now been corrected and the list of scheduled lessons displays properly:



It also displays well on mobile:



4.1.8.4 Reviewing Lesson Scheduling

Lesson scheduling has passed extensive testing in **4.1.8.2.6** and **4.1.8.3.4**, and all functionality worked as expected when combined together.

However, when navigating the interface, it became clear that there was no option to modify and delete already scheduled lessons. This would be a valuable addition because:

- If teachers input the topic wrong, it may be annoying
- (but more importantly), if a teacher inputs a lesson with the wrong duration, they are then forced to teach a lesson with a suboptimal duration
    - This will be frustrating and works in contradiction to the benefits gained by the variable lesson lengths

While this will improve the product, it is not required functionality so will be delayed until after the timetabling algorithm is finished.

NOTE: This feature was not pursued further as part of the project. More details are given in the evaluation phase.

4.2 Timetabling Algorithm

The timetabling algorithm is made up of three separate parts that can be developed independently:

- The heuristic algorithm to minimise the value of the objective function
- The objective function itself
- The code that calls this function to schedule the algorithms

This algorithm has been designed in detail in **3.4**

4.2.1 Implementing a Genetic Algorithm

The genetic algorithm requires the following key functions:

- Generating a random timetable
- The cost / objective / fitness function
- Parents selection
- Offspring generation / crossover
- Mutation
- New population selection

The code for this section is available at the end, in **4.2.1.9**

4.2.1.1 Defining the Classes

The classes were designed in **3.6.2**

The two classes needed are:

- A class storing the full population of solutions
    - It will include methods relating to the genetic algorithm
- A class storing each individual solution
    - It will include methods that operate on a single solution, including the cost function

4.2.1.1.1 Population Class

This class represents a population of solutions. It has been created as designed in **3.6.2.2**

It takes in key values relating to the genetic algorithm such as:

- Size of population
- Number of parents
- Stopping condition

It also takes in configuration variables to pass into the Timetable classes used throughout the algorithm

Some additional validation was added as defensive programming:

- If the number of parents is more than the size of the population, this is not valid
- If the guaranteed surviving parents is more than the number of parents, this is also not valid

- The population size cannot be negative (or equal to 0)

The constructor for the class is as follows:

```
class Population:
    """Represents a population of timetables for use in the genetic
algorithm"""

    def __init__(self, timetable_init_kwargs,
                 popsize=100, num_parents=50,
                 mutation_amount=2, guaranteed_parent_survival=5,
                 stopping_condition=should_stop,
                 first_day: Optional[datetime.datetime] = None, days: int =
1,
                 time_per_day: int = 114, seconds_per_unit_time: float =
300,
                 day_start=datetime.time(8, 30, 0)):
        """
        :param popsize: The population size
        :param stopping_condition: A function taking in:
         - the current population (of type Population)
         - previous population (THIS WILL BE NONE ON THE FIRST ITERATION)
         - the number of iterations / generations
         and returning True to stop and False to continue
        """

        if first_day:
            self.first_day = first_day
        else:
            self.first_day = datetime.datetime.now()
        self.days = days
        self.time_per_day = time_per_day
        self.seconds_per_unit_time = seconds_per_unit_time
        self.day_start = day_start

        self.popsize = popsize
        self.num_parents = num_parents
        self.guaranteed_surviving_parents = guaranteed_parent_survival
        self.stopping_condition = stopping_condition
        self.generations = 0
        self.mutation_amount = mutation_amount

        if self.num_parents > self.popsize:
            raise ValueError("Number of parents cannot be greater than size
of population")
        if self.guaranteed_surviving_parents > self.num_parents:
            raise ValueError("Surviving parents cannot be more than the
number of parents")
        if self.popsize < 1:
            raise ValueError("Population must be positive")

        self.population: list[tuple[Timetable, float]] = []  # (timetable,
cost)
        for x in range(self.popsize):
```

```
self.population.append((Timetable(**timetable_init_kwargs).random(),
float('inf')))
```

The methods are detailed throughout the section and are also available at the end.

4.2.1.1.2 Timetable Class

The timetable should first be created as blank. Lessons can be added to the timetable through other methods.

The `__init__()` function takes in configuration variables such as:

- Days (that this timetable refers to)
- Time per day (in time units)
- Seconds per time unit
- Start time of each day

These must be assigned to attributes of the class

A timetable will be represented simply as a list of lessons. This is the simplest and most flexible approach which makes timetabling easy.

The constructor is as follows:

```
class Timetable:
    """Represents a potential timetable for a given period of time"""

    def __init__(self, first_day: Optional[datetime.datetime] = None, days:
int = 1, time_per_day: int = 114,
                 seconds_per_unit_time: float = 300,
day_start=datetime.time(8, 30, 0), year_start=None,
                 unscheduled_lessons=None, group_data=None,
desired_allocations=None, lessons=None):
        if first_day:
            self.first_day = first_day
        else:
            self.first_day = datetime.datetime.now()
        self.days = days
        self.time_per_day = time_per_day
        self.seconds_per_unit_time = seconds_per_unit_time
        self.day_start = day_start
        self.desired_allocations = desired_allocations

        if year_start:
            self.year_start = year_start
        else:
            first_lesson = Lesson.objects.order_by('start')[:1].get()
            self.year_start = first_lesson.start

        if unscheduled_lessons:
            self.unscheduled_lessons = unscheduled_lessons
            random.shuffle(self.unscheduled_lessons)
        else:
```

```
            self.unscheduled_lessons = []
            per_class = {}
            for unscheduled_lesson in
Lesson.objects.filter(fixed=False).exclude(start__lte=first_day):
                if unscheduled_lesson.group in per_class:
                    per_class[unscheduled_lesson.group] += 1
                else:
                    per_class[unscheduled_lesson.group] = 1

                if per_class[unscheduled_lesson.group] <= days:  # this
helps to reduce the possibilities to consider
                    self.unscheduled_lessons.append(unscheduled_lesson)
            random.shuffle(self.unscheduled_lessons)

        if group_data:
            self.group_data = group_data
        else:
            self.group_data = {}
            for group in Group.objects.filter():
                previous = None
                time_allocated = 0
                for lesson in
Lesson.objects.filter(group_id__id__exact=group.id,
start__lte=datetime.datetime.now()).order_by('start'):  # oldest first
                    previous = lesson
                    time_allocated += lesson.duration
                days_since_previous =
(datetime.datetime.now().replace(hour=0, minute=0, second=0) -
previous.start.replace(hour=0, minute=0, second=0)).days
                self.group_data[group.id] = [time_allocated,
days_since_previous]

        if not lessons:
            self.lessons = {}  # format {day: lesson} of type {int: Lesson}
```

As with the `Population` class, the methods belonging to this class are detailed throughout the section. The full code is available at the end of the section.

4.2.1.2 Generating a Random Timetable

4.2.1.2.1 Introduction

Generating a random timetable was more difficult than originally anticipated. While it would be possible to schedule lessons at random times, it would not be appropriate to schedule every single lesson into the day as this would product many clashes and would never form a feasible solution.

The goal of this algorithm is to find a potentially feasible solution in a very short amount of time that can be iterated on. The algorithm should product a wide variety of outputs, including the optimal solution (however this is extremely unlikely).

As with all parts of the algorithm, it is a tradeoff between finding a good solution and doing it quickly. While a better solution could be found if more time was dedicated, this would ultimately

come at the cost of further iterations of the algorithm. Equally, the solutions produced by this algorithm must be very random so that there is lots of variation within the population. Without sufficient variation, the algorithm may sometimes complete very competently (if the randomly generated solutions are good) however other times will fail (if all the randomly generated solutions don't happen to be very good), whereas it should consistently produce a good solution.

4.2.1.2.2 Considering the Traditional Method

To consider how to timetable for the new dynamic case, the traditional static case was considered. In the static case, it is considerably easier as there are a fixed number of lessons that need to be allocated to a fixed time period. The whole timetable can be considered at once. Unfortunately, this is not available for the dynamic case. However, inspiration can be taken from the traditional method. As with the traditional method, the algorithm could attempt to fill up available slots.

4.2.1.2.3 First Iteration

The first iteration of the algorithm will be as follows:

- For each (random) teacher:
  - For each unscheduled (random) lesson:
    - Attempt to schedule at a random point on the day where it does not clash with another lesson for this teacher
    - If there is no time for the lesson, move onto the next teacher

Part of the initial program for choosing a random teacher was as follows:

```
for day in range(self.days):
    dt = self.first_day + datetime.timedelta(days=day)
    after = dt.replace(day=0, minute=0, hour=0)  # this should be 0 anyway
if first_day is actually a day
    before = dt.replace(day=23, hour=59, minute=59, second=59)
    for teacher in
User.objects.filter(user_type__exact='teacher').order_by('?'):
        # for each (randomly ordered) lesson that involves this teacher
        for lesson in
Lesson.objects.filter(group_id__link__user_id__id__exact=teacher.id,
start__gte=after, start__lte=before).order_by('?'):
```

4.2.1.2.4 Issues with this Algorithm

However, this had multiple problems:

- Every random lesson required querying the database directly
  - This is slower
  - The function should be abstracted from the specific database implementation for modularity and procedural abstraction.
- Lessons were returned out of order (they must be taught in the same order that the teacher added them to the database)

4.2.1.2.5 Solving the First Problem

The first problem could be solved by passing in a parameter `self.unscheduled_lessons`, which would then be used to get the lessons from.

This parameter can be generated as follows:

```
self.unscheduled_lessons = []
per_class = {}
for unscheduled_lesson in
Lesson.objects.filter(fixed=False).exclude(start__lte=datetime.datetime.now
()):
    if unscheduled_lesson.group in per_class:
        per_class[unscheduled_lesson.group] += 1
    else:
        per_class[unscheduled_lesson.group] = 1

    if per_class[unscheduled_lesson.group] <= days:  # this helps to reduce
the possibilities to consider
        self.unscheduled_lessons.append(unscheduled_lesson)
random.shuffle(self.unscheduled_lessons)
```

It can then be generated once and passed into all timetables in the population:

```
self.unscheduled_lessons = unscheduled_lessons
random.shuffle(self.unscheduled_lessons)
```

The algorithm for generating a random timetable would differ after this change:

- Choose a random lesson
    - Attempt to schedule at a random gap in the teacher's timetable
    - If there are no gaps with enough space for the new lesson, increment a counter
    - If this counter exceeds a fixed threshold, or all the lessons are scheduled, the timetable is complete

This can be implemented as follows:

```
counter = 0
for lesson in self.unscheduled_lessons:
    lesson:PotentiallyScheduledLesson
    teacher = self.get_teacher(lesson.id)
    day = random.randint(0, self.days-1)
    gaps = self.get_gaps(user_id=teacher.id, days=[day], random_order=True)

    for gap_start, gap in gaps:  # for each (random) gap,
        if gap > lesson.duration + 1:  # if there's enough space for a
lesson (need at least 1 unit either side)
            if gap < lesson.duration * 1.5:  # if there's not much space...
                lesson.start = gap_start + 1  # schedule for start of gap
(plus 1 unit break)
            else:
                latest_end = gap_start + gap - 2
                lesson.start = random.randint(gap_start, latest_end -
```

```
lesson.duration)  # allocate to random position
            self.lessons[day] = lesson
            break
        else:
            continue
    else:  # this triggers if the end of the loop is reached without a
break statement
        counter += 1
        if counter > threshold:
            break
```

where `threshold` is a parameter with default value `100`

While it was intended for this to be fully abstracted from the database, a `get_teacher()` method was added which has to use the database to retrieve the relevant teacher for each lesson

```
def get_teacher(self, lesson_id):
    return User.objects.filter(link__class_id__lesson__id__exact=lesson_id,
user_type__exact='teacher')[:1].get()
```

However, this was not actually necessary. The teachers could all be found in advanced and added to the lesson objects in `self.unscheduled_lessons`. This could either be done initially, or as and when the teacher is required. The second approach would be favourable as:

- It reduces the amount of database access required
- The operations are not time sensitive
    - there is no difference between using 1 minute of time initially and using 1 minute during processing
        - unlike a search engine, for example

This leads to an updated `get_teacher()` method which takes in the full lesson object rather than the `lesson_id`

```
def get_teacher(self, lesson):
    """Gets a teacher who teaches the given lesson
    Teachers are cached, so the cached version will be returned upon any
future calls"""
    if not hasattr(lesson, 'teacher'):
        lesson.teacher =
User.objects.filter(link__class_id__lesson__id__exact=lesson.id,
user_type__exact='teacher')[:1].get()
    return lesson.teacher
```

4.2.1.2.6 Solving the Second Issue

The second issue, concerning lessons being returned out of order, is more challenging to solve.

The algorithm currently produces a timetable concerning any given number of days. However, it is not yet known how many days will need to be timetabled at once and it is likely that only a single day will be timetabled at once. This issue is not present when only one day is timetabled at once. This will be experimented with further while implementing the algorithm within the wider project.

4.2.1.3 Implementing the Cost Function

The cost function was defined in **3.4.2.2**.

Relevant extracts from this table are provided where appropriate.

4.2.1.3.1 Constraints 1 & 2: Clashes

#	Constraint	Cost Value
1	Teacher clashes	n = number of teacher clashes k = points per teacher clash = 10000 $C_1(n) = kn$
2	Student clashes	n = number of student clashes k = points per student clash = 100 $C_2(n) = kn$

Designing this algorithm presented a tradeoff between space complexity and time complexity. An algorithm with a high space complexity could store a lookup table of every lesson in an easily accessible format which would be quickly accessible. Other approaches would take longer but would be able to find a solution without creating a large data structure that would use lots of memory.

As the cost function is being evaluated frequently, it is critical that it is as quick as possible. The memory will only be used for each execution of the cost function and is cleared afterwards. There should not be an issue with insufficient memory, especially as it was stated the project needs at least 2 GB of RAM.

Timetables are stored as a list of lessons, therefore the only sensible approach is to iterate over every lesson. Then, it should store, for every user, which time slots are busy and which time slots are free.

It became clear while developing this that the initial definition of the objective function, using a fixed number of points for each clash, was suboptimal. It was changed in the implementation to use a fixed number of points per time unit per clash, so that if more lesson time clashed then more points would be added, scaling proportionally. The initial values for the constants were divided by 10 to accommodate for this change.

After this changed, a fixed number of points could be added for every time slot that clashes (upon attempting to add a time slot, but noticing that it is already in the list)

```
# constraints 1 & 2: clashes
teacher_clashes = 0
student_clashes = 0
schedules = {}  # {user_id: [time_slot]}
for lesson in self.lessons:
    for user in lesson.get_users():
        clashes = 0
        if user.id not in schedules:
            schedules[user.id] = []
        for x in range(lesson.duration):
            time_slot = lesson.start + x
            if time_slot in schedules[user.id]:
```

```
                clashes += 1
            else:
                schedules[user.id].append(time_slot)
        if user.user_type == 'teacher':
            teacher_clashes += clashes
        else:
            student_clashes += clashes
clashes_cost = POINTS_PER_STUDENT_CLASH * student_clashes +
POINTS_PER_TEACHER_CLASH * teacher_clashes
```

4.2.1.3.2 Constraints 3, 4, 5

3	Even allocation of lesson times	$$C_3(x,s,t) = \sum_{i=0}^{n} k\, f(x_i, s_i)\, w(t)$$ $$f(x,s) = \frac{\lvert x-s \rvert}{s}$$ $$w(t) = sig(a+bt)$$
4	Consecutive lessons gap	n = number of occurrences k = points per occurrence = 10 $$C_4(n) = kn$$
5	Variety of subjects	$$C_5(\dots) = \sum_{t=t_0}^{T+t_0} \sum_{i=0}^{N-1} (f(n_{i,t}) + g(d_{i,t}))$$ $$f(n) = max\,\{k(n-1), 0\}$$ $$g(t) = ka^t$$

Constraints 3 and 5 require information from the past timetable. They are explored later in this section.

Constraint 4 is computed with constraint 7

4.2.1.3.3 Constraint 6: Max Daily Workload

6	Maximum daily workload	$x = x_0, x_1, x_2, \dots, x_{n-1}, x_n$ = workload of users x_i = daily workload in hours for user *i* n = total number of users k = arbitrary constant = 5 c = arbitrary constant (determines max load) = 4 **NOTE**: max workload = $clnc$ $$C_6(x,n) = \sum_{i=0}^{n-1} f(x_i)$$ $$f(x) = k\, max\,\{e^{\frac{x}{c}} - c, 0\}$$

This dict `schedules` turned out to be very useful for this constraint (and others).

One potential weakness of this format is that when a clash occurs it is only recorded once, rather than once for each occurrence. However, this is entirely appropriate given the situation. If there is a clash, the user will only be able to attend one of their lessons so the other lessons will not contribute to their daily workload.

For each user, the length of the list represents the workload on that day in time units. This could then be implemented into the function detailed earlier. The value of the constant was also adjusted as time is measured in time units rather than in hours.

```
# constraint 6: max daily workload
daily_workload_cost = 0
for user_id in schedules:
    # NOTE: It is assumed that if any clashes occur, the user will miss out
on
    # all but one of the concurrent events, so it does not contribute to
their overall workload
    daily_workload_cost +=
max(math.exp(len(schedules[user_id])/MAX_LOAD_CONSTANT)-MAX_LOAD_CONSTANT,
0)
```

4.2.1.3.4 Constraint 7: Gaps

7	Gaps	Explored in **3.4.2.2.3** $x_{i,t} = x_{i,t,0}, x_{i,t,1}, \ldots, x_{i,t,n}$ = length of each gap for student *i* on day *t* k = arbitrary constant t_0 = start day; T = number of days n = number of students; $N_{i,t}$ = number of gaps student *i* day *t* a,b,c,d = arbitrary constants; a = 10; b = 5; c = 2; d = 1 $$C_7(x) = \frac{1}{k}\sum_{t=t_0}^{T-t_0}\sum_{i=0}^{n}\sum_{j=0}^{N_{i,t}} f(x_{i,t,j})$$ $$f(x) = \{a, \quad x = 0 \quad b, \quad x = 10{,}15\ c, \quad x = 20 \quad \{d, \quad x = 25\ 0, \quad otherwise$$

The `schedules` dict could again be used, but this time simply to get a list of users. For each user, their gaps could be calculated and then for each gap, points were added accordingly.

```
# constraint 7: gaps
gaps_cost = 0
for user_id in schedules:  # NOTE: schedules is only used for a list of
user ids
    gaps = self.get_gaps(user_id)
    for gap in gaps:
        gaps_cost += self.get_gap_cost(gap)
```

```
def get_gaps(self, user_id, days=None, random_order=False) ->
List[Tuple[int, int]]:
    """Gets a list of the duration of every gap on the given day, in time
units

    Days should be a list of numbers, each indicating the number of days
since first_day

    Returns dict of form [(start of gap, length of gap)] of type [(int,
int)]"""
    if not days:
        days = range(self.days)

    previous = None
    gaps = []
    for day in days:
        sorted(self.lessons[day])
        for lesson in self.lessons[day]:
            if previous:
                gaps.append((previous, lesson.start - previous))
            previous = lesson.start

    if random_order:
        random.shuffle(gaps)

    return gaps

def get_gap_cost(self, gap_length):
    """Returns the value to add to the cost function for a gap of length
gap_length, in time units
    Gaps should never be negative, but if so this function will return 0"""
    if gap_length == 0:
        return 10
    # 1 is the perfect length -> 0 is returned
    elif gap_length in (2, 3):
        return 5
    elif gap_length == 3:
        return 2
    elif gap_length == 4:
        return 1
    else:
        return 0
```

4.2.1.3.5 Constraint 8: Early Finish Time

8	Early Finish	t_i = number of hours user *i* has to be in school; n = number of users k = arbitrary constant = 10 $$C_8(t) = \frac{1}{k}\sum_{i=0}^{n-1} t_i$$

The `schedules` dict was also used for this to retrieve the latest occupied time slot on any given user's day and add points accordingly.

```
early_finish_cost = 0
for user_id in schedules:
    finish_time = max(schedules[user_id])
    early_finish_cost += finish_time / EARLY_FINISH_CONSTANT
```

4.2.1.3.6 Retrieving Past Data

Constraints 3 and 5 both require information on the past timetable.

The following values need computing for each class/group:

- Time allocated (over the course of a year)
    - Constraint **3**
- Time since last lesson (in days)
    - Constraint **5**

Calculating these values each time the cost function is called would be very inefficient. As these values will not change throughout a given execution of the algorithm, it would be appropriate to pre-compute the values at the start of the iteration. They can then be accessed quickly in each generation of the algorithm

These can be represented as a `dict` and stored separately to the lessons

```
if group_data:
    self.group_data = group_data
else:
    self.group_data = {}
    for group in Group.objects.filter():
        previous = None
        time_allocated = 0
        for lesson in Lesson.objects.filter(group_id__id__exact=group.id,
start__lte=datetime.datetime.now()).order_by('start'):  # oldest first
            previous = lesson
            time_allocated += lesson.duration
        days_since_previous = (datetime.datetime.now().replace(hour=0,
minute=0, second=0) - previous.start.replace(hour=0, minute=0,
second=0)).days
        self.group_data[group.id] = [time_allocated, days_since_previous]
```

4.2.1.3.7 Constraint 3: Even Allocation

3	Even allocation of lesson times	$x = x_1, x_2, \ldots, x_n$ = lesson time allocated to each class so far $s = s_1, s_2, \ldots, s_n$ = lesson time intended for each class n = number of classes t = school days elapsed since start of year a = arbitrary constant = 0.4 b = arbitrary constant = 8 **NOTE**: $\frac{b}{a}$ gives the value of t where $w(t) = 0.5$

		k = weighting of a 100% difference = 100 $$C_3(x,s,t) = \sum_{i=0}^{n} k\, f(x_i, s_i)\, w(t)$$ $$f(x,s) = \frac{\lvert x-s \rvert}{s}$$ $$w(t) = sig(a+bt)$$

The weighting does not depend on the group / class, so could be calculated once outside of the loop.

$$C_3(x,s,t) = k\, w(t) \sum_{i=0}^{n} f(x_i, s_i)$$

The iterative part of the function was created with no issues.

However, the weighting depended on the value for t which had not yet been calculated (t = number of school days elapsed since the start of the year). This could be determined by retrieving the oldest lesson's start time.

```
if year_start:
    self.year_start = year_start
else:
    first_lesson = Lesson.objects.order_by('start')[:1].get()
    self.year_start = first_lesson.start
```

Constraint 3 was implemented as shown below:

```
def sigmoid(a):
    return 1 / (1 + math.exp(-a))
weighting = sigmoid(EVEN_ALLOCATION_CONSTANT_A + EVEN_ALLOCATION_CONSTANT_B
* (self.first_day - self.year_start).days)
diffs = 0
for group_id in self.group_data:
    diff = abs(self.group_data[group_id][0] -
self.desired_allocations[group_id]) / self.desired_allocations[group_id]
    diffs += diff
even_allocation_cost = WEIGHTING_OF_DIFF * weighting * diffs
```

NOTE: `self.desired_allocations` will be set up later

A separate function was used for the sigmoid so it is separate from the rest of the code.

4.2.1.3.8 Constraint 5: Variety

5	Variety	$n_{i,t}$ = number of lessons with class *i* on day *t*; N = number of classes t_0 = start day; T = number of days $d_{i,t}$ = days between the lesson on day t and the next lesson a = arbitrary constant $$C_5(\dots) = \sum_{t=t_0}^{T+t_0} \sum_{i=0}^{N-1} \left(f(n_{i,t}) + g(d_{i,t})\right)$$ $$f(n) = max\,\{k(n-1), 0\}$$

		$g(t) = ka^t$

Part of constraint 5 involves adding points if a lesson has multiple occurrences on the same day. If only one day is calculated at once, this is prevented by the timetable. The second part could be implemented through a simple exponential model as shown:

```
variety_cost = 0
for group_id in self.group_data:
    variety_cost += VARIETY_COEFFICIENT * (VARIETY_BASE **
self.group_data[group_id][1])
```

4.2.1.3.9 Final cost function

It is quickly becoming apparent that this cost function is very expensive to compute. While it may be possible to optimise this further, some other aspects of the algorithm may need be tweaked to try and reduce the number of times it is called and increase the effectiveness of each generation.

The full cost function for use in testing is as follows:

[Jump to bottom: **4.2.1.4**]

```
def cost(self):
    """Evaluates the cost function for the current solution"""
    POINTS_PER_TEACHER_CLASH = 1000
    POINTS_PER_STUDENT_CLASH = 10
    MAX_LOAD_CONSTANT = 23  # max load = c ln c, where c is this value
    EARLY_FINISH_CONSTANT = 10
    EVEN_ALLOCATION_CONSTANT_A = 0.4
    EVEN_ALLOCATION_CONSTANT_B = 8
    WEIGHTING_OF_DIFF = 100
    VARIETY_BASE = 2
    VARIETY_COEFFICIENT = 1

    total_cost = 0
    for day in range(self.days):
        # constraints 1 & 2: clashes
        teacher_clashes = 0
        student_clashes = 0
        schedules = {}  # {user_id: [time_slot]}
        for lesson in self.lessons:
            for user in lesson.get_users():
                clashes = 0
                if user.id not in schedules:
                    schedules[user.id] = []
                for x in range(lesson.duration):
                    time_slot = lesson.start + x
                    if time_slot in schedules[user.id]:
                        clashes += 1
                    else:
                        schedules[user.id].append(time_slot)
                if user.user_type == 'teacher':
```

```
                    teacher_clashes += clashes
                else:
                    student_clashes += clashes
        clashes_cost = POINTS_PER_STUDENT_CLASH * student_clashes +
POINTS_PER_TEACHER_CLASH * teacher_clashes

        # constraint 3: even allocation of lesson times
        def sigmoid(a):
            return 1 / (1 + math.exp(-a))
        weighting = sigmoid(EVEN_ALLOCATION_CONSTANT_A +
EVEN_ALLOCATION_CONSTANT_B * (self.first_day - self.year_start).days)
        diffs = 0
        for group_id in self.group_data:
            diff = abs(self.group_data[group_id][0] -
self.desired_allocations[group_id]) / self.desired_allocations[group_id]
            diffs += diff
        even_allocation_cost = WEIGHTING_OF_DIFF * weighting * diffs

        # NOTE: constraint 4 is being computed with constraint 7

        # constraint 5: variety of subjects
        variety_cost = 0
        for group_id in self.group_data:
            variety_cost += VARIETY_COEFFICIENT * (VARIETY_BASE **
self.group_data[group_id][1])

        # constraint 6: max daily workload
        daily_workload_cost = 0
        for user_id in schedules:
            # NOTE: It is assumed that if any clashes occur, the user will
miss out on
            # all but one of the concurrent events, so it does not
contribute to their overall workload
            daily_workload_cost +=
max(math.exp(len(schedules[user_id])/MAX_LOAD_CONSTANT)-MAX_LOAD_CONSTANT,
0)

        # constraint 7: gaps
        gaps_cost = 0
        for user_id in schedules:  # NOTE: schedules is only used for a
list of user ids
            gaps = self.get_gaps(user_id)
            for gap in gaps:
                gaps_cost += self.get_gap_cost(gap)

        # constraint 8: early finish time
        early_finish_cost = 0
        for user_id in schedules:
            finish_time = max(schedules[user_id])
            early_finish_cost += finish_time / EARLY_FINISH_CONSTANT

        debug_info = {
            'teacher clashes': teacher_clashes,
            'student clashes': student_clashes,
```

```
            'clashes cost': clashes_cost,
            'total diffs': diffs,
            'even allocation cost': even_allocation_cost,
            'variety cost': variety_cost,
            'daily workload cost': daily_workload_cost,
            'gaps cost': gaps_cost,
            'early finish cost': early_finish_cost
        }

        total_cost += clashes_cost + even_allocation_cost + variety_cost +
daily_workload_cost + gaps_cost + early_finish_cost

    return total_cost
```

4.2.1.4 Parents Selection

Parents selection should be done after the cost function has been computed for every parent. It can be done by simply choosing the parents with the lowest cost function.

```
def choose_parents(self):
    """Chooses the best parents from the population to mate"""

    candidates = self.population
    parents = []

    # choose the best parents
    for x in range(self.num_parents):
        best_parent = min(candidates, key=lambda t, c: c)
        parents.append(best_parent)
        candidates.remove(best_parent)

    return parents
```

It would be possible to sort the parents by cost value, however this is unnecessarily expensive. Sorting runs in at least $O(n \log n)$ whereas this algorithm runs in $O(n)$

4.2.1.5 Offspring Generation

NOTE: `num_offspring` has been changed post-testing.

To generate the offspring, parents should be randomly chosen until enough offspring have been generated. Many approaches would be valid for this, however an entirely random approach was chosen as it will ensure there is lots of variation within the population.

Note that a list of parents have already been selected – it is not choosing randomly from the whole population.

```
def generate_offspring(self, parents):
    """Generates all the offspring"""
```

```
    num_offspring = self.popsize - self.num_parents
    offspring = []

    for x in range(num_offspring):
        parent1 = random.choice(parents)
        parent2 = random.choice(parents)
        offspring.append(self.crossover(parent1, parent2))

    return offspring
```

Note that each parent could be chosen more than once.

The crossover function must produce a timetable containing (approximately) half of the information from each parent. As a timetable is defined by its lessons, this means that half of the lessons should come from each parent.

```
def crossover(self, parent1, parent2):
    """Returns one offspring containing half information from each
parent"""

    new_lessons = []
    for day in range(self.days):
        potential_new_lessons = parent1.lessons[day] + parent2.lessons[day]
        random.shuffle(potential_new_lessons)
        for x in range(len(potential_new_lessons)//2):
            new_lessons.append(potential_new_lessons.pop(0))

    timetable = Timetable(lessons=new_lessons, first_day=self.first_day,
days=self.days,
                          time_per_day=self.time_per_day,
seconds_per_unit_time=self.seconds_per_unit_time,
                          day_start=self.day_start)
    return timetable, float('inf')  # the cost function may not be needed,
so it does not need to be executed here
```

4.2.1.6 Mutation

Mutation is essential to keep the population changing. Without mutation, species do not evolve as they do not vary. Mutations must only change an individual slightly, as changing it too much could spoil a good solution. If individuals mutate a lot, the algorithm will not manage to converge to a solution.

```
def mutate(self, offspring):
    """Randomly changes the offspring slightly"""

    for timetable in offspring:
        timetable.mutate()

    return offspring
```

The key characteristic for a mutation is that it is random.

Three mutations were used in the implementation:

- Changing the start time of a random lesson
    - This helps move lessons around the day in search for better solutions
- Deleting a random lesson
    - Without deleting any lessons, the number of lessons will stay the same and there will not be enough variation in the population
- Adding a random lesson
    - Without adding any lessons, the size of the population will tend to 0 (as they will all be deleted)

It was decided that two mutations would be performed on each offspring. This value may need to be tweaked. It will likely also depend on the size of the school.

```
def mutate(self, mutate_lessons_per_day=2):
    """Mutates the given solution (for use in a genetic algorithm)
    NOTE: The same lesson could be mutated twice (although unlikely)"""

    for day in range(self.days):
        for x in range(mutate_lessons_per_day):
            n = random.randint(1, 3)
            if n == 1:
                # mutate start time of random lesson
                i = random.randint(0, len(self.lessons[day])-1)
                lesson = self.lessons[day][i]
                latest_time = self.time_per_day - lesson.duration
                lesson.start = random.randint(0, latest_time)
            elif n == 2:
                # delete a random lesson
                self.lessons[day].pop(random.randint(0,
len(self.lessons[day])-1))
            elif n == 3:
                # add a random lesson
                lesson = self.unscheduled_lessons.pop(random.randint(0,
len(self.unscheduled_lessons)-1))
                latest_time = self.time_per_day - lesson.duration
                lesson.start = random.randint(0, latest_time)
                self.lessons[day].append(lesson)

    return self
```

4.2.1.7 New Population Selection

Selecting the new population is important to ensure that the good solutions survive to future generations and the bad solutions do not survive. However, there must still be a probability that a bad solution will remain in the population as otherwise it will converge to only local minima.

Before selecting randomly, a few solutions from the previous generation are carried forward. This ensures that the current best solution does not get any worse between generations.

```
# carry forward the best solutions from the previous generation
for x in range(self.guaranteed_surviving_parents):
    new_population.append(candidates.pop(0))
```

The method for selection will be that in which the probability of not selecting is directly proportional to the value of the cost function (a higher cost value means a lower chance of selection). Note that the cost function does not have to be evaluated for all solutions if these are not chosen randomly, which is particularly helpful given that the cost function is expensive to compute.

```
def choose_new_population(self, candidates):
    """Chooses the new population from the list of candidates
    The value of the cost function should be either the correct value or
infinity if not evaluated"""
    new_population = []
    previous_best_cost = min(candidates, key=lambda t, c: c)[1]

    # carry forward the best solutions from the previous iteration
    for x in range(self.guaranteed_surviving_parents):
        new_population.append(candidates.pop(0))

    # choose the remaining solutions randomly, with a probability
proportional to the cost function
    # sorting the list based on cost would be ideal however is too
expensive
    while len(new_population) < self.popsize:
        i = random.randint(0, len(candidates)-1)
        candidate = candidates[i]

        # evaluate the cost function if not already
        if candidate[1] == float('inf'):
            candidate[1] = candidate[0].cost()

        # choose with probability (roughly) proportional to the value of
the cost function
        #   - this will be incorrect if an uncalculated cost is greater
than that of the previous iteration
        p = candidate[1] / previous_best_cost
        # higher cost is worse, so this gives the probability that it will
fail
        if p < random.uniform(0, 1):
            new_population.append(candidates.pop(i))

    return new_population
```

4.2.1.8 Parallel Processing

While the hardware requirements mention multiple cores, this code is not currently written to take advantage of this. Python, by default, only ever runs in a single CPU thread.

It is possible to parallelise the process by splitting up into multiple populations which each evolve independently. It is not currently known whether this will be necessary to achieve the desired level of performance. This is explored further in **5.5.2.3**.

4.2.1.9 Final Code (pre-testing)

Some aspects of this file have not yet been discussed in this section (however these should only be minor features).

The full contents of `timetabling.py` before any testing is as follows:

[**WARNING**: This code spans **450** lines (**9** pages). Jump to bottom: **4.2.2**]

```
import datetime
import math
import random
from typing import List, Sequence, Optional, Tuple

from src.django_project.timetable.models import Lesson, User, Group


def should_stop(current_population, previous_population, iterations):
    if iterations > 1000:
        return True
    else:
        return False


class Population:
    """Represents a population of timetables for use in the genetic
algorithm"""

    def __init__(self, timetable_init_kwargs,
                 popsize=100, num_parents=50,
                 mutation_amount=2, guaranteed_parent_survival=5,
                 stopping_condition=should_stop,
                 first_day: Optional[datetime.datetime] = None, days: int =
1,
                 time_per_day: int = 114, seconds_per_unit_time: float =
300,
                 day_start=datetime.time(8, 30, 0)):
        """
        :param popsize: The population size
        :param stopping_condition: A function taking in:
         - the current population (of type Population)
         - previous population (THIS WILL BE NONE ON THE FIRST ITERATION)
         - the number of iterations / generations
         and returning True to stop and False to continue
        """

        if first_day:
            self.first_day = first_day
        else:
            self.first_day = datetime.datetime.now()
        self.days = days
        self.time_per_day = time_per_day
        self.seconds_per_unit_time = seconds_per_unit_time
        self.day_start = day_start
```

```
        self.popsize = popsize
        self.num_parents = num_parents
        self.guaranteed_surviving_parents = guaranteed_parent_survival
        self.stopping_condition = stopping_condition
        self.generations = 0
        self.mutation_amount = mutation_amount

        if self.num_parents > self.popsize:
            raise ValueError("Number of parents cannot be greater than size
of population")
        if self.guaranteed_surviving_parents > self.num_parents:
            raise ValueError("Surviving parents cannot be more than the
number of parents")
        if self.popsize < 1:
            raise ValueError("Population must be positive")

        self.population: list[tuple[Timetable, float]] = []  # (timetable,
cost)
        for x in range(self.popsize):

self.population.append((Timetable(**timetable_init_kwargs).random(),
float('inf')))

    def start(self):
        """Iterate over the solution until self.stopping_condition returns
True
         self.stopping_condition should have a fallback condition on the
number of iterations to prevent an infinite loop"""

        previous = None
        self.population = self.evaluate_all_costs(self.population)
        while not self.stopping_condition(self, previous,
self.generations):
            previous = self.copy()
            self.iterate()
            self.generations += 1

        best, best_cost = self.select_best_solution()

        return best

    def iterate(self):
        """Performs one iteration of the genetic algorithm on the current
population"""

        parents = self.choose_parents()
        offspring = self.generate_offspring(parents)
        candidates = parents + offspring
        self.population = self.choose_new_population(candidates)

    def select_best_solution(self):
        """Chooses the best solution, re-evaluating the cost function for
all"""
```

```
        self.population = self.evaluate_all_costs(self.population)
        best = None, float('inf')
        for timetable, cost in self.population:
            if cost < best[1]:
                best = timetable, cost

        return best

    def choose_new_population(self, candidates):
        """Chooses the new population from the list of candidates
        The value of the cost function should be either the correct value
or infinity if not evaluated"""
        new_population = []
        previous_best_cost = min(candidates, key=lambda t, c: c)[1]

        # carry forward the best solutions from the previous iteration
        for x in range(self.guaranteed_surviving_parents):
            new_population.append(candidates.pop(0))

        # choose the remaining solutions randomly, with a probability
proportional to the cost function
        # sorting the list based on cost would be ideal however is too
expensive
        while len(new_population) < self.popsize:
            i = random.randint(0, len(candidates)-1)
            candidate = candidates[i]

            # evaluate the cost function if not already
            if candidate[1] == float('inf'):
                candidate[1] = candidate[0].cost()

            # choose with probability (roughly) proportional to the value
of the cost function
            #   - this will be incorrect if an uncalculated cost is greater
than that of the previous iteration
            p = candidate[1] / previous_best_cost
            # higher cost is worse, so this gives the probability that it
will fail
            if p < random.uniform(0, 1):
                new_population.append(candidates.pop(i))

        return new_population

    def evaluate_all_costs(self, population):
        """Evaluates the cost function for every timetable in the given
population"""

        for i, timetable in enumerate(population):
            timetable, cost = timetable
            timetable: Timetable
            population[i][1] = timetable.cost()

        return population
```

```
    def choose_parents(self):
        """Chooses the best parents from the population to mate"""

        candidates = self.population
        parents = []

        # choose the best parents
        for x in range(self.num_parents):
            best_parent = min(candidates, key=lambda t, c: c)
            parents.append(best_parent)
            candidates.remove(best_parent)

        return parents

    def generate_offspring(self, parents):
        """Generates all the offspring"""

        num_offspring = self.popsize - self.num_parents
        offspring = []

        for x in range(num_offspring):
            parent1 = random.choice(parents)
            parent2 = random.choice(parents)
            offspring.append(self.crossover(parent1, parent2))

        return offspring

    def crossover(self, parent1, parent2):
        """Returns one offspring containing half information from each
parent"""

        new_lessons = []
        for day in range(self.days):
            potential_new_lessons = parent1.lessons[day] +
parent2.lessons[day]
            random.shuffle(potential_new_lessons)
            for x in range(len(potential_new_lessons)//2):
                new_lessons.append(potential_new_lessons.pop(0))

        timetable = Timetable(lessons=new_lessons,
first_day=self.first_day, days=self.days,
                              time_per_day=self.time_per_day,
seconds_per_unit_time=self.seconds_per_unit_time,
                              day_start=self.day_start)
        return timetable, float('inf')  # the cost function may not be
needed, so it does not need to be executed here

    def mutate(self, offspring):
        """Randomly changes the offspring slightly"""

        for timetable in offspring:
            timetable.mutate()

        return offspring
```

```
class Timetable:
    """Represents a potential timetable for a given period of time"""

    def __init__(self, first_day: Optional[datetime.datetime] = None, days:
int = 1, time_per_day: int = 114,
                 seconds_per_unit_time: float = 300,
day_start=datetime.time(8, 30, 0), year_start=None,
                 unscheduled_lessons=None, group_data=None,
desired_allocations=None, lessons=None):
        if first_day:
            self.first_day = first_day
        else:
            self.first_day = datetime.datetime.now()
        self.days = days
        self.time_per_day = time_per_day
        self.seconds_per_unit_time = seconds_per_unit_time
        self.day_start = day_start
        self.desired_allocations = desired_allocations

        if year_start:
            self.year_start = year_start
        else:
            first_lesson = Lesson.objects.order_by('start')[:1].get()
            self.year_start = first_lesson.start

        if unscheduled_lessons:
            self.unscheduled_lessons = unscheduled_lessons
            random.shuffle(self.unscheduled_lessons)
        else:
            self.unscheduled_lessons = []
            per_class = {}
            for unscheduled_lesson in
Lesson.objects.filter(fixed=False).exclude(start__lte=first_day):
                if unscheduled_lesson.group in per_class:
                    per_class[unscheduled_lesson.group] += 1
                else:
                    per_class[unscheduled_lesson.group] = 1

                if per_class[unscheduled_lesson.group] <= days:  # this
helps to reduce the possibilities to consider
                    self.unscheduled_lessons.append(unscheduled_lesson)
            random.shuffle(self.unscheduled_lessons)

        if group_data:
            self.group_data = group_data
        else:
            self.group_data = {}
            for group in Group.objects.filter():
                previous = None
                time_allocated = 0
                for lesson in
Lesson.objects.filter(group_id__id__exact=group.id,
```

```
start__lte=datetime.datetime.now()).order_by('start'):  # oldest first
                    previous = lesson
                    time_allocated += lesson.duration
                days_since_previous =
(datetime.datetime.now().replace(hour=0, minute=0, second=0) -
previous.start.replace(hour=0, minute=0, second=0)).days
                self.group_data[group.id] = [time_allocated,
days_since_previous]

        if not lessons:
            self.lessons = {}  # format {day: lesson} of type {int: Lesson}
            # Lesson objects will have their start time updated (but will
NOT be saved)

    def random(self, threshold=100):
        """Generates a random solution
        This algorithm makes some attempt to minimise teacher clashes while
being quick to execute"""

        counter = 0
        for lesson in self.unscheduled_lessons:
            lesson:PotentiallyScheduledLesson
            teacher = self.get_teacher(lesson)
            day = random.randint(0, self.days-1)
            gaps = self.get_gaps(user_id=teacher.id, days=[day],
random_order=True)

            for gap_start, gap in gaps:  # for each (random) gap,
                if gap > lesson.duration + 1:  # if there's enough space
for a lesson (need at least 1 unit either side)
                    if gap < lesson.duration * 1.5:  # if there's not much
space...
                        lesson.start = gap_start + 1  # schedule for start
of gap (plus 1 unit break)
                    else:
                        latest_end = gap_start + gap - 2
                        lesson.start = random.randint(gap_start, latest_end
- lesson.duration)  # allocate to random position
                    self.lessons[day] = lesson
                    break
                else:
                    continue
            else:  # this triggers if the end of the loop is reached
without a break statement
                counter += 1
                if counter > threshold:
                    break

        return self

    def get_teacher(self, lesson:'PotentiallyScheduledLesson'):
        """Gets a teacher who teaches the given lesson
        Teachers are cached, so the cached version will be returned upon
any future calls"""
```

```
        if not hasattr(lesson, 'teacher'):
            lesson.teacher =
User.objects.filter(link__class_id__lesson__id__exact=lesson.id,
user_type__exact='teacher')[:1].get()
        return lesson.teacher

    def get_gaps(self, user_id, days=None, random_order=False) ->
List[Tuple[int, int]]:
        """Gets a list of the duration of every gap on the given day, in
time units

        Days should be a list of numbers, each indicating the number of
days since first_day

        Returns dict of form [(start of gap, length of gap)] of type [(int,
int)]"""
        if not days:
            days = range(self.days)

        previous = None
        gaps = []
        for day in days:
            sorted(self.lessons[day])
            for lesson in self.lessons[day]:
                if previous:
                    gaps.append((previous, lesson.start - previous))
                previous = lesson.start

        if random_order:
            random.shuffle(gaps)

        return gaps

    def get_gap_cost(self, gap_length):
        """Returns the value to add to the cost function for a gap of
length gap_length, in time units
        Gaps should never be negative, but if so this function will return
0"""
        if gap_length == 0:
            return 10
        # 1 is the perfect length -> 0 is returned
        elif gap_length in (2, 3):
            return 5
        elif gap_length == 3:
            return 2
        elif gap_length == 4:
            return 1
        else:
            return 0

    def cost(self):
        """Evaluates the cost function for the current solution"""
        POINTS_PER_TEACHER_CLASH = 1000
        POINTS_PER_STUDENT_CLASH = 10
```

```
        MAX_LOAD_CONSTANT = 23  # max load = c ln c, where c is this value
        EARLY_FINISH_CONSTANT = 10
        EVEN_ALLOCATION_CONSTANT_A = 0.4
        EVEN_ALLOCATION_CONSTANT_B = 8
        WEIGHTING_OF_DIFF = 100
        VARIETY_BASE = 2
        VARIETY_COEFFICIENT = 1

        total_cost = 0
        for day in range(self.days):
            # constraints 1 & 2: clashes
            teacher_clashes = 0
            student_clashes = 0
            schedules = {}  # {user_id: [time_slot]}
            for lesson in self.lessons:
                for user in lesson.get_users():
                    clashes = 0
                    if user.id not in schedules:
                        schedules[user.id] = []
                    for x in range(lesson.duration):
                        time_slot = lesson.start + x
                        if time_slot in schedules[user.id]:
                            clashes += 1
                        else:
                            schedules[user.id].append(time_slot)
                    if user.user_type == 'teacher':
                        teacher_clashes += clashes
                    else:
                        student_clashes += clashes
            clashes_cost = POINTS_PER_STUDENT_CLASH * student_clashes +
POINTS_PER_TEACHER_CLASH * teacher_clashes

            # constraint 3: even allocation of lesson times
            def sigmoid(a):
                return 1 / (1 + math.exp(-a))
            weighting = sigmoid(EVEN_ALLOCATION_CONSTANT_A +
EVEN_ALLOCATION_CONSTANT_B * (self.first_day - self.year_start).days)
            diffs = 0
            for group_id in self.group_data:
                diff = abs(self.group_data[group_id][0] -
self.desired_allocations[group_id]) / self.desired_allocations[group_id]
                diffs += diff
            even_allocation_cost = WEIGHTING_OF_DIFF * weighting * diffs

            # NOTE: constraint 4 is being computed with constraint 7

            # constraint 5: variety of subjects
            variety_cost = 0
            for group_id in self.group_data:
                variety_cost += VARIETY_COEFFICIENT * (VARIETY_BASE **
self.group_data[group_id][1])

            # constraint 6: max daily workload
            daily_workload_cost = 0
```

```
            for user_id in schedules:
                # NOTE: It is assumed that if any clashes occur, the user
will miss out on
                # all but one of the concurrent events, so it does not
contribute to their overall workload
                daily_workload_cost +=
max(math.exp(len(schedules[user_id])/MAX_LOAD_CONSTANT)-MAX_LOAD_CONSTANT,
0)

            # constraint 7: gaps
            gaps_cost = 0
            for user_id in schedules:  # NOTE: schedules is only used for a
list of user ids
                gaps = self.get_gaps(user_id)
                for gap in gaps:
                    gaps_cost += self.get_gap_cost(gap)

            # constraint 8: early finish time
            early_finish_cost = 0
            for user_id in schedules:
                finish_time = max(schedules[user_id])
                early_finish_cost += finish_time / EARLY_FINISH_CONSTANT

            debug_info = {
                'teacher clashes': teacher_clashes,
                'student clashes': student_clashes,
                'clashes cost': clashes_cost,
                'total diffs': diffs,
                'even allocation cost': even_allocation_cost,
                'variety cost': variety_cost,
                'daily workload cost': daily_workload_cost,
                'gaps cost': gaps_cost,
                'early finish cost': early_finish_cost
            }

            total_cost += clashes_cost + even_allocation_cost +
variety_cost + daily_workload_cost + gaps_cost + early_finish_cost

        return total_cost

    def fitness(self):
        """Returns the fitness value for a solution
        This is simply the negative of the cost value"""
        return -self.cost()

    def mutate(self, mutate_lessons_per_day=2):
        """Mutates the given solution (for use in a genetic algorithm)
        NOTE: The same lesson could be mutated twice (although unlikely)"""

        for day in range(self.days):
            for x in range(mutate_lessons_per_day):
                n = random.randint(1, 3)
                if n == 1:
                    # mutate start time of random lesson
```

```
                    i = random.randint(0, len(self.lessons[day])-1)
                    lesson = self.lessons[day][i]
                    latest_time = self.time_per_day - lesson.duration
                    lesson.start = random.randint(0, latest_time)
                elif n == 2:
                    # delete a random lesson
                    self.lessons[day].pop(random.randint(0,
len(self.lessons[day])-1))
                elif n == 3:
                    # add a random lesson
                    lesson = self.unscheduled_lessons.pop(random.randint(0,
len(self.unscheduled_lessons)-1))
                    latest_time = self.time_per_day - lesson.duration
                    lesson.start = random.randint(0, latest_time)
                    self.lessons[day].append(lesson)

        return self

    def add(self):
        """Update the database to include the start times for all lessons
currently stored within this object"""
        # TODO


class PotentiallyScheduledLesson(Lesson):
    """A Lesson used as part of a Timetable
    Notably, this abstracts the start time to make computation easier"""

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.start = None  # override this field with abstracted start time
        self.users = None

    def get_users(self):
        """Retrieves all users in this lesson, caching the result for
subsequent queries"""
        if not self.users:
            self.users =
User.objects.filter(link__group_id__lesson__id__exact=self.id)
        return self.users
```

[**WARNING**: The above code spans **450** lines (**9** pages). Jump to top: **4.2.1.9**]

4.2.2 Testing

All of this code has been produced without any testing, so it is expected that there will be many issues to resolve. The code was produced without testing as:

- It would have been significantly more time consuming to test each piece of code separately
- Test data takes some time to produce, so it is more efficient to test at the same time

Test data has been produced in **3.8.3**

4.2.2.1 Setting up a Testing Environment

The code cannot be run without setting up a script to run the code. The code is not entirely isolated from the Django environment, so setting up a script to run it from outside of the Django project will produce an error, as no database access is available. Therefore, a new temporary webpage will be created to run this script.

First, an additional function was created inside `timetabling.py` to create the timetable. This simply passes all arguments given directly to the `Population` class and returns the best timetable. This is beneficial because:

- It is quicker to type a single line each time
- If anything needs to be changed about the way the timetabling process is initiated, it can be changed here and take effect everywhere
- Calling the timetabling process is abstracted from the actual classes used in the implementation

```
def schedule(*args, **kwargs):
    """A helper function to generate the best timetable using a genetic
algorithm"""
    population = Population(*args, **kwargs)
    return population.start()
```

The URL `/test/` was then added

```
path('test/', views.test)
```

Finally, the view could be added to call the scheduling function

```
def test(request):
    print(schedule())
    return None
```

4.2.2.2 An Empty School

To ensure the code runs without error, the first test case will be on an entirely empty school – no lessons scheduled. The algorithm should output an empty timetable every time (as there are no lessons for it to schedule).

While there are lessons in the database, these have timestamps that have long since passed so should be ignored by the algorithm.

Each iteration below details the result and mitigation after the code was executed. There are **42** iterations before the test passed.

4.2.2.2.1 Iteration 1 – Fixing Imports

Upon, running the program, there were import related issues

```
File "/home/josh/Documents/coding/a-level-
timetabling/src/django_project/timetable/timetabling.py", line 6, in
<module>
```

```
    from src.django_project.timetable.models import Lesson, User, Group
ModuleNotFoundError: No module named 'src'
```

This could be corrected by making the import relative

```
from .models import Lesson, User, Group
```

4.2.2.2.2 Iteration 2 – Running the Test Script

The website was now running, and the rest of the website was functioning as intended.

Unsurprisingly, navigating to `/test/` produced an error

```
TypeError: __init__() missing 1 required positional argument:
'timetable_init_kwargs'
```

This parameter had been made required when it should be optional, which was easily fixed.

4.2.2.2.3 Iteration 3 – Generating Random Timetables

The algorithm had now progressed to generating a population of random timetables. The error occurred when querying for unscheduled lessons, on line 221

```
for unscheduled_lesson in
Lesson.objects.filter(fixed=False).exclude(start__lte=first_day):
```

Clearly, `first_day` was set to `None` when it should not be. An additional line was added to ensure that `first_day` is not `None` (to aid debugging / maintenance)

```
assert first_day
```

This assertion was triggered

However, `first_day` was being correctly set to the current day if not supplied

```
if first_day:
    self.first_day = first_day
else:
    self.first_day = datetime.datetime.now()
```

The issue was that the supplied parameter `first_day` was being used instead of `self.first_day`, which has been properly configured.

The issue was fixed by changing the line to the following:

```
for unscheduled_lesson in
Lesson.objects.filter(fixed=False).exclude(start__lte=self.first_day):
```

4.2.2.2.4 Iteration 4

```
TypeError: unsupported operand type(s) for +=: 'int' and
'datetime.timedelta'
```

(error)

```
time_allocated += lesson.duration
```

(code)

This is because `time_allocated` is of type `int`, whereas `lesson.duration` is of type `datetime.timedelta`. A `datetime.timedelta` object can be converted into seconds by using `.total_seconds()`

```
time_allocated += lesson.duration.total_seconds()
```

4.2.2.2.5 Iteration 5 – Timezones

```
TypeError: can't subtract offset-naive and offset-aware datetimes
```

```
days_since_previous = (datetime.datetime.now().replace(hour=0, minute=0,
second=0) - previous.start.replace(hour=0, minute=0, second=0)).days
```

Both were changed to be in UTC as standard

```
days_since_previous = (datetime.datetime.utcnow().replace(hour=0, minute=0,
second=0) - previous.start.replace(tzinfo=datetime.timezone.utc, hour=0,
minute=0, second=0)).days
```

4.2.2.2.6 Iteration 6 – Timezones 2

Changing them to UTC in this way did not fix the issue.

For some reason, a datetime object created with `.utcnow()` does not have timezone awareness (despite clearly being in UTC). Changing to the following fixed the issue:

```
datetime.datetime.now(datetime.timezone.utc)
```

4.2.2.2.7 Iteration 7

The timetable has now been successfully initialized. The next error was when retrieving the teacher for a given lesson

```
lesson.teacher =
User.objects.filter(link__group_id__lesson__id__exact=lesson.id,
user_type__exact='teacher')[:1].get()
```

`class` had been written instead of `group` in the above line

4.2.2.2.8 Iteration 8

The next issue resides in the `get_gaps` function

```
sorted(self.lessons[day])
```

```
KeyError: 0
```

Clearly, `self.lessons` is empty. It should be `{0: []}` (containing all days specified in the range)

A new piece of code was added at the start of `random()` to add an empty list to `self.lessons` for each day in the range

```
for d in range(self.days):
    self.lessons[d] = []
```

4.2.2.2.9 Iteration 9

The program has now progressed to the cost function without producing an exception.

```
for user in lesson.get_users():
```

AttributeError: 'int' object has no attribute 'get_users'

Clearly, lesson is an int.

```
for lesson in self.lessons:
```

Needed to be changed to:

```
for lesson in self.lessons[day]:
```

4.2.2.2.10 Iteration 10 – Timezones 3

In constraint 3, there was an issue with subtracting the first day and the start of the year

```
weighting = sigmoid(EVEN_ALLOCATION_CONSTANT_A + EVEN_ALLOCATION_CONSTANT_B
* (self.first_day - self.year_start).days)
```

```
TypeError: unsupported operand type(s) for -: 'datetime.datetime' and
'NoneType'
```

The start of the year could not be found as it relies on there being a single lesson in the database to use for the start of the year. As a fallback option, the start of the year will be set to the first of September in the event that there are no lessons.

```
if not self.year_start:  # if no lessons in database
    dt = datetime.datetime.now(datetime.timezone.utc)
    if dt.month < 9:  # jan - aug
        # previous september
        self.year_start = datetime.datetime(year=dt.year-1, month=9, day=1)
    else:
        # in current year
        self.year_start = datetime.datetime(year=dt.year, month=9, day=1)
```

4.2.2.2.11 Iteration 11

```
diff = abs(self.group_data[group_id][0] -
self.desired_allocations[group_id]) / self.desired_allocations[group_id]
```

```
TypeError: 'NoneType' object is not subscriptable
```

Either `self.group_data` or `self.desired_allocations` is None

It is likely that this is `self.desired_allocations`, as no default has been set in the event that this is not given (and this was not provided when testing). A default of `0` will be generated when setting up the group data

```
# default desired allocations in case it is not provided
if group.id not in self.desired_allocations:
    self.desired_allocations[group.id] = 0
    print(f"Warning: Group {group.id} had no desired allocation. Set to 0")
```

However, this may not be intuitive behaviour, so a warning is output when this occurs

4.2.2.2.12 Iteration 12

But, of course, this did not work when calculating percentage change

`ZeroDivisionError: float division by zero`

The default value was changed to be `1`. An allocation of `0` is not valid input data.

4.2.2.2.13 Iteration 13 – Changing how potential solutions are stored

```
population[i][1] = timetable.cost()
```

```
TypeError: 'tuple' object does not support item assignment
```

A tuple is being used to represent the timetable and its cost, but tuples are read-only.

It does not make sense to store these as tuples, when the cost can be stored within the timetable object itself. Much of the code was changed to now store the most recently evaluated cost function of a timetable.

```
population[i].cost = timetable.get_cost()
```

These changes were not inspected thoroughly, so more issues are expected as a result.

4.2.2.2.14 Iteration 14

This line produced an error because the method `.copy()` was never actually created

```
previous = self.copy()
```

However, this variable is only being used for the stopping condition

```
self.stopping_condition(self, previous, self.generations)
```

But this variable is not currently being used in the stopping condition. It is unlikely that it will be used in any stopping condition, so the syntax for the stopping condition will be changed to remove this parameter.

```
def should_stop(current_population, iterations):
    if iterations > 1000:
        return True
    else:
        return False
```

4.2.2.2.15 Iteration 15

Consequence of iteration 13

```
best_parent = min(candidates, key=lambda t, c: c)
```

```
TypeError: <lambda>() missing 1 required positional argument: 'c'
```

Changed to:

```
best_parent = min(candidates, key=lambda t: t.cost)
```

4.2.2.2.16 Iteration 16

Applied to line 105 also

```
previous_best_cost = min(candidates, key=lambda t: t.cost).cost
```

4.2.2.2.17 Iteration 17

4.2.2.2.17.1 The Error

```
AttributeError: 'tuple' object has no attribute 'cost'
```

Some lessons were still created as tuples.

4.2.2.2.17.2 Diagnosis 1

It would be useful to know whether this is the first iteration or a future iteration

```
print(self.generations)
```

The output was `0` – it is the first iteration

4.2.2.2.17.3 Diagnosis 2

```
print([type(candidate) for candidate in candidates])
```

The output was an array with a mixture of `<class 'timetable.timetabling.Timetable'>` and `<class 'tuple'>`. All the `Timetables` were at the start and all the `tuples` were at the end, meaning the issue must be to do with offspring generation.

4.2.2.2.17.4 Fixing the Error

The `crossover` function was outputting timetables as tuples and had not been changed.

```
return timetable, float('inf')
```

This could be changed to simply

```
return timetable
```

4.2.2.2.18 Iteration 18

```
p = candidate[1] / previous_best_cost
```

Was changed to

```
p = candidate.cost / previous_best_cost
```

4.2.2.2.19 Iteration 19 – No Exceptions Raised

This test appeared very promising as a result was not achieved initially - the algorithm must've been working. The code did not raise any exceptions.

However, after at least 30 seconds, the algorithm had still not produced a result.

Two changes were made:

- The current iteration number is printed to console at the start of each iteration
    - This helps monitor the progress of the algorithm
- The number of iterations to complete was reduced to 100
    - Lots of iterations is not currently necessary to test core functionality

4.2.2.2.20 Iteration 20 – A Warning About Timezones

*[NOTE: This issue continues until it is fixed in iteration **33**]*

The algorithm has not yet output a solution however the console is being occupied by many of the following message:

```
RuntimeWarning: DateTimeField Lesson.start received a naive datetime (2022-04-04 15:36:07.797784) while time zone support is active.
```

This could be due to the start time being updated in the `PotentiallyScheduledLessons`

The following line is not achieving its intended purpose and the start time is still being passed through Django's checks

```
self.start = None  # override this field with abstracted start time
```

A new field should be created instead

```
self.relative_start = None  # contains start time in time units relative to start of day
```

Usages were updated where possible

4.2.2.2.21 Iteration 21

This fix created a new issue

```
AttributeError: 'Lesson' object has no attribute 'relative_start'
```

Not all lessons were of the proper class `PotentiallyScheduledLesson` and were instead of type Lesson.

The unscheduled lessons are not properly converted to this new type.

```
unscheduled_lesson =
PotentiallyScheduledLesson.from_lesson(unscheduled_lesson)
```

4.2.2.2.22 Iteration 22

This had accidentally been changed to `relative_start`

```
self.year_start = first_lesson.relative_start
```

4.2.2.2.23 Iteration 23

The same issue was present on line **255**

4.2.2.2.24 Iteration 24

These changes have not removed the warning.

```
RuntimeWarning: DateTimeField Lesson.start received a naive datetime (2022-
04-04 15:59:43.828798) while time zone support is active.
```

Clearly, some were still of type Lesson.

In addition to the type hinting while generating a random lesson, type assertion was introduced so that an error would be raised if lessons are of the wrong type

```
assert isinstance(lesson, PotentiallyScheduledLesson)
lesson: PotentiallyScheduledLesson
```

The assertion error was raised

Despite being converted here, the conversion did not happen properly and the object is still of type Lesson

```
unscheduled_lesson =
PotentiallyScheduledLesson.from_lesson(unscheduled_lesson)
print(f"Type is {type(unscheduled_lesson)}")
```

All fields were being copied from the old object instead of just those that are necessary

```
DESIRED_FIELDS = [
    'id',
    'duration',
    'group_id'
]
@classmethod
def from_lesson(cls, lesson, *args, **kwargs):
    new_lesson = cls(*args, **kwargs)
    for field in cls.DESIRED_FIELDS:
        new_lesson.__dict__[field] = lesson.__dict__[field]
```

```
    new_lesson.get_users = cls.get_users
    return lesson
```

4.2.2.2.25 Iteration 25

This did not fix the issue. A new assertion was introduced

```
assert isinstance(new_lesson, PotentiallyScheduledLesson)
```

4.2.2.2.26 Iteration 26

This assertion was not triggered

The issue was that the wrong object was being returned:

```
return lesson
```

Needs to be changed to:

```
return new_lesson
```

4.2.2.2.27 Iteration 27

The lessons were now of the correct type. Despite all these issues, the warning was still displaying:

```
RuntimeWarning: DateTimeField Lesson.start received a naive datetime (2022-
04-04 16:16:22.800622) while time zone support is active.
```

The `PotentiallyScheduledLesson` object currently inherits from the `Lesson` class, however this is not actually necessary. Provided all relevant fields are given, the class does not need to be linked to a Django model.

The new class definition is as follows:

```
class PotentiallyScheduledLesson:
    """A Lesson used as part of a Timetable
    Notably, this abstracts the start time to make computation easier"""

    DESIRED_FIELDS = [
        'id',
        'duration',
        'group_id'
    ]

    def __init__(self, lesson):
        for field in self.DESIRED_FIELDS:
            self.__dict__[field] = lesson.__dict__[field]
        self.relative_start = None  # contains start time in time units
relative to start of day
        self.users = None
```

```
    def get_users(self):
        """Retrieves all users in this lesson, caching the result for
subsequent queries"""
        if not self.users:
            self.users =
User.objects.filter(link__group_id__lesson__id__exact=self.id)
        return self.users

    @classmethod
    def from_lesson(cls, *args, **kwargs):
        return cls(*args, **kwargs)
```

The classmethod `from_lesson` was included simply for backwards compatibility

4.2.2.2.28 Iteration 28

```
unscheduled_lesson.group
```

Had to be renamed to

```
unscheduled_lesson.group_id
```

4.2.2.2.29 Iteration 29

Again, the algorithm did not complete. The same warning was still produced.

Searching by '`.start`' showed that there is no instance in which the `start` field from an object derived from `Lesson` is updated.

As this issue is a warning, a full traceback is not produced. However, this would be very useful to diagnose the issue.

The following code was copied from StackOverflow for debugging purposes. It was initially posted by mgab and later edited by Gareth Rees. All code on StackOverflow is available under the Creative Commons License.

```
import traceback
import warnings
import sys

def warn_with_traceback(message, category, filename, lineno, file=None,
line=None):
    traceback.print_stack()
    log = file if hasattr(file,'write') else sys.stderr
    log.write(warnings.formatwarning(message, category, filename, lineno,
line))

warnings.showwarning = warn_with_traceback
```

After extensive debugging, it became clear that this issue is potentially not actually with this algorithm and is instead an issue with how the start times are being stored. It is possible that they are stored as naïve instead of time zone aware.

The following code was executed to make all lesson start times aware

```
for lesson in Lesson.objects.filter():
    if lesson.start and is_naive(lesson.start):
        lesson.start = make_aware(lesson.start)
```

4.2.2.2.30 **Iteration 30**

It didn't work.

The issue was in fact contained within this file and was not with how the lesson times are stored. The issue is that the timezone passed in for comparison is naïve instead of aware.

```
for lesson in Lesson.objects.filter(group_id__id__exact=group.id,
start__lte=datetime.datetime.now(tz=datetime.timezone.utc)).order_by('start
')
```

4.2.2.2.31 Iteration 31

The issue was now fixed for the end of the algorithm, however this was still being output during initialisation.

`first_day` was made timezone aware

```
self.first_day = datetime.datetime.now(tz=datetime.timezone.utc)
```

4.2.2.2.32 Attempt 32

This produced an error as `year_start` was not yet timezone aware. This was fixed

4.2.2.2.33 Attempt 33 – The Warning is now Fixed

After 13 iterations, the issue is now fixed.

Much of the console is now occupied by the following message:

```
Warning: Group 1 had no desired allocation. Set to 1
Warning: Group 2 had no desired allocation. Set to 1
```

A desired allocation will be provided to remove these warnings

```
desired_allocation = {0: 1, 1: 1}
print(schedule(timetable_init_kwargs={'desired_allocations':
desired_allocation}))
```

4.2.2.2.34 Iteration 34

However, `timetable_init_kwargs` was not always being provided when initialising new Timetable objects. This was fixed.

4.2.2.2.35 Iteration 35

The algorithm is now running but not completing. There are no warnings in the console.

It now needed to be determined where the algorithm was stopping

```
print('Choosing parents')
parents = self.choose_parents()
print('Generating offspring')
offspring = self.generate_offspring(parents)
candidates = parents + offspring
print('Choosing new population')
self.population = self.choose_new_population(candidates)
print('End of iteration')
```

4.2.2.2.36 Iteration 36

The algorithm was stopping while choosing the new population. These print statements have been removed.

There is only one while loop within this function. The while loop will likely never be reaching its stopping condition:

```
while len(new_population) < self.popsize:
```

This is because it only stops when the population exceeds the desired population size. It may be true that there are not enough candidates to choose from to form the new population.

The following code was added:

```
if len(candidates) <= self.popsize:
    return candidates
```

4.2.2.2.37 Iteration 37

This has prevented the issue from displaying, however the issue is broader than initially seemed. The current configuration is conducive to a slowly dying population. The population is meant to remain at a constant size.

The issue is that the number of offspring was being calculated wrongly. Using a separate parameter `num_offspring` to configure the number of offspring to produce at each iteration corrected the issue.

4.2.2.2.38 Iteration 38

```
potential_new_lessons = parent1.lessons[day] + parent2.lessons[day]
```

```
KeyError: 0
```

While `self.lessons` was properly being initiated if `.random()` was then called, it was not initiated properly if generated through other means. The following code:

```
for d in range(self.days):
    self.lessons[d] = []
```

Was moved from `.random()` into `__init__()`

```
self.lessons = {}  # format {day: lesson} of type {int:
PotentiallyScheduledLesson}
for d in range(self.days):
    self.lessons[d] = []
```

4.2.2.2.39 Iteration 39 – More Timezones

Despite both `first_day` and `year_start` being timezone aware in other parts of the code, the following error was output:

```
weighting = sigmoid(EVEN_ALLOCATION_CONSTANT_A + EVEN_ALLOCATION_CONSTANT_B
* (self.first_day - self.year_start).days)
```

```
TypeError: can't subtract offset-naive and offset-aware datetimes
```

It was first determined which of these objects was timezone aware

```
print(is_naive(self.first_day))
```

4.2.2.2.40 Iteration 40

The result was unexpected – the function is called many times with no issue but there is one instance in which the attribute is changed, and it becomes naïve.

This is because `self.first_day`, when initialising a `Population` object, is not created timezone aware

```
self.first_day = datetime.datetime.now(datetime.timezone.utc)
```

However, this shows that this `first_day` parameter is not being used initially (and is only used in future iterations of the algorithm). This is an issue as it should be passed directly to `Timetable`.

Some parameters are now passed while initialising a random `Timetable` object

```
self.population.append(Timetable(first_day=self.first_day, days=self.days,
time_per_day=self.time_per_day,

seconds_per_unit_time=self.seconds_per_unit_time, day_start=self.day_start,
                                  **timetable_init_kwargs).random())
```

4.2.2.2.41 Iteration 41 – Choosing the Best Solution

The algorithm has now progressed to choosing the best solution. The following line:

```
if timetable.get_cost < best[1]:
```

Was changed to:

```
if not best or timetable.get_cost() < best.cost:
```

4.2.2.2.42 Iteration 42

The timetabling algorithm has now successfully completed with no exceptions raised.

The final output is:

```
{0: []}
```

Which is exactly as intended.

After 42 iterations, the test has passed and the empty school had been successfully timetabled.

4.2.2.3 A Single Lesson

After the base case of the school being entirely empty, the first test data used was the simplest possible valid case – a single lesson taught by a single teacher to a single student. The expected output was simply any non-erroneous result that timetabled the lesson at any point in the school day. There is a constraint to encourage the lesson to start as early as possible, however this is a minor constraint and not required to form a valid solution.

The lesson was newly created within group 1 (Ma1), which contains a single test teacher and a single test student.

Then, the output was updated to include the relevant information – the time each lesson was scheduled at.

```
def test(request):
    desired_allocation = {0: 1, 1: 1}
    output = schedule(timetable_init_kwargs={'desired_allocations':
desired_allocation})
    print('BEGIN TIMETABLED LESSONS')
    for day in output.lessons:
        for lesson in output.lessons[day]:
            print(lesson.start)
    print('END TIMETABLED LESSONS')
    return HttpResponse('Timetabling complete - printed to console')
```

Clashes will not be an issue in this timetable, however the algorithm will now have to manage to schedule a lesson at some point in the day which will be more challenging than the base case earlier. It is expected to be possible in fewer iterations than the previous scenario as many errors have already been fixed.

In the end, **9** iterations were required before this test initially passed, however a further **13** iterations were required before the test passed after a change was made.

4.2.2.3.1 Iteration 1 – Nothing Scheduled + Optimisations

4.2.2.3.1.1 Initial Test

The algorithm completed with no errors. Unfortunately, the lesson was not scheduled – the output was the same as before.

```
Timetabling complete!
BEGIN TIMETABLED LESSONS
END TIMETABLED LESSONS
```

4.2.2.3.1.2 Diagnosing

The next step was to inspect the contents of `self.unscheduled_lessons` to diagnose if the issue relates to querying from the database or scheduling.

```
print(self.unscheduled_lessons)
```

4.2.2.3.1.3 Optimising

When writing the diagnosis for this issue, it became apparent that the code was not properly optimised.

The `Timetable` class is currently fetching from the database each time it is created unless `self.unscheduled_lessons` is provided. This is not intended behaviour and is highly inefficient. It should be calculated once while constructing the `Population` class and then passed into all individuals in the population.

```
def get_unscheduled_lessons(first_day, days=1):
    unscheduled_lessons = []
    per_class = {}
    for unscheduled_lesson in
Lesson.objects.filter(fixed=False).exclude(start__lte=first_day):
        unscheduled_lesson = PotentiallyScheduledLesson(unscheduled_lesson)

        if unscheduled_lesson.group_id in per_class:
            per_class[unscheduled_lesson.group_id] += 1
        else:
            per_class[unscheduled_lesson.group_id] = 1

        if per_class[unscheduled_lesson.group_id] <= days:  # this helps to
reduce the possibilities to consider
            unscheduled_lessons.append(unscheduled_lesson)

    return unscheduled_lessons
self.unscheduled_lessons = get_unscheduled_lessons(self.first_day,
self.days)
if unscheduled_lessons:
    self.unscheduled_lessons = unscheduled_lessons
else:
    self.unscheduled_lessons = get_unscheduled_lessons(self.first_day,
self.days)
random.shuffle(self.unscheduled_lessons)
```

The same change was also applied to the generation of group data

4.2.2.3.2 Iteration 2 – Further Diagnosis

The output showed that the unscheduled lessons were being generated correctly:

```
[<timetable.timetabling.PotentiallyScheduledLesson object at
0x7f630e278bb0>]
```

The issue must be to do with it not being scheduled.

A potential next step is to output all the generated solutions. It may simply be the case that the algorithm is generating the correct solution but does not recognise it as optimal.

This must be done by manually interfacing with the Population class and cannot be done with the schedule function

```
print('BEGIN POPULATION')
for individual in population.population:
    print(individual.lessons)
print('END POPULATION')
```

4.2.2.3.3 Iteration 3 – Further Diagnosis 2

The output consisted of entirely empty populations.

The next step is to test that random generation is working properly. It is expected that some of the lessons produced by the random generator will include the lesson, and some will not.

```
print(self.lessons)
```

The output was not as intended – all of the timetables were empty.

A minor change was added to ensure a copy is being created of the list for each timetable. This ensures that timetables do not interfere with each other.

```
self.unscheduled_lessons = list(unscheduled_lessons)
```

4.2.2.3.4 Iteration 4 – Further Diagnosis 3

This did not make any difference to the result.

Some additional print statements were added to diagnose the issue further

4.2.2.3.5 Iteration 5 – An Issue with Gaps

This revealed that the gaps for each teacher was empty. However, this was not the intended behaviour for this algorithm (however is intended when this is used in other places).

When used in the cost function, for example, the algorithm should return the gap between each lesson i.e. for N lessons, there are $N-1$ gaps

However, this algorithm also needs gaps bounding the start and end of the day, i.e. for N lessons, there are $N+1$ gaps (provided the lessons do not touch a day boundary).

The function can take a new Boolean parameter `boundaries` to determine whether the day boundaries should be included in gap determination.

```
if boundaries:
    previous = 0
for lesson in self.lessons[day]:
    if previous:
        gaps.append((previous, lesson.relative_start - previous))
    previous = lesson.relative_start
if boundaries:
    gaps.append((previous, self.time_per_day - previous))
```

4.2.2.3.6 Iteration 6

Executing the code produced an error

```
if gap > lesson.duration + 1:  # if there's enough space for a lesson (need
at least 1 unit either side)
```

```
TypeError: unsupported operand type(s) for +: 'datetime.timedelta' and
'int'
```

The above change has worked, and lessons are now being handled within the random timetable generation. However, this issue needs to be fixed.

`lesson.duration` is of type `datetime.timedelta`, when it is meant to be in time units. A new attribute of `PotentiallyScheduledLesson` will be created called `lesson.relative_duration` which will include the duration in time units.

```
self.relative_duration =
math.floor(lesson.duration.total_seconds()/seconds_per_time_unit)
```

```
gap > lesson.relative_duration + 1
```

4.2.2.3.7 Iteration 7

```
potential_new_lessons = parent1.lessons[day] + parent2.lessons[day]
```

```
AttributeError: 'Timetable' object has no attribute 'lessons'
```

`Timetable` objects were only having lessons defined if not passed in as a parameter

```
if not lessons:
    self.lessons = {}  # format {day: lesson} of type {int:
PotentiallyScheduledLesson}
    for d in range(self.days):
        self.lessons[d] = []
```

The following code was added:

```
else:
    self.lessons = lessons
```

4.2.2.3.8 Iteration 8

This same line produced a different error:

```
TypeError: unsupported operand type(s) for +: 'PotentiallyScheduledLesson'
and 'list'
```

One of the timetables must have been updated to be stored as `[lesson]` instead of `{day: lesson}`

In fact, it was in this function that the issue was occurring

```
new_lessons = []
for day in range(self.days):
    potential_new_lessons = parent1.lessons[day] + parent2.lessons[day]
    random.shuffle(potential_new_lessons)
    for x in range(len(potential_new_lessons)//2):
        new_lessons.append(potential_new_lessons.pop(0))
```

Was changed to:

```
new_lessons = {}
for day in range(self.days):
    new_lessons[day] = []
    potential_new_lessons = parent1.lessons[day] + parent2.lessons[day]
    random.shuffle(potential_new_lessons)
    for x in range(len(potential_new_lessons)//2):
        new_lessons[day].append(potential_new_lessons.pop(0))
```

4.2.2.3.9 Iteration 9 – Test Passed

This iteration successfully produced a timetable which satisfied the given constraints and passed the test.

However, an issue was noticed with the data printed out regarding the randomly generated lessons – none of them were empty. It should be introduced a random probability that a lesson will be skipped and not scheduled so that the randomly generated timetables are random.

This was set to `0.2` initially.

```
if random.uniform(0, 1) < self.random_lesson_skip_probability:
    continue
```

4.2.2.3.10 Iteration 10 – Still No Lessons

This resolved the issue.

However, this also made the test fail and return an empty timetable. Either:

- The algorithm was removing these lessons and not adding them back, leaving final solutions consisting of only 0 lessons
- The optimal solution is present however is not being chosen as the best solution

The previous code to output the entire population was re-enabled

4.2.2.3.11 Iteration 11 – A New Constraint

The latter was true – the final algorithm involves some with one lesson and some with no lessons. Interestingly, there are fewer individuals with the lesson than are produced from random, suggesting there is an issue with the cost function. It also suggests that the genetic algorithm is working and individuals with a lower cost value are being selected, which is good.

Constraint 3 was intended to mitigate this issue and ensure that each class gets its desired allocation. However, this does not take effect towards the start of the year. This means an additional constraint will be required. This will be denoted constraint 3a.

It may be appropriate for this constraint to take effect only at the start of the year, or may be beneficial all year round. As a first test, it will be enabled all year round.

This is inherently a positive factor - a solution is better if it manages to schedule more lessons. However, the cost function should not be negative as this may interfere with other code (notably selecting individuals with probability that is directly proportional to their cost value – this would give negative probability).

Two potential approaches seem most feasible:

- If the cost function goes negative, make it positive
    - It is unlikely that the cost function will go negative, so this will likely not have to take effect
    - If it goes negative, solutions that are not the same are considered just as good as each other even though one is better than the other
- Do the total number of lessons subtract the number scheduled (add points based on number failed to schedule)
    - In a large school, this will add massive values to the cost functions of all timetables. This will make solutions appear more similar and will not be as easily selected.

It is not possible to determine which would be better without testing. The first approach will be used initially.

Constraint 3a was added to the cost function as shown:

```
# constraint 3a: how many lessons
lessons_scheduled_cost = POINTS_PER_LESSON_SCHEDULED *
len(self.lessons[day])
```

Note that POINTS_PER_LESSON_SCHEDULED is negative

The final cost is then made 0 if negative

```
max(total_cost, 0)
```

4.2.2.3.12 Iteration 12

This did not fix the issue.

It would be useful to output the `debug_info` created inside the function to understand which constraints are contributing most to the value of the cost function

```
debug_info = {
    'teacher clashes': teacher_clashes,
    'student clashes': student_clashes,
    'clashes cost': clashes_cost,
    'total diffs': diffs,
    'even allocation cost': even_allocation_cost,
    'lessons scheduled cost': lessons_scheduled_cost,
    'variety cost': variety_cost,
    'daily workload cost': daily_workload_cost,
    'gaps cost': gaps_cost,
    'early finish cost': early_finish_cost
}
```

```
print(debug_info)
```

A typical output is as follows:

```
{'teacher clashes': 0, 'student clashes': 0, 'clashes cost': 0, 'total
diffs': 20698.0, 'even allocation cost': 2069800.0, 'lessons scheduled
cost': -1, 'variety cost': 8392704, 'daily workload cost': 0, 'gaps cost':
0, 'early finish cost': 5.0}
```

Clearly, there was a bug with constraint 3 – even allocation.

Determining the weighting would be useful to determine whether this is an issue with the weighting or with computing the percentage difference.

4.2.2.3.13 Iteration 13 – Investigating Weighting

The weighting was `1.0`, when it was expected it would be very close to `0`. It is possible that `first_day` is incorrect due to incorrectly entered data. This is manually passed into the scheduling process.

```
population = Population(desired_allocations=desired_allocation,
year_start=datetime.datetime.now(datetime.timezone.utc))
```

4.2.2.3.14 Iteration 14

`year_start` was not added as an argument to the `Population` class, so has now been added. This could be modified as it was for `unscheduled_lessons` and `group_data`

```
if year_start:
    self.year_start = year_start
```

```
else:
    self.year_start = get_year_start()
```

```
def get_year_start():
    first_lesson = Lesson.objects.order_by('start')[:1].get()
    year_start = first_lesson.start
    if not year_start:  # if no lessons in database
        dt = datetime.datetime.now(datetime.timezone.utc)
        if dt.month < 9:  # jan - aug
            # previous september
            year_start = datetime.datetime(year=dt.year - 1, month=9,
day=1, tzinfo=datetime.timezone.utc)
        else:
            # in current year
            year_start = datetime.datetime(year=dt.year, month=9, day=1,
tzinfo=datetime.timezone.utc)

    return year_start
```

The code to generate the year start was moved to a function as it is used twice and code should not be repeated.

4.2.2.3.15 Iteration 15

The weighting was still `1.0` despite seemingly having `self.first_day` and `self.year_start` both as the current date

The difference in days was evaluated

```
(self.first_day - self.year_start).days
```

4.2.2.3.16 Iteration 16

The difference in days turned out to be 217, which is also the approximate difference between the date of testing and the 1st of September.

This was confirmed

```
print(self.year_start)
```

4.2.2.3.17 Iteration 17

```
2021-09-01 00:00:00+00:00
```

Somewhere, `year_start` was not being properly passed into the cost function.

The issue turned out to be while initialising the `Timetable` – the argument was missed

4.2.2.3.18 Iteration 18

The weighting is now 0.599 (3 s.f.). It should be almost zero.

Constants A and B were the wrong way round in the equation

```
weighting = sigmoid(
    EVEN_ALLOCATION_CONSTANT_B + EVEN_ALLOCATION_CONSTANT_A *
(self.first_day - self.year_start).days)
```

4.2.2.3.19 Iteration 19

The weighting is now 1.00 (3 s.f.), which is not desired.

Changing the value of b from `8` to `−8` has finally fixed this issue.

4.2.2.3.20 Iteration 20 – Weighting Issue Fixed

Value of weighting: `0.0003353501304664781`

Debug info:

```
{'teacher clashes': 0, 'student clashes': 0, 'clashes cost': 0, 'total
diffs': 20698.0, 'even allocation cost': 694.1077000395164, 'lessons
scheduled cost': -1, 'variety cost': 8392704, 'daily workload cost': 0,
'gaps cost': 0, 'early finish cost': 9.4}
```

However, the even allocation cost is still too high. It should be divided by some large constant k.

```
even_allocation_cost = WEIGHTING_OF_DIFF * weighting * diffs /
EVEN_ALLOCATION_CONSTANT_K
```

The value of this constant was initially set to `1000`

Equally, the cost due to variety also needs to be reduced significantly

```
variety_cost /= VARIETY_CONSTANT
```

The value of this constant was initially set to `10,000`

4.2.2.3.21 Iteration 21 – Adjusting Constants

The algorithm is still favouring solutions that do not have as many lessons scheduled.

The variety constant was increased to `100,000`

(The magnitude of) `POINTS_PER_LESSON_SCHEDULED` was increased to `−10`

This will need to be reviewed for larger timetables – a linear model may not be appropriate

4.2.2.3.22 Iteration 22 – Fully Passed

The best solution was correctly chosen from the population. The test had now passed.

```
BEGIN BEST SOLUTION
16
END BEST SOLUTION
```

4.2.2.4 Two Lessons

To test the algorithm's ability to handle clashes, two lessons should be scheduled.

The best timetable must:

- Schedule both lessons...
- ...at different times (that do not clash)
- ...consistently

The best timetable could also:

- Schedule these lessons as early as possible (so everyone can get home early)

For the test to be passed, the algorithm must produce 10 consecutive timetables that schedule both lessons in a way where they do not conflict with each other.

The lessons belong to two separate classes. Each class is taught by the same teacher and attended by the same student. One lesson is 1 hour long whereas the other is 2 hours long.

If the lessons were part of the same class, the second lesson would be ignored and not scheduled.

4.2.2.4.1 Iteration 1 – Investigating Errors

Code:

```
sorted(self.lessons[day])
```

Error:

```
TypeError: '<' not supported between instances of
'PotentiallyScheduledLesson' and 'PotentiallyScheduledLesson'
```

It is possible to change this line to retrieve the start times of each lesson however it is also possible to define how Python should handle comparison operators. This would be a more elegant way to sort by start time.

```
def __gt__(self, other):
    return self.relative_start > other

def __lt__(self, other):
    return self.relative_start < other
```

4.2.2.4.2 Iteration 2 – Passed?

The algorithm completed and scheduled the two lessons in a way that does not conflict.

The second test also passed.

However, this output is not sufficient to check that the test has passed. The output was updated to include the lesson topic. This helps identify which lesson is which and check more accurately that there are no conflicts.

```
print(f"{lesson.topic} - {lesson.relative_start}")
```

4.2.2.4.3 Iteration 3 – Duplicate Lessons

This change produced a very strange output

```
BEGIN BEST SOLUTION
This was created using the form - 36
This was created using the form - 10
END BEST SOLUTION
```

Not only has the algorithm seemingly scheduled the same lesson twice, but this lesson is not actually one of the lessons intended to be scheduled.

All lessons that are not intended for scheduling have had `fixed` set to `True`

4.2.2.4.4 Iteration 4

There are no longer any lessons that should not be scheduled here. However, there are sometimes two of the same lesson. Other times, there are (correctly) one of each lesson. And sometimes only one lesson is scheduled.

```
BEGIN BEST SOLUTION
This will be timetabled - 49
Testing clashes with a long lesson - 15
END BEST SOLUTION
```

```
BEGIN BEST SOLUTION
This will be timetabled - 2
This will be timetabled - 34
END BEST SOLUTION
```

It is likely this issue is resulting from mutations.

When a lesson is removed from the timetable, it is not re-added to the unscheduled lessons.

```
# add to random position in unscheduled lessons
self.unscheduled_lessons.insert(random.randint(0,
len(self.unscheduled_lessons)), lesson)
```

However, when adding a random lesson, it now had to be considered that the list of unscheduled lessons may be empty. If so, it should do nothing

```
# add a random lesson
if self.unscheduled_lessons:
    lesson = self.unscheduled_lessons.pop(random.randint(0,
len(self.unscheduled_lessons) - 1))
    latest_time = self.time_per_day - lesson.relative_duration
    lesson.relative_start = random.randint(0, latest_time)
    self.lessons[day].append(lesson)
```

But it quickly became apparent that this was not actually the issue, as it was being correctly removed from the list when adding. The issue was more likely to be during offspring production.

Currently, offspring are produced by combining lessons randomly from each parent. It is easy to see that this will easily end up with duplicate lessons. Duplicates will now be removed (and will not contribute to the counter of how many lessons to choose).

```
for x in range(len(potential_new_lessons) // 2):
    if potential_new_lessons:
        lesson = potential_new_lessons.pop(0)
        if lesson.id not in added_ids:
            new_lessons[day].append(lesson)
            added_ids.append(lesson.id)
```

4.2.2.4.5 Iteration 5 – Duplicate Lessons Fixed

Five tests were conducted and not one of them scheduled the same lesson twice. Of these, four scheduled both lessons and one only scheduled one out of two lessons.

This only happens when the lesson is scheduled really early – the early finish time outweighs the number of lessons scheduled.

This will be fixed by only applying the early finish time beyond a certain point in the day, as it is redundant otherwise. This will be at around lunchtime, which is at approximately 48 time units into the day.

```
finish_time = max(schedules[user_id]) - EARLIEST_EARLY_FINISH
if finish_time > 0:
    early_finish_cost += finish_time / EARLY_FINISH_CONSTANT
```

4.2.2.4.6 Iteration 6

To check that a solution is valid:

- If “`This will be timetabled`” is first, there must be a difference of at least `12` between the start time of the two lessons
- If “`Testing clashes with a long lesson`” is first, the difference must be at least `24`

The algorithm produced eight correct solutions, before outputting a solution that only scheduled one lesson.

```
BEGIN BEST SOLUTION
Testing clashes with a long lesson - 3
END BEST SOLUTION
```

It would be useful to have the breakdown of the cost function for the final solution that is chosen. `get_cost` will have a new Boolean parameter, `debug`. This determines whether to return the cost or the debug info with full breakdown.

This will be recomputed at the end. While this adds a minimal performance impact, it does not influence the complexity and is only present for testing.

```
print(f"BEST SOLUTION COST: {output.cost}")
print(f"BREAKDOWN: {output.get_cost(debug=True)}")
```

4.2.2.4.7 Iteration 7

```
BEGIN BEST SOLUTION
Testing clashes with a long lesson - 43
This will be timetabled - 70
BEST SOLUTION COST: 155.10726123505256
BREAKDOWN: {'teacher clashes': 0, 'student clashes': 0, 'clashes cost': 0,
'total diffs': 20699.0, 'even allocation cost': 0.6941412350525631,
'lessons scheduled cost': -20, 'variety cost': 167.81312, 'daily workload
cost': 0, 'gaps cost': 0, 'early finish cost': 6.6}
END BEST SOLUTION
```

Clearly, variety is the only important factor in the timetable.

The variety constant will be further increased to `1,000,000`. Variety can only be explored properly in a large school timetabling for multiple days.

4.2.2.4.8 Iteration 8 – Test Passed

Variety is now less important than the number of lessons that are scheduled.

Ten consecutive tests passed. There were no issues with clashes throughout all of the tests done.

It can now be assumed that the algorithm is able to select against simple student / teacher clashes.

4.2.2.5 Simulating a Full School

Full details surrounding the test data in use in this section are available in **3.8.3.4**

According to the test data, lessons are to be generated randomly according to a uniform distribution. Durations will range from 30 minutes to 2 hours (as permitted by the interface).

Three lessons were created for every group, so that there are plenty to use for timetabling. This is the minimum number in the interface before a warning is displayed.

The Python code to generate these lessons is as follows:

```
for group in Group.objects.filter():
    for x in range(3):
        lesson = Lesson()
        lesson.group_id = group.id
        duration = random.randint(6, 24)*300
        lesson.duration = datetime.timedelta(seconds=duration)
        lesson.topic = "An automatically generated lesson"
        lesson.save()
        print(f"Created lesson with duration {duration/60} minutes")
```

The remaining test data was then implemented using the admin interface.

4.2.2.5.1 Initial Testing

4.2.2.5.1.1 Iteration 1

The query:

```
lesson.teacher =
User.objects.filter(link__group_id__lesson__id__exact=lesson.id,
user_type__exact='teacher')[:1].get()
```

Did not return any results. This means there must be at least one group that does not have a teacher.

The lesson ID was output to find out which lesson is causing this issue

```
print(lesson.id)
```

4.2.2.5.1.2 Iteration 2

```
36
27
18
7
33
15
```

Lesson ID 15 did not have a valid teacher.

This lesson belongs to group 4 (Ph1).

The database was updated to link `Albert Einstein` to his group.

4.2.2.5.1.3 Iteration 3

Adding an object did not fix the issue.

Each iteration, many of the (randomly ordered) teachers are being successfully retrieved however it is always failing on the lesson with ID 15.

The issue was because `Albert Einstein` had his user type set to `student` instead of `teacher`.

4.2.2.5.1.4 Iteration 4 – Modifying Constraint 3a

The timetable completed with no errors and produced a timetable consisting of five lessons.

There is not currently the means to determine the average amount of lesson time per student. This could be implemented as a replacement to constraint 3a as it is a more accurate representation of the desired timetable than simply the number scheduled. It also avoids the issue of the cost function potentially going negative.

A new parameter was created to indicate the desired number of lessons to be scheduled on a given day. The cost function will use the average lesson time per student per day and will use an exponential model to add to the cost function.

Again, the `schedules` dict created in constraints 1 & 2 was helpful

```
# constraint 3a: how many lessons
total_lesson_time = 0
n_students = 0
for user_id in schedules:
    if user_types[user_id] == 'student':
        total_lesson_time += len(schedules[user_id])
        n_students += 1
average_lesson_time = total_lesson_time / n_students
lessons_scheduled_cost = DESIRED_LESSONS_MULTIPLIER * DESIRED_LESSONS_BASE
** (self.desired_lesson_time - average_lesson_time)
```

The base was initially set to `1.1` and the multiplier `25`.

The average lesson time was added to the debug info

4.2.2.5.1.5 Iteration 5

The algorithm gave a solution with an average lesson time of `15.96`. The target value is `20`.

However, the solution did include three student clashes (15 minutes of missed lesson time)

4.2.2.5.2 Tweaking the Algorithm

The algorithm now consistently runs without error and produces valid timetables. The final stage of the process is to tweak the variables to try and improve the solution it outputs.

The algorithm is currently completing within approximately one second. Note that it is allowed almost 14 hours to run if necessary.

Change	Justification	What happened?	Kept ?
`DESIRED_LESSONS_BASE` from `1.1` to `1.2`	This would favour solutions with more lessons scheduled	No noticeable impact	Yes
Variety cost now divided by group data	Variety cost is a bit high, and does not depend on the number of groups	No noticeable impact	Yes
Generations from `2` to `100`	Generations was only set to 2 for testing purposes	Runtime increased to approximately 5 seconds Solutions got worse (average cost value ~180 -> ~200)	Yes

4.2.2.5.3 Increasing the number of generations

For testing purposes, the number of generations was set to 2. This means that solutions will effectively be randomly chosen from the size of the population. This was not intended.

Increasing this value had the opposite effect as intended – it made the solutions noticeably worse. Clearly, there was some issue in the genetic algorithm causing it to select the wrong solutions.

A change was made for debugging purposes – the cost of the best solution at the end of each iteration was added. This increases the time taken to find a solution but will be removed in the final product.

The cost of the best solutions at the end of each iteration are as follows:

Iteration 0	152.1
Iteration 1	112.9
Iteration 2	58.6
Iteration 3	46.6
Iteration 4	54.1
Iteration 5	67.6

The cost then stayed exactly the same for the remaining 94 iterations

This solution included only a single lesson. This lesson was Economics, which was likely a long lesson.

4.2.2.5.4 A Normal School Day

There are currently 10 groups, which should consist of enough data to generate the timetable for a day. Note that the threshold should be slightly lower as a typical school would consist of more groups, giving the algorithm more choice and flexibility.

The school day will be reset to normal. The algorithm should produce a timetable that schedules at least an average of 40 time units per student. This is slightly lower than the 44 in the success criteria because:

- Groups / classes were not generated by an algorithm
- A typical school would involve more groups, allowing for more flexibility

Running this gave another poor solution which only scheduled two lessons.

Iteration 0	957
Iteration 1	449
Iteration 2	932
Iteration 3	852
Iteration 4	530
Iteration 5	108
Iteration 6	730
Iteration 7	185
Iteration 8	346

This continued to vary until converging to 457.

There are two main issues currently:

- The primary concern is that this algorithm varies massively between iterations. It should be keeping the best solution from each iteration and passing this through to the next.
- Another concern is that after about 20-25 iterations, the algorithm converges on a solution which does not vary at all. This solution is clearly suboptimal.

The number of generations has been reduced to 30 as the algorithm usually converges by this point.

4.2.2.5.5 Investigating Selection

It should be the case that the best solutions are being selected to proceed to the next iteration. No evidence of this happening has been observed so far.

Additionally, there is code to ensure that the 5 best individuals from the previous population survive unchanged into the next iteration. Clearly, this code is not working. The best solution should never get worse between iterations.

4.2.2.5.5.1 Iteration 1

There was a bug in the choose_new_population() method

```
if candidate.get_cost == float('inf'):
    candidate.get_cost = candidate.get_cost()
```

(old)

```
if candidate.cost == float('inf'):
    candidate.cost = candidate.get_cost()
```

(new)

4.2.2.5.5.2 Iteration 2

However, this was not the issue. This issue simply meant the algorithm ran slightly slower than it should.

Another issue is that when calculating the probability to reject an individual, the candidate's cost was compared to the previous best cost. It should instead be compared to the previous highest cost. This has potential to be a significant issue that will result in rejecting many good solutions.

```
previous_highest_cost = max(candidates, key=lambda t: t.cost).cost
```

```
p = candidate.cost / previous_highest_cost
```

4.2.2.5.5.3 Iteration 3

This has somewhat improved the performance of the algorithm. However, it was intended that the 5 best individuals from the population would always survive to the next iteration. Clearly this is not happening as the best cost value is sometimes increasing between iterations.

This should be happening because:

- When choosing parents, the current best solution is chosen in turn, meaning the best parents will be at the start of the list
- Offspring are generated from this list of parents without modifying the parents themselves
- The new population is passed in a list with the parents at the beginning and the offspring at the end
- The 5 elements from the start of this concatenated list are automatically passed to the new population

To test if this is working, the solution at the beginning of the list will be output at the end of each iteration (in addition to the best). It is expected that the solution at the beginning of the list will mostly be the best solution, unless the best solution has got better since the previous iteration.

Three other steps have been attempted before this was considered further. This will be explored further in iteration **7**

4.2.2.5.5.4 Iteration 4

The final population output was updated to include the teacher and start time of each lesson.

Every single lesson in the final population was taught by a single teacher. This could be the reason why not many lessons are being scheduled.

4.2.2.5.5.5 Iteration 5

A print statement was added to determine how many unscheduled lessons there are.

```
print(f"UNSCHEDULED LESSONS: {len(self.unscheduled_lessons)}")
```

There are 10 unscheduled lessons, as intended.

4.2.2.5.5.6 Iteration 6

The chance of mutation was reduced from 0.7 to 0.5 and the number of mutations was reduced from 2 to 1.

4.2.2.5.5.7 Iteration 7

There is still an issue with how the population is being chosen. The best solution is rarely at the start, when it should nearly always be.

Outputting the list of parents after parents selection shows that these parents are properly sorted.

```
[1577.4946185930135, 1748.9679623204036, 1923.9056667178802,
1996.5702280226046, 2067.8221998850677, 2119.6074167799325,
2157.7412655937173, 2295.3342090945957, 2319.986739184169, …]
```

However, the first solution (after choosing the new population) is equal to the 6th element in this list.

4.2.2.5.5.8 Iteration 8

They are being properly chosen to add to the new population:

```
[476.3775928120491, 808.4410327214134, 867.2221998850677,
965.305841838732, 1319.581059450895]
```

But this condition was clearly triggering as the function did not complete:

```
if len(candidates) <= self.popsize:
    return candidates
```

This does not include the individuals already chosen, which it should do.

```
if len(candidates)+self.guaranteed_surviving_parents <= self.popsize:
    return new_population + candidates
```

4.2.2.5.5.9 Iteration 8

This has fixed the issue and drastically improved the performance of the algorithm. The final population includes many solutions each with many lessons taught by a variety of teachers.

The final solution, despite only scheduling two lessons, allegedly achieved an average lesson time per student of 46 time units. This is clearly incorrect.

The issue was that students are only being considered if they have a lesson scheduled. Students which have no lessons scheduled are not considered as part of this algorithm. This makes the algorithm favour giving some students empty days, as they are not considered an issue without any lessons.

A new list is being generated to contain all of the students in the school that are a member of a group. Students not currently a member of a group will not be included in the timetable.

```
if not self.all_students:
    for user_id in schedules:
        if user_types[user_id] == 'student':
            total_lesson_time += len(schedules[user_id])
            n_students += 1
else:
    for student in self.all_students:
        if student.id in schedules:
            total_lesson_time += len(schedules[student.id])
        n_students += 1
```

4.2.2.5.5.10 Iteration 9 – A good timetable, but still increasing in cost

The timetable produced now included 6 lessons with an average lesson time of 30. This is much more reasonable (yet only ¾ of the way to the threshold required).

However, it is still the case that the solutions are sometimes increasing in cost between generations, despite the best solutions supposedly being conserved.

The issue was that, somehow, the parents were being modified. After modification, their costs were not being re-evaluated so the solutions were unexpectedly getting worse.

Re-evaluating the costs somewhat addresses the problem but does not identify the root cause of the issue. It needs to be understood why the parents are being changed while producing offspring.

```
def generate_offspring(self, parents):
    """Generates all the offspring"""

    offspring = []
    for x in range(self.num_offspring):
        parent1 = random.choice(parents)
        parent2 = random.choice(parents)
        offspring.append(self.crossover(parent1, parent2))
```

```
    offspring = self.mutate(offspring)

    return offspring
```

This must be happening during the crossover function

4.2.2.5.5.11 Iteration 10

`unscheduled_lessons` was not being passed into the offspring. This was fixed, but not the issue.

4.2.2.5.5.12 Iteration 11

```
print(f"Parents has been modified: {sum(initial_parents[i] ==
list(parents)[i] for i in range(len(parents)))}")
```

All of the parents have been modified during crossover

4.2.2.5.5.13 Iteration 12

It was then determined which attribute(s) have been modified

```
for attribute in initial_parents[0].__dict__:
    if initial_parents[0].__dict__[attribute] !=
parents[0].__dict__[attribute]:
        print(attribute)
```

Despite claiming all parents had been modified, this code did not output any attributes.

If all of the attributes of the object are the same, then the object should be the same.

4.2.2.5.5.14 Iteration 13

The equality operator will be redefined for timetables to compare whether the list of lessons are the same, as this is the only factor in the cost function for a timetable.

However, the parents were still displayed as having been modified.

The following code did not trigger:

```
if parents[0].lessons != initial_parents[0].lessons:
    print("Lessons have changed!")
```

4.2.2.5.5.15 Iteration 14

Clearly, there must be some issue with the way that it is being determined if the parents have been modified or not. Outputting the lessons for each showed that they are identical and are not modified in any way.

However, the total cost of the parents does change between iterations. Typically, it increases.

4.2.2.5.5.16 Iteration 15

Some print statements confirmed that no changes happen to the parent from within the `crossover()` function.

4.2.2.5.5.17 Iteration 16

It also does not change anywhere within the `generate_offspring()` function.

4.2.2.5.5.18 Iteration 17

The next test was to run the cost function multiple times and print the total cost function each time. It may be the case that the cost function is causing the issue

However this was not the case – the cost of the parents is somehow changing outside of the `generate_offspring()` call, but nowhere inside of the function.

4.2.2.5.5.19 Iteration 18

There is no change as the parameter is passed into the function

```
def generate_offspring(self, parents, initial_parents=None):
    """Generates all the offspring"""

    if parents != initial_parents:
        print("IT CHANGED!!!")
```

Yet the cost function is still changing after this call to generate offspring.

The parents appear to change between the end of the generate offspring function and directly after it has finished exiting.

This cannot be correct, so a new approach will be attempted

4.2.2.5.5.20 Iteration 19

This issue can be avoided by passing in a copy of parents into the `generate_offspring()` function

```
offspring = self.generate_offspring(list(parents))
```

There are no longer any changes to the parents.

However, even with no apparent changes to the parents, the solutions still manage to get worse.

4.2.2.5.5.21 Iteration 20

Disabling mutation has solved the issue (however this is not a solution as mutation is required for the algorithm to function). Clearly, the issue relates to mutation.

Mutation is only being called on offspring. Mutation does not directly make any changes to the parents, only the offspring.

4.2.2.5.5.22 Iteration 21

Another potential issue was that lessons weren't being copied when passed into offspring (so both individuals are referencing the same lesson objects). This would be an issue as if the start time is updated in one timetable, it will unintentionally update in the other.

```
new_lessons[day].append(lesson.copy())
```

```
def copy(self):
    return copy.copy(self)
```

This was not the sole issue causing parents to change.

4.2.2.5.5.23 Iteration 22

However, this does not affect the list of unscheduled lessons. It is possible that the lists of unscheduled lessons are not being copied so are all referencing each other, causing solutions to get worse.

Copies are now created for all lists of unscheduled lessons

```
self.unscheduled_lessons = []
for lesson in unscheduled_lessons:
    self.unscheduled_lessons.append(lesson.copy())
```

4.2.2.5.5.24 Iteration 23 – Issue Fixed

Despite using more memory, this has fixed the issue. Each instance does require a unique copy as they must be added to and removed from (and shuffled), while remaining independent of other timetables.

4.2.2.5.6 Other Issues

The following issues were found throughout the main investigation process. They have been moved here to avoid interrupting the flow of the primary investigation

4.2.2.5.6.1 Mutation

Mutation was not properly being called.

This has been fixed.

It was not properly being checked if there are actually any lessons on a given day when mutating.

```
if self.lessons[day]:
```

This fixed the issue.

4.2.2.5.6.2 Choosing the Best Solution

The code for choosing the best solution was replaced with a single line

```
best = min(self.population, key=lambda t: t.cost)
```

This had no impact on the result, but made the code shorter and easier to read for a Python programmer.

4.2.2.5.6.3 Not Enough Offspring

However, there should be enough offspring generated that this condition should not have been triggering at all. The number of offspring generated at the end of each generation is now being output.

The issue was that only the parents and the offspring were selected for the new generation, rather than the entire population (including those not selected as parents). This has since been fixed.

4.2.2.5.6.4 Reducing Guaranteed Surviving Parents

Guaranteed surviving parents will be reduced to 2 as this could give better results on the algorithm

4.2.2.5.7 More Tweaking

The algorithm now functions entirely as expected. A typical solution has a cost of 400, producing a timetable with 6-7 lessons with an average lesson time per student of 30 and no clashes. This is good, but not good enough (target value is 40).

Further tweaking the constants should improve the performance of the algorithm.

Change	Justification	What happened?	Kept ?
Generations from 30 to 100	It was only set to 30 for testing purposes	The algorithm took longer. Solutions were slightly better	Yes
Simplified random function*	Would introduce more variance within population	No noticeable impact ***NOTE**: Explained in **4.2.2.5.7.1***	No
Population size 100 -> 200	Larger populations give better results	Solutions were noticeably better	Yes
Generations 100 -> 50	In most instances, the best solutions achieved within the first 30 iterations	Algorithm ran twice as fast No impact on quality of solutions	Yes
Parents AND offspring 50 -> 100	Population size was doubled, so these were doubled	Solutions were slightly worse	No
Offspring 50 -> 100	More offspring means more variation	No noticeable impact	Yes
Mutation amount 1 -> 2	Algorithm is converging early so not enough variation in population	Slightly better	Yes
Mutations 2 -> 3	Increasing further could increase solutions even more	Not converging as early. Solutions about the same quality	Yes
Surviving parents 2 -> 5	This could increase the cost value	Significant improvement: cost 329, lesson time 35 (in first test)	Yes
Generations 50 -> 100	Solutions are not converging as quickly any more	Best solution so far achieved: cost 291, lesson time 35	Yes
Generations 100 -> 200	Increasing further could have an even bigger impact	Runtime 25 seconds. New best achieved (264), however this was before generation 100	No

Mutations 3 -> 5	Testing what happens with lots of mutations	Solution was significantly worse	No

The algorithm is now able to consistently schedule 35 time units of lessons per student. Given that:

- These classes were not generating using an algorithm, so may have conflicts that make a good solution difficult or even impossible
- The new solution will increase productivity, so fewer total hours of lessons are required
- Each student is only a member of three groups, when they should be a member of six

Despite being lower than the target value of 40, this result can be deemed satisfactory.

The development of the timetabling algorithm is now complete.

4.2.2.5.7.1 An Alternative Random Function

It was tested if simplifying the random function to generate 6 random lessons would produce better solutions (since there would be more variation in the population).

```
for x in range(random.randint(6, 8)):
    self.lessons[0].append(random.choice(self.unscheduled_lessons))

return self
```

Surprisingly, this had very little impact on the quality of solutions produced. It appears that the algorithm does enough mutations that the initial solutions do not matter significantly.

The function will be modified to accept a Boolean variable so that it can be chosen which algorithm to use. It will initially be configured to use the algorithm which was created first.

4.2.2.5.7.2 A different approach to store the cost function

A variable will contain a value that states if the timetable has been modified. If it has not been modified since the last cost function calculation, the cached value can be returned. Otherwise, it needs to be recalculated.

Running this made the algorithm take significantly longer to execute, so clearly has not worked. The cost function must have been calculated many more times.

The issue was that `self.modified` was not being set to `False` at the end of the cost function, so it was always regarded as having been modified and would always be recomputed.

```
self.modified = False
```

The issue is now fixed.

4.2.3 Implementing into the Wider Project

The final stage is to implement this algorithm into the wider project. This will consist of:

- Calling the algorithm

- Adding the result into the database

This can be contained within one function, `schedule_lessons()`, which must then be called every day.

The function should schedule the next full 2-week cycle. If any lessons are already scheduled on a day, that day can be assumed to be fully timetabled.

4.2.3.1 Creating the Function

The function was implemented as shown:

```
def schedule_lessons(iterations=10, look_ahead_period=14):
    """Creates a timetable using the unscheduled lessons from the
database"""
    base_day =
datetime.datetime.now(datetime.timezone.utc).replace(minute=0, hour=0,
second=0)
    for x in range(look_ahead_period):
        day = base_day + datetime.timedelta(days=x)
        end_of_day = day.replace(hour=23, minute=59, second=59)
        if day.weekday() <= 4:  # a weekday
            if not Lesson.objects.filter(start__after=day,
start__before=end_of_day):
                print(f"Scheduling on {day}")
                best_result: Optional[Timetable] = None
                for x in range(iterations):
                    print(f"Iteration: {x+1}")
                    population = Population(year_start=base_day)
                    output: Timetable = population.start()
                    if best_result is None or output.get_cost() >
best_result.get_cost():
                        best_result = output

                print(f"Adding best result (cost:
{best_result.get_cost()})")
                best_result.add()
```

Notably, for each day the best timetable is being computed 10 separate times (with different populations). This is because the quality of the solutions was varying between iterations, so it would be beneficial to take the best solution from multiple iterations. There are no major time constraints with the execution, so it would be beneficial to use additional time to improve the solutions. Using more than 10 iterations would simply use more energy for little benefit.

4.2.3.2 Calling this Function Daily

This has a very standard implementation in Django through a decorator

```
@periodic_task(run_ever=crontab(hour=20, minute=0))
```

It was configured to run at 8pm each evening to allow plenty of time to schedule (starting at midnight would suffice, however doing it earlier is beneficial as people are able to see their timetables earlier).

4.2.3.3 Adding to the Database

The following function was initially created to test whether it works. It does not yet update the database as there may potentially be issues with the timetable.

```
for day in self.lessons:
    for lesson in self.lessons[day]:
        new_lesson = Lesson()
        new_lesson.group_id = lesson.group
        new_lesson.duration = lesson.duration  # relative_duration not
required because duration is never modified
        new_lesson.topic = lesson.topic
        new_lesson.fixed = True
        start_time = self.first_day + day + self.day_start +
lesson.relative_start * self.seconds_per_unit_time
        print(start_time)
        new_lesson.start = start_time
        # new_lesson.save()
```

First, a new object of class `Lesson` is initialised. Then, the attributes are set appropriately. The start time is then output for debugging purposes. Once confirmed to work, the line to save it to the database will be uncommented.

4.2.3.4 Testing

Initially, a `return` statement was added at the end so that only the first day was scheduled.

The program should not be updating the database in any way.

4.2.3.4.1 Iteration 1

```
if not Lesson.objects.filter(start__after=day, start__before=end_of_day):
```

Should be:

```
if not Lesson.objects.filter(start__gte=day, start__lte=end_of_day):
```

4.2.3.4.2 Iteration 2

The algorithm completed in approximately 2 minutes.

The number of iterations was reduced from `10` to `5` and the number of generations within each iteration was reduced from `100` to `75`. This makes testing quicker and shouldn't have a significant impact on the quality of solutions. These changes will likely be reversed in the final product.

4.2.3.4.3 Iteration 3

The algorithm completed in 55 seconds.

There was a bug in `add()`

```
new_lesson.group_id = lesson.group
```

Was changed to

```
new_lesson.group_id = lesson.group_id
```

4.2.3.4.4 Iteration 4

```
datetime.timedelta(days=day)
```

4.2.3.4.5 Iteration 5

```
start_time = self.first_day + datetime.timedelta(days=day) + self.day_start
+ datetime.timedelta(seconds=lesson.relative_start *
self.seconds_per_unit_time)
```

4.2.3.4.6 Iteration 6

The function `add()` was configured to return the start time of just one of the lessons.

4.2.3.4.7 Iteration 7 – Incorrect Start Times

The start time returned was on the wrong day and was not at a time during the school day

```
2022-04-16 20:37:09.987392+00:00
```

The date should be the 8th April, however this represents the 16th April. The school day begins at `8:30` and ends at `18:00`.

It was changed to return the start times of all the lessons.

4.2.3.4.8 Iteration 8

`first_day` was not being supplied to the timetabling algorithm

```
population = Population(year_start=base_day, first_day=base_day +
datetime.timedelta(days=x))
```

4.2.3.4.9 Iteration 9

The times were all between `00:00` and `7:30`.

The durations were at times that were a multiple of 5 plus 30 seconds. They should be exactly a multiple of 5.

Each individual component that is being summed to form the start time is now being output separately.

4.2.3.4.10 Iteration 10

`day_start` had value 8 days and 30 seconds, when it should actually be 8 hours and 30 seconds.

A `timedelta` takes parameters days and then seconds, with hours and minutes being keyword parameters. It was only being supplied with non-keyword parameters.

This was fixed:

```
day_start=datetime.timedelta(hours=8, minutes=30)
```

4.2.3.4.11 Iteration 9 – Start Times Fixed

All the lessons are at times which fall on the correct day and are within the school day.

The number of iterations was temporarily reduced to 1, and the algorithm was allowed to run for the full two-week period to test how it functions timetabling for the full period. It is still the case that it should not be updating the database

4.2.3.4.12 Iteration 10 – Producing a Full Timetable

There were no clear issues as a result of this test.

The algorithm was now ready to timetable a full two-week period, updating the database. This will be the final test.

Temporary code was added such that each time a lesson was scheduled with a given group, a new lesson is added (of random duration).

```
for lesson in best_result.lessons[0]:
    # group =
Group.objects.filter(group_id__exact=lesson.group_id)[:1].get()
    new_lesson = Lesson()
    new_lesson.group_id = lesson.group_id
    duration = random.randint(6, 24)
    new_lesson.duration = datetime.timedelta(seconds=duration * 300)
    new_lesson.topic = f"Automatically generated while timetabling"
    new_lesson.fixed = False
    new_lesson.save()
```

The number of generations has been reverted back to `100`, and the number of iterations back to `10`.

It is expected that the algorithm will take 20-30 minutes to produce a timetable for the full 2-week period.

4.2.3.4.13 Iteration 11 – Testing Complete

The timetable has been successfully produced.

Runtime: 22 minutes 5 seconds

[Note that this runtime is not too high as the final product will be allowed to run between 6pm and 8am to timetable a single day, which is significantly more time than what is used]

The cost functions were recorded for each day that was scheduled

- 265
- 244 (lowest yet)
- 269
- 267
- 260
- 269
- 206 (beats this record to become the new lowest)
- 246
- 271

Average (arithmetic mean): 255

The performance of this timetable will be evaluated in the evaluation section.

No further changes will be made to the timetabling algorithm. The development phase of the project is now complete.

5 Evaluation

5.1 Introduction

Overall, the project was a massive success. Despite the limited time constraints, the final product managed to produce a timetable that is almost matching the quality of existing solutions, while introducing the hugely useful variable-length lessons. There is a powerful admin interface to input the students, teachers and classes. Teachers can then use an easy-to-use interface to choose exactly how long they need for each lesson. Each day, an algorithm uses these lengths to produce a well-balanced timetable with minimal clashes. Students and teachers alike can access their timetables securely from anywhere using a web-based interface.

However, the project is not yet ready to be used in a school environment. This section will consider which areas need to be developed further before this product can be used in schools.

5.2 Success Criteria

In this section, it is determined, for every requirement, whether that requirement has been fully met, partially met or not met. Where appropriate, testing is conducted to inform this process.

The requirements were initially determined in **2.7.3**. Where appropriate, extracts from this table have been included under each section.

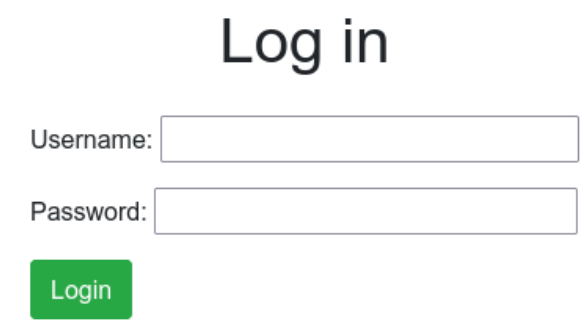
5.2.1 Essential Requirements

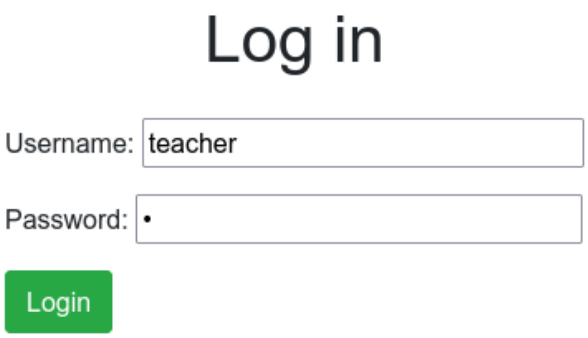
These requirements are integral to the solution. They are the highest priority and must be met for the solution to be successful in a school environment.

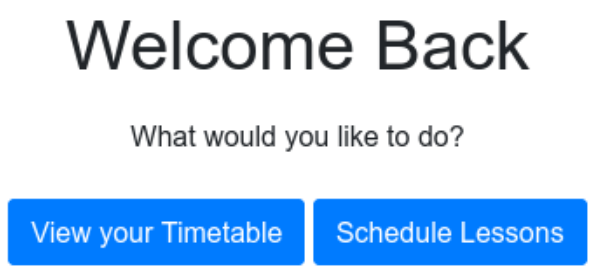
5.2.1.1 Requirement 1 – Viewing Timetables

#	Requirement	Test	Justification / Notes
1	Students and teachers must be able to log in securely and view their timetable		Core functionality

This was tested as follows:







(The password was set to ‘p’ for testing purposes. Password validation has since been enabled so that users are not able to have such insecure passwords)

< Mon 11 Tue Wed Thu Fri >

Ma1

This will be timetabled

Room

12:10 - 13:10

Ma2

An automatically generated lesson

Room

14:50 - 16:45

Back to Home

Logout

Logged out

You have successfully logged out

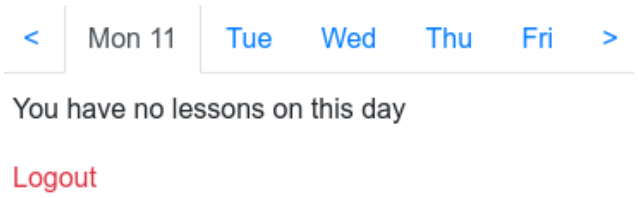
Log in again

Log in

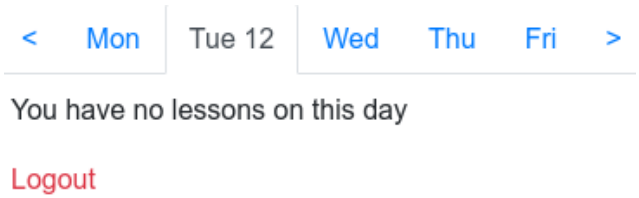
Username: student

Password: •

Login



(The test student is not a part of any group)



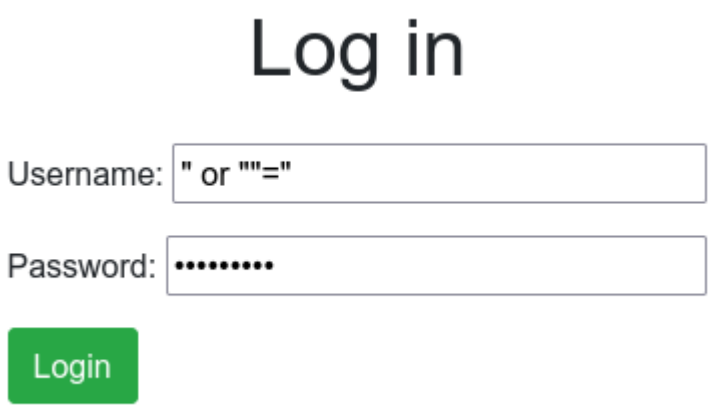
All expected functionality was present and worked as expected.

This requirement was entirely fulfilled.

5.2.1.2 Requirement 2 – Security

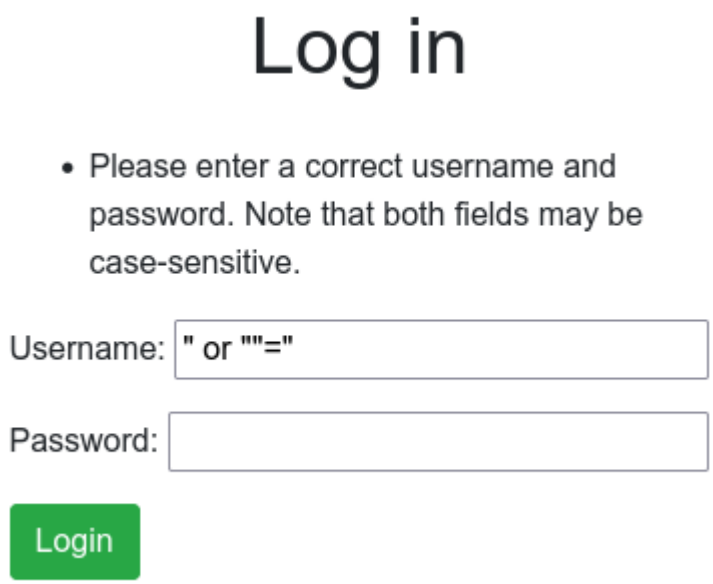
2	It must be resistant to the following forms of attack: - Cross-site request forgery - SQL injection	SQL injection can be tested by inputting SQL special characters into the username field on the login page. CSRF can be tested by manually sending a POST request to the server and ensuring it is rejected	While this data is not very sensitive, it is always good practice to keep data private.

5.2.1.2.1 SQL Injection



NOTE: Contents of password field are identical to username field

The login attempt was rejected:



5.2.1.2.2 Cross-site Request Forgery

On the form in `/schedule/` a POST request was sent to add the lesson to the database. The request was correctly rejected as it was not authorized.

5.2.1.2.3 Conclusions

This requirement has been entirely met

5.2.1.3 Requirement 3 – Accessibility

3	The student and teacher interfaces must be very accessible. It must be clear and easily usable for those with less experience using IT. No prior knowledge of the software is required.	An average student / teacher should, given relevant credentials, be able to log in and understand their timetable within **one** minute. Out of at least **3** randomly selected students, all **3** should be able to complete this task. The software should satisfy modern accessibility standards.	Many students and teachers will be used to the traditional system so it must be a positive experience to migrate to the new system. If the UI is confusing, it will be difficult for them to adjust.

5.2.1.3.1 Conducting the Test

Three students were selected using opportunity sampling.

Students were told the following information:

"Your account credentials are written on the paper in front of you. The username is '12345', and the password is 'chickenNuggets'. Note that all of your subjects will display as maths – this is only because test data is being used. You must identify which teacher is teaching you at 16:00 next Monday 11th April."

This test is demanding and requires students to understand the interface fully. Students must:

- Type in their login information
- Recognise that the timetable defaults to the current date and that they need to change the date
    - Students were not explicitly told about the need to change the date
- Understand that the right arrow advances forward one week
- Look at the list of lessons and correctly identify 'Mr Hamilton' as the teacher

This test was conducted on a Friday. The test was conducted on the mobile interface. The timer began as soon as the login page was presented onscreen. Students were not allowed to view previous students completing the test.

The timetable to be read is as follows:

The results are as follows:

Student	Result	Comments
1	21 seconds	The student took 14 seconds to type in their login details. Afterwards, they instantly recognised they had to use the arrows in the bar at the top of the screen and quickly found the correct teacher.
2	16 seconds	This student was even faster as they could type quickly. As with student 1, they immediately recognised how to use the top bar and read the timetable correctly.
3	29 seconds	This student was stopping to look around the interface, however this did not bring their final time up too much.

All three students passed the test. This requirement has been fully met.

5.2.1.3.2 Accessibility Standards

The software was tested for accessibility features. Many browsers include these features by default, however some require websites to be written in a certain way for the features to work.

5.2.1.3.2.1 Improving Accessibility on the Login Page

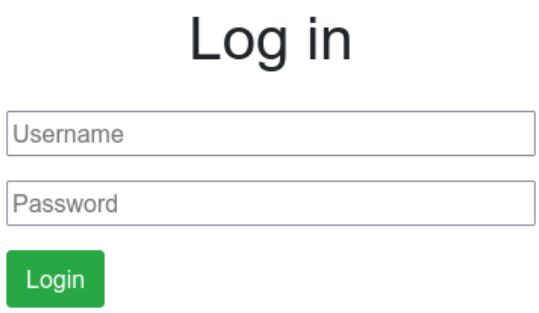
Upon visiting the page, the student is first shown the login page. It is very clear and readable, however does not work well with any students who may be using a screen reader. Browsers will not

read out ‘username’ or ‘password’ when the user clicks on the relevant field. The use of placeholders to indicate the purpose of each field does look good, however is not very accessible.

Despite it being the evaluation phase, this issue is very easy to fix.

To improve this, labels can be used instead of the placeholder text. The labels are properly assigned with the `for` attribute for two reasons:

- In many browsers, clicking on the label will focus the relevant text field
- Clicking on either the text field or the label will read out the label for users relying on screen readers



First, the placeholder was replaced by a label in the `LoginForm`

```
class LoginForm(AuthenticationForm):
    username = UsernameField(widget=forms.TextInput(
        attrs={'class': 'w-100 mb-3',
               'placeholder': '',
               'id': 'username-input'}),
        label='Username')
    password = forms.CharField(widget=forms.PasswordInput(
        attrs={
            'class': 'w-100 mb-2',
            'placeholder': '',
            'id': 'password-input',
        }),
        label='Password')
```

However, this made the width of the elements incorrect, as they should no longer span the width of their parent. To change this, the `w-100` class was removed and the widths were instead manually assigned (as they do not fit one of the predefined Bootstrap classes). By trial and error, the desired widths are `267px` and `270px` respectively

```
#username-input {
    width: 267px;
}
#password-input {
    width: 270px;
}
```

This now worked:

5.2.1.3.3 Conclusion

Overall, accessibility and usability are excellent. Students are able to intuitively understand the interface very quickly and there are also many accessibility features for those who need them.

This requirement has been entirely fulfilled.

5.2.1.4 Requirement 4 – Browser Support

4	All major browsers should be supported, including both desktop and mobile versions. This includes, at minimum: - Chrome (and other chromium-based browsers) - Firefox - Safari All must be supported on the latest version at the time of release.		There are many students and teachers in a school which will be using many different devices with many different browsers. It should work on as many as possible so that it is convenient to view the timetable (and not any more difficult than currently)

Throughout the project, the program has been locally hosted on Linux and tested in Firefox. To test this requirement, the project must be uploaded to a VPS (Virtual Private Server) and made accessible through a domain. Without this step, it will only be possible to test browsers from within Linux.

The repository was cloned into the directory and the program was ran as usual. The credentials had to be uploaded separately as they are not contained within the public repository. The port also had to be changed to 8001 as port 8000 was in use. Some settings were changed, then the website was accessible remotely so can be tested on other devices.

The results of testing are as follows:

Platform	Browser	Result
Desktop	Firefox	Already tested throughout
Desktop	Chrome	No issues

Desktop	Safari	No device available to test*
Mobile	Firefox	No issues
Mobile	Chrome	No issues
Mobile	Safari	No device available to test*

*This requires an apple device. None were available at the time of testing.

Each test consisted of general UI navigation, to include logging in as a teacher or a student, viewing some pages and then logging back out again.

It is highly unlikely that there will be any additional issues on Safari, therefore the requirement will be considered as most likely met. All browsers that were tested worked with no issues.

5.2.1.5 Requirement 5 – Speed of Website Load

5	The website must load fast	**Two** of **three** attempts must pass: - The latency to the server should be calculated by pinging the IP address - The server must respond in less than 50ms more than this latency *NOTE: Each test must be conducted on a different page*	Accessing timetable should not be any less convenient than currently, otherwise users may respond negatively to the change. A slow interface will make the interface frustrating to use. *NOTE: The latency to the server depends on implementation which is not in the scope of the project*

The server will be hosted on localhost as this eliminates the ping to the server.

Test #	Page	Latency
Test 1	`/student/`	24ms
Test 2	`/login/`	22ms
Test 3	`/scheduled/`	22ms

All three tests passed. This requirement has been met

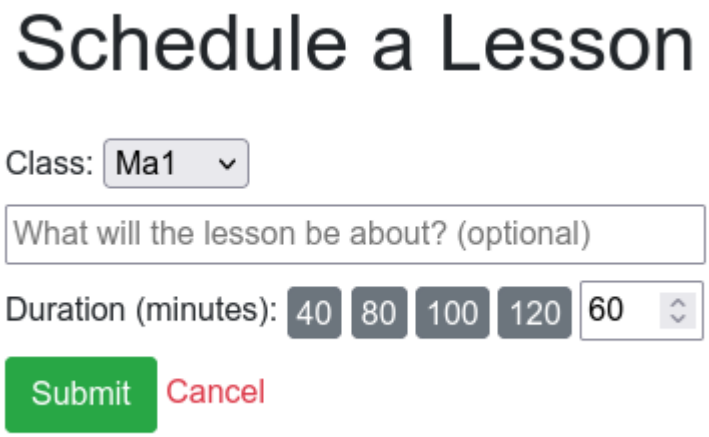
5.2.1.6 Requirement 6 – Scheduling Lessons

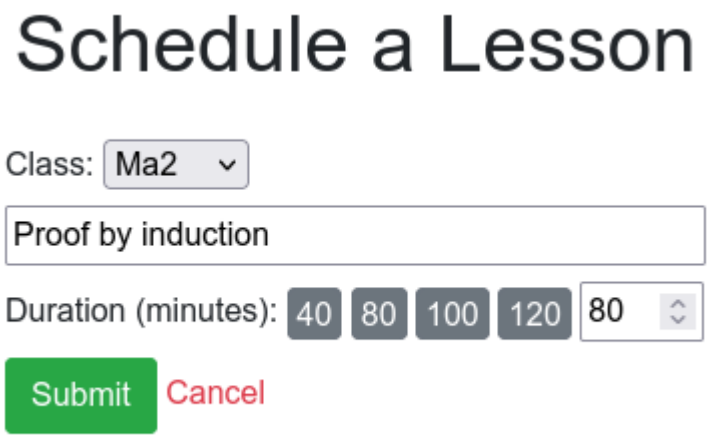
6	Teachers must be able to input lessons to be scheduled along with their duration		Core functionality

Navigating to `/scheduled/` produced a list of lessons that have not yet been scheduled.

Clicking 'new lesson':

Produced a user-friendly interface:





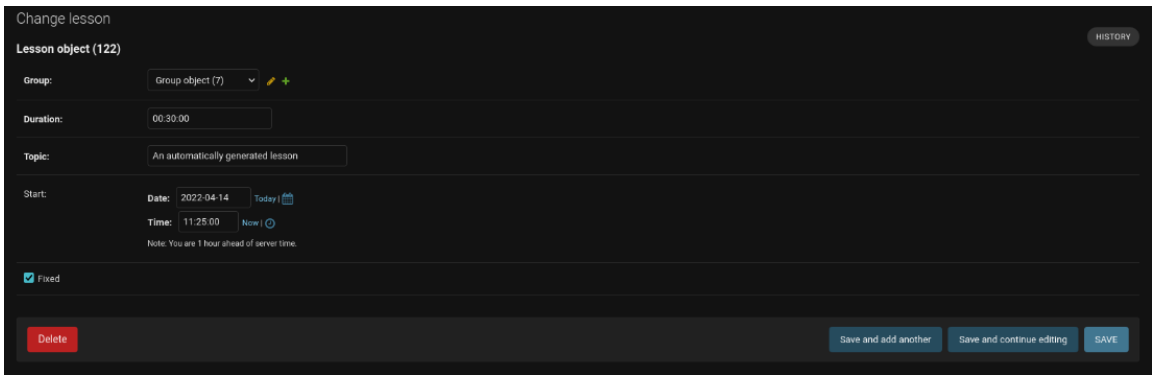
Clicking submit correctly redirected to the list of lessons, in which the displayed properly.

This requirement has been fully met.

5.2.1.7 Requirement 7 - Admin Interface

7	The administrator(s) must have full control over the entire system. This must include, but is not limited to: - Manually changing lesson times / overriding algorithmic decisions - Adding / removing users - Changing classes	Listed functionality should be tested	The head of timetabling is responsible for ensuring the timetable is working properly and, in the event that something goes wrong, may be held accountable. A school requires 100% uptime, so if something did go wrong there must be full control to override any decisions made and ensure temporary functionality of the system. *NOTE: The admin interface is not bound by requirement **3***

5.2.1.7.1 Manually changing lesson times

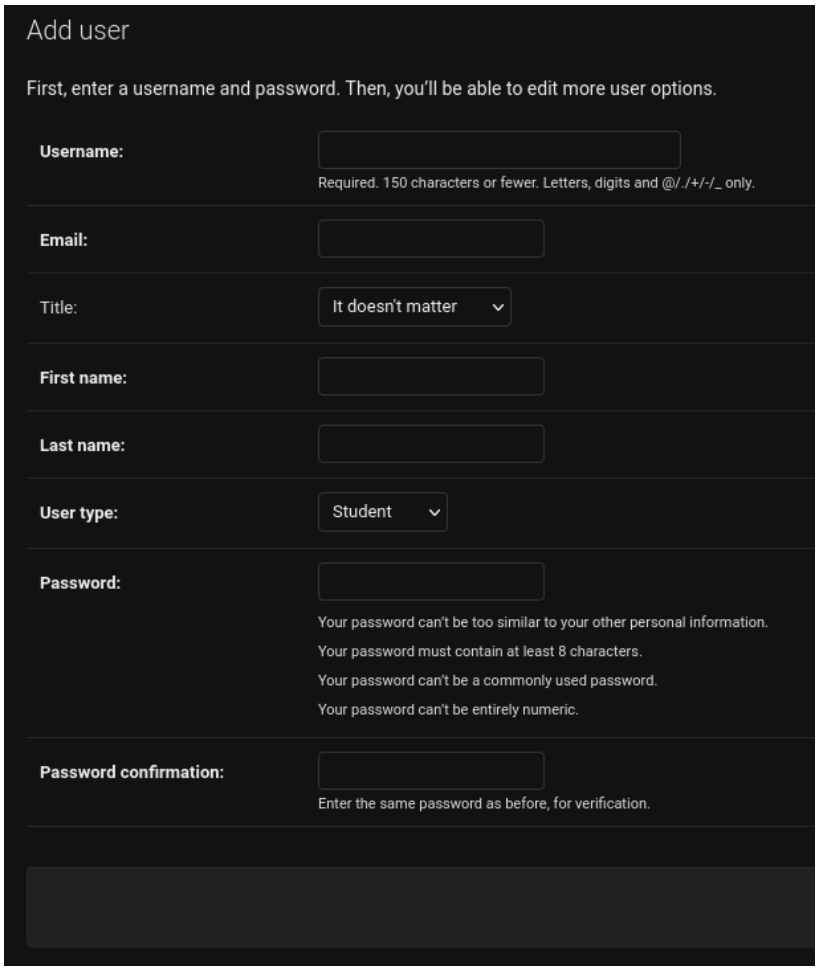


The start date and time can easily be modified by changing this field and clicking SAVE.

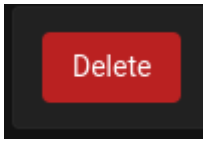
5.2.1.7.2 Adding / removing users

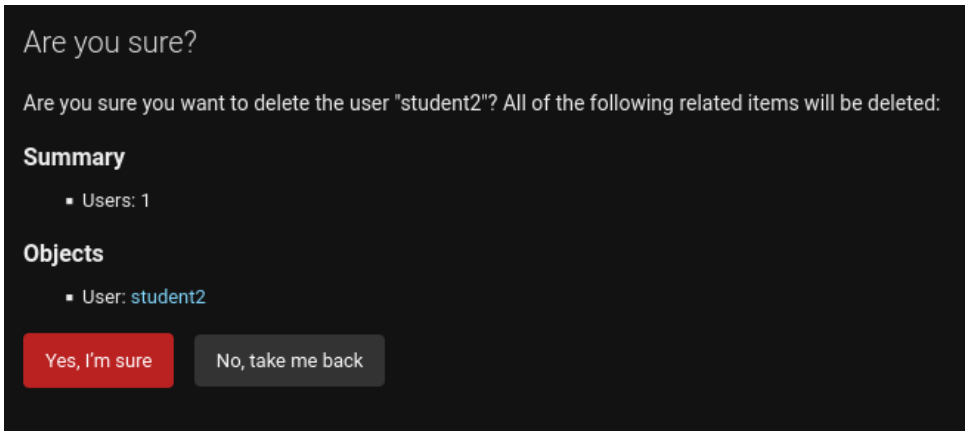
Many users have been added for testing purposes, so this interface has already been tested.

Users can be added with the following form:



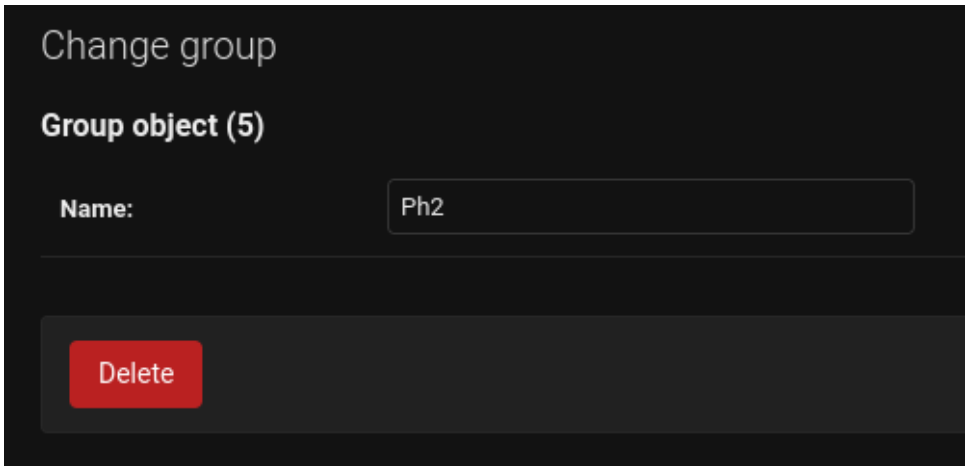
They can be deleted by clicking delete:



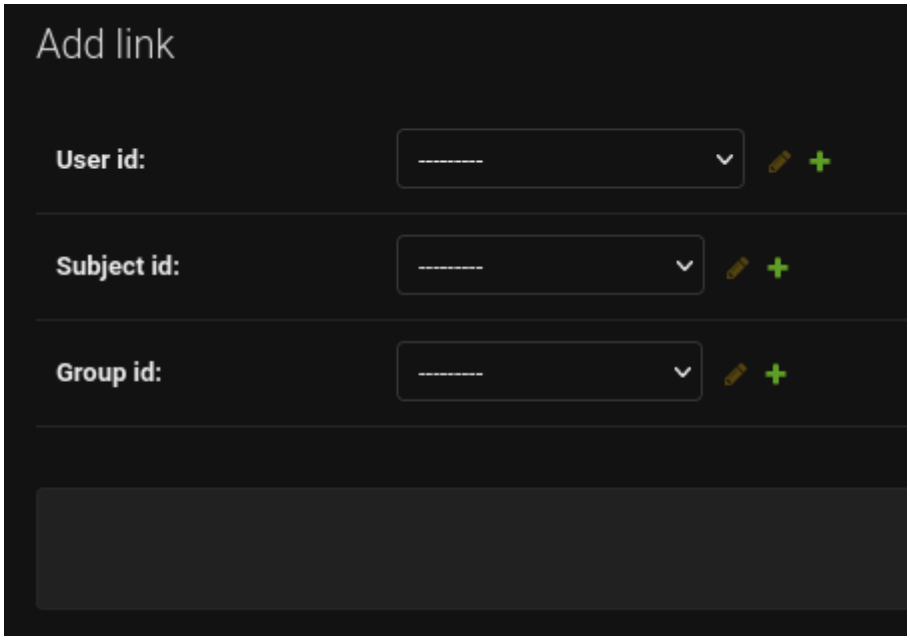


5.2.1.7.3 Changing classes

Classes can be modified as shown:



Students can be moved between classes by removing the record to link them to their old class, and adding a new record to link them to their new class.



5.2.1.7.4 Conclusion

This interface provides full control over the timetabling process, including all three factors specifically mentioned in the success criteria. This requirement has been fully met.

5.2.1.8 Requirement 8 – Even Allocation Between Classes

8	The timetabling algorithm must ensure that, over a given academic year, each subject gets approximately	Over the course of a year, no subject should differ more than 1% from its desired allocation.	The school has an obligation to provide a certain amount of time with each subject which must be fulfilled.

	the correct total lesson time, as input by the admin	Additionally, in an average day, the average student should have at least **44** 5-minute slots of lesson time (out of a 96-slot day). At least **10** days must be considered randomly to produce an average. Lesson lengths can be assumed to be randomly and uniformly distributed.	Teachers must be able to plan based on a given amount of available teaching time. Calculations are available below (marked *)

5.2.1.8.1 Even Allocation (over the course of a year)

This was implemented into the cost function, however will only be evident as the school year progresses. Testing the algorithm for a full year would be infeasible as it would take too long.

As this has not been tested, the requirement cannot be considered as met.

This must be tested further post-development.

5.2.1.8.2 Allocating Enough Lesson Time

Part of this requirement also entails allocating enough time overall.

The final timetable consistently achieved over 35 units of lessons, sometimes achieving 37-38. This is below the required **44**.

This is a significant issue. The final timetable did not achieve full capacity for the school. This makes a big difference as more classes are required to schedule the same number of lessons, so the school will need more teachers. While students and teachers should be more productive with the time available, this will not outweigh the loss in lesson time due to inefficient scheduling. The timetabling algorithm is not suitable to be used in a school environment until it is able to schedule a school at full capacity.

This also means it will not be possible for it to achieve the desired number of hours for each subject over the course of the year, therefore the even allocation requirement can't have been met (without the algorithm making compromises such as lots of clashes).

5.2.1.8.3 Conclusion

While smaller schools may not have an issue with the program, it is unable to fully schedule for a larger school environment. Therefore, the requirement is only partially met.

5.2.1.8.4 Addressing in Further Development

This could be addressed by further tweaks to constants and improvements to the timetabling algorithm. Improvements to the timetabling algorithm are considered further in **5.5.2**.

This also applies to requirements **10**, **11** and **15**, which can be addressed in further development by improving the quality of the timetabling algorithm.

5.2.1.9 Requirement 9 – Speed of Timetabling

9	The timetabling algorithm must run fast.	The timetable for the next day to be timetabled (in two weeks) must be complete by 8am. The algorithm can begin as early as is necessary, but not before 6pm on the previous day.	If the algorithm does not complete by the start of the school day, students will not be able to view their upcoming timetable.

The requirements specification allow 14 hours to schedule a single day of lessons. The algorithm took 22 minutes to produce a full two weeks of lessons for a school with 30 students.

The complexity of this algorithm is very difficult to estimate as genetic algorithms are complex and there are many factors involved in achieving a solution that converges.

- Suppose the complexity was O(n), where n is the number of students in the school, the school would have to have 11411 students for the requirement to fail.
- Suppose the complexity was $O(n^2)$, the school would need to have 585 students for the requirement to fail.

Large schools may be a possibility for the future of the product however has not been a goal for this project.

This will be considered further in the optimisation section in **5.5.2**

This requirement has been fully met.

5.2.1.10 Requirement 10 – Minimising Clashes

10	The timetabling algorithm must minimise teacher / student clashes.	The algorithm can be run with sensible scenarios. No clashes should be observed. Given more challenging scenarios, few clashes should be observed.	Teacher clashes are very bad as the lesson cannot be taught without a teacher. Student clashes are bad as students will miss out on part of their learning.

The final timetable produced has a few student clashes. The final timetable has no teacher clashes.

While the success criteria does allow for a few student clashes, this will not be considered fully met as these clashes do reduce the productivity of the school below that of the current timetabling system.

This requirement has been partially met.

5.2.1.11 Requirement 11 – Timetabling Algorithm

11	The software must schedule lessons for a sixth form with up to 100 students in each year group, each choosing 3 or 4 subjects out of up to 15 subject choices.		Core functionality *NOTE: Classes will be predetermined and input into the algorithm*

This software lists the functionality that the timetabling algorithm must achieve. Breaking this down into separate parts:

Aspect	Comments
Schedule lessons for a sixth form	This feature is present. Requirement met
With up to 100 students	It was tested for 30 students. There is no data to suggest it would not work equally well for 100 students. Partially met.
Each choosing 3 or 4 subjects	Only 3 choices was tested. 4 subject choices makes timetabling significantly harder, however few students will opt to do this. Partially met.
Out of up to 15 subject choices	15 subject choices is approximately double what was tested. The impact of this on the quality of the timetables remains unknown. Partially met

While the software does provide the ability for all of these criteria to be met, the impact on the other success criteria (regarding the quality of the timetable) remains unknown.

These criteria have not been tested as producing the test data would be too time consuming. The impact on the quality of timetables will likely not be favourable.

This requirement has been partially met.

5.2.2 Desirable Aspects

These requirements are not essential features of the solution (however they will improve the solution)

5.2.2.1 Requirement 12 – Lesson Topics

#	Requirement	Success Criteria / Test	Justification / Notes
12	Teachers could be able to input a topic for each lesson to identify what each lesson will be about		This helps teachers plan ahead and organise what they are planning to teach in each lesson.

The required functionality is present.

The requirement has been met.

5.2.2.2 Requirement 13 – Modifying Topics

13	Teachers could be able to modify the topic for lessons they have already scheduled		This would help correct typos or change their plan. *NOTE: Modifying the duration should not be supported, as this interferes with timetabling*

5.2.2.2.1 Comments

This functionality was not added to the final solution as it was not deemed high priority. While it would improve the product, the product will still function without this functionality. It was more important to focus on core aspects of the product such as the timetabling algorithm.

This would be implemented in future development of the product.

This requirement has not been met.

5.2.2.2.2 Addressing in Future Development

Future development of this functionality will not be challenging. It will simply require:

- Adding a link on each lesson to redirect to an 'edit' screen. The link should have a URL parameter corresponding to the ID of the lesson to be modified.
- This form can be created with Django in a very similar way to the creation form, with only minor tweaks (to only include the relevant elements)
- The page containing the form must ensure the user is logged in with the decorator `@login_required`
- The form will then properly allow users to update the lesson topics, and all necessary validation will be carried out both client-side and server-side.

In total, this feature will require less than an hour of development time to implement.

5.2.2.3 Requirement 14 – Cancelling Scheduled Lessons

14	Teachers should be able to cancel lessons they have scheduled accidentally		Accidentally scheduled lessons would be inefficient and would likely waste teacher / student time if they have to be carried out at an undesired length, or the admin's time if it is to be cancelled.

5.2.2.3.1 Comments

This functionality was not deemed high priority so did not make it in the final solution.

This requirement has not been met.

5.2.2.3.2 Addressing in Future Development

As with requirement 13, implementation will be easy. Implementation will entail:

- Adding a delete button beside each lesson
- Adding a delete button on the form to modify lessons (this improves usability as it means users can easily delete the lesson even if they have already clicked on it, or do not see the delete button initially)
- Programming the delete buttons to, pending authentication, delete the relevant lesson.
- Ensuring that lessons that have already been scheduled to a time slot cannot be deleted.

5.2.2.4 Requirement 15 – Balanced Timetable

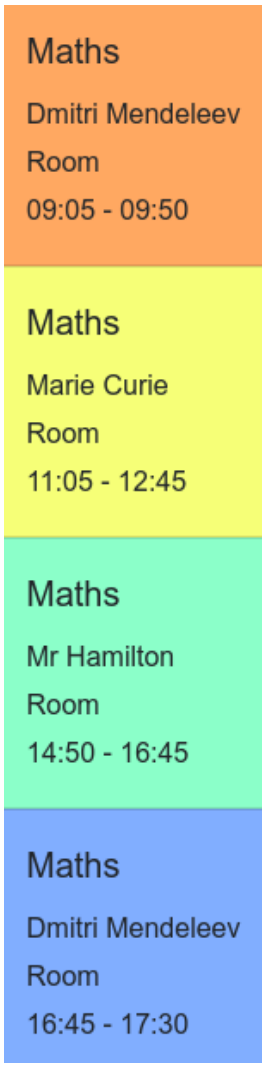
15	The timetabling algorithm should deliver a balanced timetable that is conducive to productive work, where lessons are spread out appropriately according to sensible criteria.	This requirement is vague and very difficult to test. The best way of testing would be to take sample outputs and judge whether it seems like a good balanced timetable.	The principal objective for the project is to improve productivity. The algorithm must achieve this. *NOTE: The criteria are decided in* ***3.4.2.1***

5.2.2.4.1 General Comments

The timetables produced are not bad, however not optimal for productive work. There are multiple instances of lessons not being properly spread out (e.g. having lots of one teacher in a given day) and other scenarios which are not conducive to productive work.

5.2.2.4.2 A Specific Example

The following timetable will be used as an example:



5.2.2.4.3 Evaluation

This timetable is good because:

- Lessons are spread out across the day
- There are a variety of teachers

However, it is suboptimal because:

- There is a 15-minute gap between the first and second lessons
- There is no gap in between the penultimate and final lessons
- The finish time is slightly later than desired

The next day includes 6 lessons, which suffer from many of the same issues as before. However, this timetable is not bad and is somewhat conducive to productive work. It is not significantly worse than a timetable produced with the traditional method.

This timetable also schedules one teacher twice. It immediately became clear that this student only has three teachers when they should have six, so this is making timetabling far more difficult. For the algorithm to reach a valid solution with more than three lessons, there must be a situation where a teacher is repeated on a given day. Instead, the test data should include six classes for each student,

one for each teacher. This will have reduced the quality of the final solutions, and should be considered in context of the results of timetabling. Creating an entirely new set of test data and re-doing the tests is not feasible.

This requirement has been partially met.

5.2.2.4.4 Addressing in Future Development

As mentioned previously, this can also be addressed by improving the timetabling algorithm.

5.2.2.5 Requirement 16 – Unexpected Changes

16	The software could notify students if their timetable has changed within the next two weeks		Students should be able to rely on the timetable for the next two weeks remaining constant. *NOTE: Timetables could change for unexpected reasons, such as teacher absences.*

5.2.2.5.1 Comments

While the admin interface allows for changing of lessons at any time, it does not notify students of these changes.

The software does not currently allow for rescheduling due to teacher absences. This would be a valuable addition for future development that would further increase the productivity of the school. This was not prioritised during the project as it is not core functionality – the product will function adequately without this functionality.

This requirement has not been met.

5.2.2.5.2 Addressing in Further Development

Adding in the ability to notify students directly would pose a more challenging implementation, as it requires either:

- Notifying users through their browsers. While possible, this is not commonly used and can be unreliable.
- Developing an app, which can then send notifications. This would be ideal, however requires significantly longer to develop.
    - An app would also be beneficial as some students would find this easier to view their timetable. It could also support functionality such as widgets so students can see their timetable from their home screen without even having to open an app. However, this is significantly lower priority than the rest of the project.

It also requires improving the admin interface to make it easy to reschedule lessons at the last minute in a way that notifies students and teachers accordingly.

While entirely possible, this requirement will be time consuming to implement.

5.2.3 Potential Extra Features

These features are by no means required. While they will improve the solution, the solution will function adequately without any of these features.

5.2.3.1 Requirement 17 – Class Allocation

17	The software could be able to assign students to classes	Requiring a third-party solution may cost money and is inconvenient. If done in the solution, it can be customised to minimise clashes throughout the year.	There are already many solutions on the market offering this functionality. Optimal class allocation to minimise clashes still appears to be a traditional block-based method.

5.2.3.1.1 Comments

Class allocation was explored as a possibility but did not form part of the final product.

More information regarding this decision can be found in **3.3.6**.

This requirement has not been met.

5.2.3.1.2 Addressing in Further Development

This functionality was designed in detail throughout **3.3**, and can be implemented according to how it is designed in that section.

5.2.3.2 Requirement 18 – User Registration

18	User registration / password management	If users could register themselves, the head of timetabling would not have to manually input each user.	This can be achieved on the admin interface. Data input by the admin will be more reliable and less vulnerable to malicious activity.

5.2.3.2.1 Comments

User registration is currently done exclusively through the admin interface.

A separate interface to allow users to register themselves would take time to implement (which is discussed further below).

This requirement has not been met.

5.2.3.2.2 Addressing in Further Development

Django has very good support for this functionality and implementation should be very easy. The default template for a registration form can be modified slightly and styled appropriately, which can then be made a part of the website.

Excluding any validation, this will take under an hour to fully implement.

However, this is open to abuse as it allows people to create many accounts which could disrupt the system. In reality, there will need to be some system that allows each student / teacher to only create one account. This will have to be researched further and consulted with schools to ensure this integrates well with the schools' systems.

Implementing with proper validation will likely take longer.

5.2.3.3 Requirement 19 – All Year Groups

19	Timetabling for all year groups (7-11)	It could be rolled out across schools with more year groups.	As the system needs 100% uptime, a limited rollout for sixth form colleges would help to test the system to ensure it works properly.

5.2.3.3.1 Comments

Timetabling for all year groups has not been tested. In other year groups, students choose many more subjects but have less time with each subject. It is not known how this will affect the performance of the algorithm as it has not been tested.

As no specific work has been dedicated to adding this functionality, the requirement can be assumed to not be met.

5.2.3.3.2 Addressing in Future Development

As with the rest of the timetabling algorithm, this will be satisfied by extensive testing and tweaking of the algorithm.

It may also be necessary to add more rules to timetabling to satisfy specific schools' needs. These will need to be determined by contacting schools to find out their requirements.

Implementing each school's individual requirements will be time consuming, as will improving the algorithm. However, this is crucial for the product to be successful.

5.2.3.4 Requirement 20 – Room Allocation

20	Room allocation	Lessons must take place in rooms. These need to be allocated to everyone knows where to go.	This is not a focus of the project. Room allocation can be done randomly (however this will lead to lots of moving around site)

5.2.3.4.1 Comments

Room allocation was deemed outside of the scope of the project early on in development.

This requirement has not been met.

5.2.3.4.2 Addressing in Further Development

Room allocation will be easy to add into the algorithm, simply requiring an additional constraint in the cost function. However, this will reduce the quality of solutions given by the algorithm.

The constraint would be added in a similar manner to the teacher and student clashes, where a fixed number of points is added for each time slot worth of overlap. It would likely be deemed of a similar priority to that of a teacher clash, as the lesson is unable to take place without a room (although there may be another room available to move to).

After the constraint is added, further optimisation and tweaking will need to be done to improve the results of the algorithm. This is discussed in **5.5.2**.

5.2.4 Conclusions

5.2.4.1 **Success Criteria**

The final product must satisfy all the following requirements:

#	Comments	Met?
1	Functionality was present	Fully
2	Both attacks prevented	Fully
3	All three tests passed	Fully
4	Safari could not be tested. Other browsers worked with no issues	Most likely
5	All three tests passed	Fully
6	Functionality was present	Fully
7	All three features present	Fully
8	Timetable does not achieve full capacity	Partially
9	Ran significantly faster than required	Fully
10	Some clashes are present in the solution	Partially
11	The test data was not as challenging as this requirement	Partially

It would be desirable if the final product satisfied the following requirements:

#	Comments	Met?
12	Functionality was present	Fully
13	Functionality was not present	No
14	Functionality was not present	No
15	Timetable is somewhat balanced and conducive to productive work	Partially
16	Functionality was not present	No

Should there be sufficient time, the final product could include:

#	Comments	Met?
17	Functionality was not present	No
18	Functionality was not present	No
19	No testing conducted	No

20	Functionality was not present	No

5.2.4.2 Analysis

All the requirements that the product must satisfy have either been fully met or partially met. The 'desirable' features were generally not implemented, as this time was dedicated towards improving the timetabling algorithm. None of the 'potential extra features' were included as the project was time constrained.

This shows that the development time allocated exceptionally well to the key areas of the project.

With more time, it would have been possible to achieve more of the requirements. However, all core functionality was successfully implemented, and all key requirements were either fully met or partially met. Overall, within the time available, the project was very successful.

5.3 User Testing

5.3.1 The Need for Stakeholder Testing

While the tests conducted so far are sufficient to conclude the project has satisfied the success criteria, it would still be valuable to allow users to use the product themselves. This is because:

- Some requirements that would make the product successful in a school may have been missed from the success criteria
- Using people who have not yet seen the code or the product is a great way to test the effectiveness of usability features
- Users are often not afraid to try to break features and test the product for robustness

5.3.2 Outline of the Tests

Users will be given the chance to use both the student and teacher interfaces and will be encouraged to attempt to break the program. If these issues are fixed now, the program will be more robust upon implementation in a school environment.

Users are not given any specific guidance as to how they are to use the program and are encouraged to experiment and use the software in any way they choose.

5.3.3 User 1

This user is a current student. The user was involved in testing requirement **3**, so is already somewhat familiar with the student interface.

5.3.3.1 Student Interface

They first logged in to reach the timetabling page. Having already used this interface, they quickly moved on to trying to break it.

First, they repeatedly clicked the back arrow to go back by a number of weeks. They then proceeded to click the forward arrow until the lessons began to appear again. After continuing to click these buttons, they became lost. They were confused as to why the month was not visible, and had no idea which month they were viewing the timetable for.

5.3.3.2 Teacher Interface

Upon assuming the role of a teacher, they first navigated to the timetable before quickly navigating back to home. They then navigated to the screen to schedule lessons and was able to very intuitively create the lesson. However, they were only using the predefined buttons to choose lesson lengths. It was possible they were considering these predefined buttons as the only options. When asked, it was clear that this was not the case and they were very aware of the ability to type any duration into the input box. When doing so, they tried to schedule a lesson that was 69 minutes long, however they were informed that it has to be a multiple of 5. This error message was clear and informative, immediately conveying the necessary information to the user.

After navigating the interface further, they commented that there should be a “back to home” button at the top of the list of scheduled lessons, rather than simply at the bottom. There were many lessons scheduled here which will not be the case for most teachers, however this comment is relevant for teachers who choose to schedule many lessons in advance. It may be more intuitive if the options were at the top instead of at the bottom of the list of lessons.

They were further invited to try and break the interface, however the attempt was unsuccessful.

5.3.3.3 User’s Comments

The user was given a chance to share their own comments regarding the program. Their comments are as follows:

“It’s a very intuitive and fairly simple to use software and I can see how it would be very useful.”

They were probed to ask about any features they did not particularly like

“I suppose the text could be a bit bigger – this whole interface [points to teacher homepage].”

They did not manage to think of any other points in the next few minutes.

Overall, they were very happy with the program and could see themselves using it to view their timetable in a school.

5.3.3.4 Conclusions

Overall, the interface held up very well when navigated by a student. The main conclusions are as follows:

Issue	Changes
Clicking the arrows many times allows you to move far away from the current date. The month is not displayed so users do not know which day they are viewing their timetable for.	Either: - The interface could restrict users to only move backwards or forwards by two weeks - The interface would have to display the month in some way
If there are lots of lessons, it can be unintuitive to have to scroll down to the bottom	'New Lesson' and 'Back to Home' buttons at top of teacher scheduled lessons instead of at the bottom
The teacher homepage does not make full use of the available space	Text could be increased in size

While it is the evaluation stage, these issues are easy to fix (if appropriate).

5.3.3.4.1 Fixing Issue 1

As the timetable is only displayed two weeks in advance, it would make sense if users were only able to navigate two weeks ahead. Equally, displaying the month would clutter the interface and would not look good.

It is important that this is clearly communicated to the end user – the arrows must be greyed out or otherwise appropriately marked as disabled (instead of just doing nothing but appearing available).

A variable was created to store the number of weeks difference between the current date and that being viewed.

```
weeks_diff = math.floor((datetime.datetime.utcnow() - current_date).days /
7)
```

Note that current_date is set to the current date being viewed, rather than the actual date.

Crucially, this is rounded down so that it never obscures any lessons, and users are always able to see at least two weeks, rather than at most two weeks.

Conditions were added such that the link will be passed into the template as `'disabled'` if it should not be rendered.

```
if weeks_diff <= -2:
    weekdays_links = {'<': 'disabled'}
else:
    weekdays_links = {'<': math.floor((weekstart -
datetime.timedelta(days=3)).timestamp())}
```

```
if weeks_diff >= 2:
    weekdays_links['>'] = 'disabled'
else:
    weekdays_links['>'] = math.floor((weekstart +
datetime.timedelta(days=7)).timestamp())
```

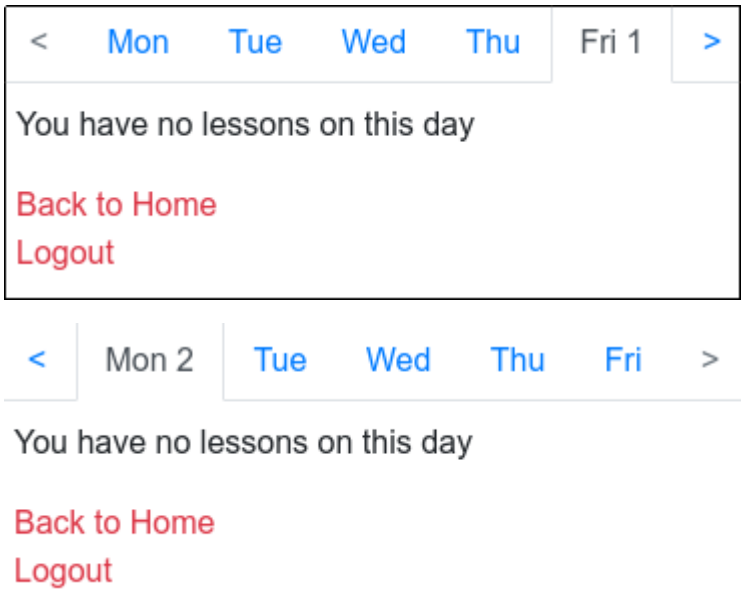
Within the template, it checks to see if the link is `'disabled'`. If so, the link is set to a dead link and the element is rendered as disabled.

```
{% if link == 'disabled' %}
<a href="#" class="nav-link disabled px-0">{{day}}</a>
{% else %}
<a href="?day={{link}}" class="nav-link px-0{% if weekday_format == day %}
active{%endif%}">{{day}}</a>
{% endif %}
```

After clicking the previous week arrow twice, it disabled the arrow corresponding to the next week. Likewise, going forwards two weeks had the opposite effect that was intended. This could be fixed by simply negating the result of the calculation:

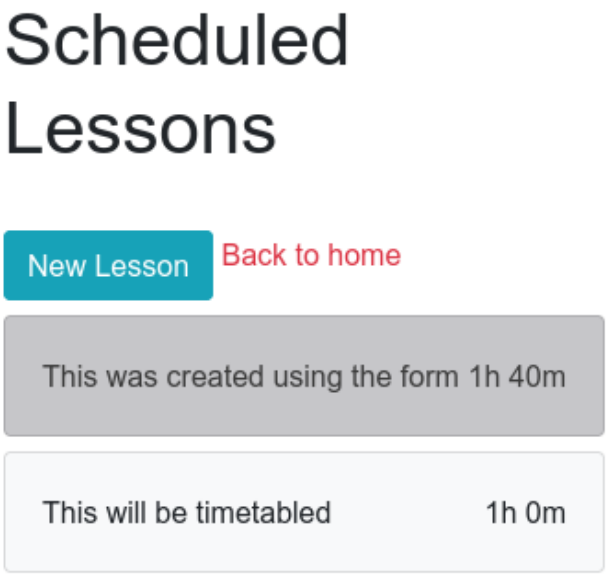
```
weeks_diff = math.floor((current_date - datetime.datetime.utcnow()).days /
7)
```

The code now worked:



5.3.3.4.2 Fixing Issue 2

This can be fixed easily by simply moving the code to the top in `scheduled_list.html`:



The new copy of the file is available at the end of the document, with the rest of the code.

5.3.3.4.3 Fixing Issue 3 – Increasing Font Size

While the user commented this could be an improvement, upon further consideration this is not an issue. It is standard for websites to have empty space (if not needed), rather than increasing the font size to compensate. Increasing the font size will leave very large text and will not be as easy to navigate as one that is small and only uses up the necessary space. The font is already large enough to be viewed even by those with poor eyesight *(note that screenshots in this project have often been resized)*.

This will not be deemed an issue and hence will not be changed.

5.3.4 User 2

The next user is of a higher age and, while they are not a teacher, they can represent the technical ability of most teachers.

5.3.4.1 Student Interface

Initially, they found the interface confusing. They did not understand where their lessons were scheduled and that there was nothing on much of the right-hand side of the interface. The key misconception was that they expected the interface to show the lessons under each day (such as in **3.2.2.2.1**), however instead there was only one list of lessons present.

However, the interface is best viewed in mobile view, so this was subsequently changed (the test was still conducted on a computer, but using the iPhone 11 Pro screen size). This immediately made sense and the user recognised the role of the top bar in terms of selecting the day to view lessons for. They also quickly understood that the arrows navigate between weeks. They were able to competently navigate their timetable with no issues.

5.3.4.2 Teacher Timetable

As a teacher, they navigated their timetable in a very similar way and was able to understand the interface. They were confused by the presence of placeholder data such as in lesson names and all subjects being displayed as maths. They also commented some of the lessons were very long, however were informed that they would have chosen the lesson lengths. They suggested that a 2-hour lesson is too long, however did accept that it may be appropriate for subjects such as DT depending on the circumstances. They were able to quickly understand the timetable as with the student interface.

5.3.4.3 Listing Scheduled Lessons

Initially, they were confused about the role of this interface. However, after being properly informed about the solution, they understood the interface much better.

They commented that clicking on scheduled lessons did not do anything and they expected it to highlight it. They stated that it should be possible to click on lessons to update them, such as changing the name or duration. This was planned but ultimately removed from the scope of the project, to which they were informed.

They did not initially understand that a lesson being dark grey meant that it had already been scheduled.

After understanding more about how the solution worked, they were able to understand and navigate this interface.

5.3.4.4 Scheduling a Lesson

They were very positive about this interface. They said the interface is ‘very intuitive’, and they were able to schedule a lesson with ease.

5.3.4.5 Comments

When asked for further comments, they commented that is has a ‘nice look and feel’.

They did not have any further comments.

5.3.4.6 Conclusions

Initially, they had more difficulty with understanding some aspects of the interface than the student. However, after using the interface for a short period of time, they found it to be intuitive and easy-to-use.

Potential issues are as follows:

Issue	Changes
The desktop timetable interface can be confusing	As already planned, with sufficient time, it will be changed into a full week view

It should be possible to modify lessons that have already been scheduled	As planned, with sufficient time, this will be addressed

Clearly, both of these issues are already planned for the future development of the product. These will be addressed in future, as planned in **5.5**

5.4 Usability Features

While there is no definitive list of usability features created as part of the project, many were created and have been very successful (as shown by the user testing).

Some examples of usability features are detailed below:

Usability feature	Evaluation	Successful?
Use of colour to distinguish UI elements	Very effective. Colours are clearly assigned, such as: - Blues are assigned to neutral buttons - Greens are assigned to successful actions - Reds indicate warning, such as logging out Importantly, these are kept consistent throughout the interface. While these colours may mean different things to different users, they are used in a way that is consistent across the site and between other websites.	Yes
Use of colour to distinguish lessons in the timetable	This is done effectively to make the timetable look better, and for lessons to stand out. However, a unique colour was not assigned to each lesson. This has potential to further improve readability as users will be able to scan over their timetable and glance for specific lessons.	Mostly*
Headings at the top of pages (as well as `<title>`s)	These clearly define the purpose of a given page and ensure a user is confident they are on the correct page.	Yes
Navigation links (e.g. 'back to home')	These ensure it is possible to navigate throughout the site effortlessly without having to modify the URL in the address bar or click on direct links.	Yes
Labels for input elements	These labels are read out to those using screen readers upon clicking on the form. This is very helpful to ensure those relying on assistive technologies are still able to access the site.	Yes

There are many other usability features used as part of the solution. All of these features were very successful (none were unsuccessful), so are not worth detailing here.

Overall, the usability features present in the solution were very successful. As determined in the user testing, users were able to navigate the interface with ease and did not have a problem with a lack of usability features.

5.4.1 Addressing in Future Development

The colours for a given lesson on a given student's timetable could be generated with the following algorithm:

- Sort their classes / groups by primary key
- Let i = 0
- For each group:
    - Assign colour $(i \mod MAX) + 1$ to this group
        - where MAX is the maximum number of colours available
        - *colours start at 1, not 0*
    - Increment i

This has a number of advantages:

- The colours with a low index are used preferentially. These are defined to be the best / highest contrast colours
    - No colours are skipped in this order
- The colours remain consistent when the user refreshes the page or comes back later to view their timetable
    - The ordering also remains consistent if the user is assigned to newly created groups (groups created after the most recent group the user is already a member of)
- The colour of each group is consistent between different days on the timetable
- It is very quick to compute and does not require any space in the database

However, there are also a few disadvantages:

- It is not consistent between students
    - This is not an issue, as students do not need to be able to quickly identify their group on another student's timetable. Ensuring it is consistent would require significantly more colours
- The colours can change if:
    - The user is removed from a group (except for the last one)
    - The user is assigned to a new group (that has been created before the most recently created group they are already a member of)
    - This could be annoying if they are used to certain colours
    - However, this is not a significant issue as they can quickly adjust to new colours

The implementation of this algorithm would be very easy.

Then, the assigned colours must be passed into the template, which can then change the CSS classes appropriately to render the correct colours.

Overall, implementation of this functionality will take less than an hour.

5.5 Future Development

5.5.1 **Timetabling for a Real School**

While the current solution is good, it is not suitable for use in an actual school. Schools require 100% uptime and often have unique requirements that have not yet been implemented as part of the developed solution.

In addition to improvements to the algorithm, to roll out the product in a school, the following will have to be achieved:

Feature	Justification
Modifying the topic of scheduled lessons	This helps teachers manage their lessons and change their plans / fix typos
Cancelling already scheduled lessons	If teachers schedule a lesson accidentally, they will be forced to teach it at a suboptimal duration
Unexpected changes – notifications	The core concept has the capability to deal with short notice changes such as teacher absences. For these to be effective, they must be communicated to students.
Part-time teachers	A few teachers are part time, so the software must be able to accommodate this
Class allocation	It is not ideal having to use a separate program to allocate classes. A custom-made program will be able to better optimise classes for the specific timetabling algorithm in use
User registration	It is not appropriate for the administrator to be typing in users' passwords
All year groups	Timetabling for more than just sixth form will allow a much wider market to be reached
Room allocation	Lessons are conducted in rooms, which also need to be optimised (particularly for large schools)
Music lessons / other commitments	While the current timetable does not offer a solution for this, it would be possible to minimise the clashes with lessons and music lessons to minimise disruption to learning
Lunch times	Lunch timings can be scheduled around lesson times to ensure everyone gets lunch at a sensible time and people don't have to wait too long after their lesson finishes to get lunch
Fixed lessons (e.g. PE)	This would help to avoid students forgetting kit (and remove the easy excuse)
Desktop timetabling interface – week view	This interface displays more information at once and is more intuitive

Additionally, the software will have to pass rigorous uptime / reliability / stress tests using actual school data before it can be used in a school.

5.5.2 Improving Timetabling

5.5.2.1 Tweaking the Algorithm

The results of a genetic algorithm are highly dependent on the initial conditions, and there was not sufficient time during the project to fully test the impact of changing each variable and determine the optimal values to produce the best timetables. As has been done already, it is essential to try changes to the values and determine whether they improve the resulting timetables or not, and choose to accept or reject the changes appropriately. It is more difficult to conduct a theoretical approach as there are many variables that affect the result of the algorithm in unpredictable ways, so testing would be the best approach.

While this could be done manually, it would be possible to write an algorithm that automates this process. This algorithm would effectively be solving an optimisation problem for an optimisation problem. While it is possible that a genetic algorithm could offer a good way of optimising the results of other genetic algorithms, this would run slowly so it may be better to take a simpler approach. It could be pre-programmed a few different values to try for each constant, and the algorithm would be automatically executed with each combination of values to determine the one with the lowest cost value. This process could be repeated, likely running overnight, to improve the results of the algorithm more quickly.

5.5.2.2 Optimisations

While the algorithm takes about 2 minutes per day to timetable a school of 30 students, larger schools remain unknown. It is always beneficial to reduce the runtime of the algorithm to reduce unnecessary power consumption and allow for less powerful hardware to be viable (this test was executed on a premium-midrange consumer-grade CPU that satisfies the hardware requirements, a Ryzen 7 3700X).

5.5.2.3 Parallel Processing

Timetabling was done on an 8-core CPU. Unless specifically programmed, Python runs on a single CPU thread. This means that the program was only ever using up to 1/8 of the performance of the CPU (ignoring thermal limitations of using all cores and other inefficiencies of parallel processing).

If ran on server-grade hardware, many more cores are often present. It would be beneficial if all hardware made available to the program is used appropriately to make the timetabling process happen as quick as possible.

In Python, it is possible to divide the process into subprocesses, which can then run independently on separate cores. This can be done with the `multiprocessing` module in Python.

The most common approach to parallelising a genetic algorithm is to run the algorithm multiple times on separate populations, and choose the best result at the end. This essentially means that all 10 iterations that are currently occurring sequentially could be occurring at the same time. This will significantly improve performance, if required in a large school.

While this will scale very well to an 8-core CPU, scheduling across 32 or 64 cores will quickly lead to diminishing returns as it is unlikely a better solution will be reached by running more iterations of the algorithm. Therefore, the quality of the solutions does not scale proportionally with the number of cores so it may not always be valuable to allocate more cores to the process.

Implementation of this functionality would not be difficult, as it simply needs to call the algorithm multiple times in separate subprocesses (which is all done using the multiprocessing module). As they can be treated entirely independently, there is no communication between them which simplifies the implementation massively.

5.5.2.4 Optimising the Cost Function

5.5.2.4.1 Introduction

The cost function is computed many times throughout the execution of the algorithm, so it is vital that it runs quickly.

5.5.2.4.2 Methods of Optimisation

Different ways this could be optimised include:

- Removing constraints that do not contribute much to the final solution
- Simplifying constraints (e.g. only consider teachers then multiply by some constant, instead of considering all users)
- Simplifying the functions used to model the number of points to add to the cost function
- Pre-computing / caching the results of some calculations (e.g. if the timetable only changes slightly since the previous iteration, it does not have to be entirely recomputed)

Tweaking constants to improve the result of the cost function will also make solutions converge more quickly, consequently the cost function will not need to be called as much.

5.5.2.4.3 Caching

Currently, the cost function is entirely recomputed upon any small change to the timetable. It may be the case that this is the best implementation, however this is unlikely. Note that if caching is done ineffectively it will lead to many cache misses and the performance will reduce rather than increase. Caching also increases the space complexity of the algorithm, which could become an issue in a larger school (especially if parallelised).

One aspect that can definitely be cached is the generation of occupied time slots by user (stored as `schedules`). Each time a lesson is added or removed, the stored version can be modified to save the need to recompute each time. Note that if a lesson is removed, as clashes are only stored once, it must be checked if there are any lessons occurring at this time. It may give better performance if all instances of lessons occurring are stored (i.e. it stores 2 if there are 2 things occurring, rather than just 1). There should be very few instances where the cost function is not recomputed after modifying its list of lessons (hence the operation would have been redundant), so this would speed up the calculation of the cost function.

The number of clashes can be modified as and when the lessons are changed rather than computing this from the `schedules` dict each time. However, should there be an issue with a value, this issue will be propagated to all future iterations. This will make any bugs have more of an impact than they

would in the current implementation. This should be considered, but is not a major obstacle to its implementation.

Likewise, other values can also be stored and only changed as and when the list of lessons changes, such as:

- Lesson time allocated to each class / group
- Average lesson time per student
- Daily workload (only storing if above the threshold)
- Early finish time
- List of gaps (this will require more complex programming to compute the difference in gaps without computing the whole list of gaps, which may potentially outweigh the benefits gained)

It is yet to be determined how much of an improvement this will have on the execution time. It is likely that the first change, to `schedules`, will have the largest improvement. It is unlikely that these changes will have a significant improvement as they still need to be partially computed each time they are updated and they are all quick to compute anyway.

5.5.3 More Flexibility

The developed solution offers a very flexible approach, however there is one key aspect that is still taken from the traditional method of timetabling: classes. A fixed number of students are allocated to a class with a single teacher, and all lessons in that subject occur with that one teacher.

It is possible that, pending further research, a more flexible approach can be offered to this part of the timetable to increase productivity even further.

For example:

- Different bits of the course could be taught by teachers that enjoy them most
- Content suitable for delivery in lecture format could be delivered as such
- Students could potentially be given a choice regarding which lesson to attend based on their knowledge / understanding of a given topic, or other factors

The specifics regarding implementation of such a change will require significantly more research, however this is another thing that is made possible by switching to a more flexible approach.

5.5.4 Maintenance of the Solution

The solution is generally well structured and easy to maintain. Maintainability informed all stages of the development process.

5.5.4.1 Maintenance features

The solution is easy to maintain because:

- Functions are properly documented using Python docstrings

- Comments are given where appropriate to divide content and explain potentially confusing lines
- Comments were not used excessively as this makes the code more difficult to read
- Obscure Python syntax was avoided. However, when appropriate, a comment was added to detail what the code does (e.g. for-else)
- The code was structured logically, with classes and functions being used extensively, where appropriate.
    - Object-oriented programming uses abstraction and modularity to significantly improve the readability and maintainability of the code.
    - Reusable components reduce the amount of code and make it possible to change one aspect and for other aspects to also change.
    - File names were clear, and files were located in a logical place.
- Where appropriate, standard Django implementations were used (particularly for security-centric features). These will be automatically patched to deal with the latest security threats.
- Variable names were sensible, such that it is easy to understand what a variable contains but it is not too long to type.
- Naming conventions were adhered to so that any Python developer is able to easily understand different types of variables. For example:
    - Identifiers are in `snake_case`
    - Constants are in `ALL_CAPS`
    - Classes use `PascalCase`
    - `camelCase` is not used in Python
- Variable types were annotated, including function return types, where appropriate
- Where appropriate, code was broken down into functions/methods each containing only a few lines of code. This improves readability and aids debugging as problems can be quickly isolated to specific functions.
- Much of PEP 8 was adhered to, ensuring good code style.

5.5.4.2 Maintenance limitations

Maintenance may be difficult because:

- It is possible that comments may not have been used enough.
    - However, this is not clear as no other developer has yet attempted to understand the code as part of the project.
    - This was avoided to ensure there are not too many comments (having too many comments makes code more difficult to read)
- Dependencies were not documented in a `requirements.txt` file. This is because there are very few dependencies, however this will mean that it may take a minute for a developer to install the necessary libraries to allow the code to work.
    - There was no version information on the dependencies used. New versions of dependencies have the potential to break the project if not made backwards compatible.
- There is no file detailing the command to run the product. While this would be trivial for any user experienced with Django, it may be unclear for those with no experience working with Django. Therefore, it may have been beneficial to include a file that details how to run the program.

- There is no README file.

5.5.4.3 Evaluating Maintainability

Overall, maintainability was very good in the developed solution. Many standards for code style were adhered to and the code was logically structured. While it would have been beneficial to have a full list of dependencies and more documentation surrounding how to run the product, this would not be an issue for a developer with experience in Django.

The maintainability of the final product is excellent.

5.6 Conclusion

5.6.1 Comparing to the Traditional Method

This product seeks to compete with the traditional method of timetabling which is employed by many different pieces of timetabling software available currently. It would be very appropriate to compare against current implementations to decide whether the project has been successful or not.

Some advantages of the developed solution versus the traditional method are as follows:

- Variable length lessons increase productivity
- Timetabling is fully automated
    - Currently, it requires 7-8 hours work (according to Mr Chard)

Some disadvantages are as follows:

- The timetable does not yet schedule enough lessons to reach full capacity
- Some students / teachers will not enjoy the disruption to their routine

Overall, increasing productivity while fully automating the process is a massive gain, however this does not outweigh the fact that the timetabling algorithm simply does not schedule a school at full capacity yet. Once this issue can be resolved by further development and tweaking of the algorithm, the product will be a massive improvement over the traditional method.

5.6.2 The Primary Objective

5.6.2.1 Introduction

The primary objective of the project was to improve productivity in schools. If the product is to be considered a success, it must have achieved this objective.

However, measuring productivity is incredibly difficult. ‘Productivity’ is not even clearly defined, so designing an experiment to measure this and achieve a result is very difficult.

5.6.2.2 Measuring Productivity

Various factors can be considered to influence the productivity of a school:

- Grades attained
    - An increase in productivity will increase the results of the school
    - However, the difference in these grades will be minimal and difficult to distinguish from random variation
- Interviewing teachers / students
    - Teachers may notice they are able to progress faster with the course or have more time to spend on certain topics that they would otherwise have had to skip past
    - People may simply 'feel' more productive, without having any numbers to go alongside this result

Overall, it seems regular interviews with different kinds of users throughout the implementation will be appropriate to determine how much of an impact this solution is having on the productivity of the school.

5.6.2.3 Expected Results

Evidently, results will only be obtained after the solution is used in a school. However, it is possible to make approximate predictions.

Assuming the solution is able to timetable at full capacity:

- A few, but only a few, teachers will complain at the additional work having to schedule lessons and will fail to see the benefit. They will only see the disruption to their routine and not weigh this up against the productivity gain. These teachers may report up to a 5% decrease in productivity.
- Most teachers will take advantage of this and notice a 5-10% overall improvement in their, and students', productivity
- Some teachers will find this especially useful, reporting gains of 20-30%. They will be able to schedule completely different lesson lengths depending on what they are doing, or always schedule lessons of the same length (e.g. DT would find this especially useful, reporting that they are able to get significantly more done throughout a 2-hour lesson than two 1-hour lessons)

These predictions are based on the research conducted throughout the project (including interviews with students / teachers) and the current state of the solution.

5.6.2.4 Conclusions

Clearly, if these predictions are true, the project will be a massive success. However, these are only approximate predictions, so drawing any meaningful conclusions is meaningless.

5.6.3 Summary

Overall, the solution was excellent considering the time available. The final product satisfies all important features and was created with maintainability in mind to support future development. Time was allocated well to important areas in the project, which were almost entirely met.

While it is clearly not yet suitable for use in a real school, the developed solution is able to competently produce sensible timetables that minimise clashes and display these timetables through an intuitive interface.

The solution is innovative, disrupting the traditional timetabling method and offering the potential for a significant productivity boost in schools. The fundamental switch to a flexible approach in school timetabling leads to many productivity improvements, many of which have yet to be discovered.

After some further development, the final product will be very successful in a school environment and will enhance the education of many students for years to come.

6 The Code

6.1 Introduction

While comprehensive snippets are given throughout, the final iteration of the code is given in this section. The files are ordered in the same way as the file structure.

Some minor changes may not have been fully documented during development. These modifications will only be viewable in this section.

6.2 File Structure

Clicking on a file in the file structure will navigate to that file. Each file includes a return link to the file structure.

Items marked * do not have their source code listed here. Generally, this is because the file is either empty or has not been modified in any way from the pre-generated code.

The project resides within `/src/django_project/`

- django_project
    - __init__.py*
    - credentials.json*
    - settings.py
    - urls.py
    - wsgi.py*
- timetable
    - migrations*
    - static
        - timetable
            - CSS
                - Bootstrap files*
            - JS
                - Bootstrap files*
            - style.css
    - templates
        - timetable
            - scheduling
                - schedule.html
                - scheduled_list.html
            - base.html
            - home.html
            - login.html
            - logout.html
            - teacher.html

There are **19** total files with their contents accessible in this section.

6.3 Files

6.3.1 django_project

This directory contains anything not specific to the app ‘timetable’

6.3.1.1 __init__.py

*[Back to file structure: **6.2**]*

This file is empty

6.3.1.2 credentials.json

*[Back to file structure: **6.2**]*

This file has been excluded for security reasons

6.3.1.3 settings.py

*[Back to file structure: **6.2**]*

```
"""
Django settings for django_project project.

Generated by 'django-admin startproject' using Django 2.2.24.

For more information on this file, see
https://docs.djangoproject.com/en/2.2/topics/settings/

For the full list of settings and their values, see
https://docs.djangoproject.com/en/2.2/ref/settings/
```

```
"""

import os
import json

with open('django_project/credentials.json') as credentials_bytes:
    credentials_text = credentials_bytes.read()
    credentials = json.loads(credentials_text)

# Build paths inside the project like this: os.path.join(BASE_DIR, ...)
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))


# Quick-start development settings - unsuitable for production
# See https://docs.djangoproject.com/en/2.2/howto/deployment/checklist/

# SECURITY WARNING: keep the secret key used in production secret!
SECRET_KEY = credentials['django_secret']

# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = True

ALLOWED_HOSTS = []

from django.core.management.utils import get_random_secret_key
get_random_secret_key()


# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'timetable'
]

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 = 'django_project.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',
            ],
        },
    },
]

WSGI_APPLICATION = 'django_project.wsgi.application'


# Database
# https://docs.djangoproject.com/en/2.2/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}


# Password validation
# https://docs.djangoproject.com/en/2.2/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',
    },
]

AUTH_USER_MODEL = 'timetable.User'
```

```
# Internationalization
# https://docs.djangoproject.com/en/2.2/topics/i18n/

LANGUAGE_CODE = 'en-uk'

TIME_ZONE = 'UTC'

USE_I18N = True

USE_L10N = True

USE_TZ = True


# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/2.2/howto/static-files/

STATICFILES_DIRS = (
    'timetable/static/',
)
STATIC_URL = '/static/'

LOGIN_URL = '/login'
LOGIN_REDIRECT_URL = '/'
```

6.3.1.4 urls.py

*[Back to file structure: **6.2**]*

```
"""django_project URL Configuration

The `urlpatterns` list routes URLs to views. For more information please
see:
    https://docs.djangoproject.com/en/2.2/topics/http/urls/
Examples:
Function views
    1. Add an import:  from my_app import views
    2. Add a URL to urlpatterns:  path('', views.home, name='home')
Class-based views
    1. Add an import:  from other_app.views import Home
    2. Add a URL to urlpatterns:  path('', Home.as_view(), name='home')
Including another URLconf
    1. Import the include() function: from django.urls import include, path
    2. Add a URL to urlpatterns:  path('blog/', include('blog.urls'))
"""
from django.contrib import admin
from django.urls import path, include

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

6.3.1.5 wsgi.py

This file has not been modified

6.3.2 timetable

*[Back to file structure: **6.2**]*

This directory corresponds to the Django app 'timetable'

6.3.2.1 migrations

This folder contains temporary files that are used as part of development. They have been excluded.

6.3.2.2 static

6.3.2.2.1 timetable

6.3.2.2.1.1 CSS

This directory contains files for Bootstrap, which have been excluded

6.3.2.2.1.2 JS

The Bootstrap files in this directory have also been excluded.

6.3.2.2.1.3 style.css

*[Back to file structure: **6.2**]*

```
.login-width {
    width: 350px;
}

.scheduled {
    background-color: #c6c6c9 !important;
    color: #3b3b3b !important;
}

#duration-input-number {
    width: 62px;
}

#username-input {
    width: 267px;
}

#password-input {
    width: 270px;
}
```

```
.subject-1 {
    background-color: #ffa861 !important;
}
.subject-8 {
    background-color: #ffabac !important;
}
.subject-2 {
    background-color: #f6ff77 !important;
}
.subject-5 {
    background-color: #a4ff77 !important;
}
.subject-3 {
    background-color: #8bffc9 !important;
}
.subject-6 {
    background-color: #87eefc !important;
}
.subject-4 {
    background-color: #81aeff !important;
}
.subject-7 {
    background-color: #b49afa !important;
}
.subject-9 {
    background-color: #ffa7f7 !important;
}
```

6.3.2.3 templates

6.3.2.3.1 timetable

6.3.2.3.1.1 schedule.html

*[Back to file structure: **6.2**]*

```
{% extends "timetable/base.html" %}
{% block content %}
    <div class="text-center login-width m-auto">
        <h1 class="my-4">Schedule a Lesson</h1>
        <form class="text-left" method="POST">
            {% csrf_token %}
            {{form}}
            <input type="submit" value="Submit" class="btn btn-success">
            <a href="/teacher/scheduled" class="text-danger">Cancel</a>
        </form>
    </div>
{% endblock %}
```

6.3.2.3.1.2 scheduled_list.html

*[Back to file structure: **6.2**]*

```
{% extends "timetable/base.html" %}
{% block content %}
    <h1 class="m-4">Scheduled Lessons</h1>
    <a href="/teacher/schedule"><button class="btn btn-info ml-4 mt-2">New
Lesson</button></a>
    <a href="/" class="text-danger">Back to home</a>
    {% for lesson in lessons %}
    <div class="card my-2 mx-4 {% if lesson.fixed %}scheduled{% else %}bg-
light{% endif %}">
        <div class="card-body">
            <p class="float-left mb-0">{{lesson.topic}}</p>
            <p class="float-right mb-0">{{lesson.duration}}</p>
        </div>
    </div>
    {% endfor %}
    {% if not enough %}
    <div class="card my-2 mx-4 bg-warning">
        <h5 class="mx-2 mt-2">Not enough lessons scheduled</h5>
        <p class="mx-2">You must always have some lessons scheduled in
advance. If not enough are scheduled, 55-minute lessons will be
automatically scheduled.</p>
    </div>
    {% endif %}

{% endblock %}
```

6.3.2.3.1.3 base.html

*[Back to file structure: **6.2**]*

```
{% load static %}
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, height=device-
height">
    <meta name="keywords" content="timetabling, timetable, scheduler">
    <meta name="author" content="Joshua Humphriss">

    <title>
        {% if title %}{{title}} - {% endif %}Timetabling
    </title>

    <link rel="stylesheet" type="text/css" href="{% static
'timetable/style.css' %}">
    <link rel="stylesheet" type="text/css" href="{% static
'timetable/css/bootstrap.css' %}">
```

```
    <script src="{% static 'timetable/js/bootstrap.js' %}"></script>
</head>

<body>
    {% block content %}
    {% endblock %}
</body>

</html>
```

6.3.2.3.1.4 home.html

*[Back to file structure: **6.2**]*

```
{% extends "timetable/base.html" %}
{% block content %}
    <div class="container">
        <h1 class="text-center mt-4">Timetabling</h1>
        <p class="text-center mt-3 mb-4">Choose one of the options to
proceed to the appropriate interface</p>
        <div class="text-center">
            <a href="/student" class="w-100"><button class="btn btn-
primary">Student</button></a>
            <a href="/teacher"><button class="btn btn-
primary">Teacher</button></a>
        </div>
    </div>
{% endblock %}
```

6.3.2.3.1.5 login.html

*[Back to file structure: **6.2**]*

```
{% extends "timetable/base.html" %}
{% block content %}
    <div class="text-center login-width m-auto">
        <h1 class="my-4">Log in</h1>
        <form method="POST" class="text-left">
            {% csrf_token %}
            {{form}}
            <input type="submit" class="btn btn-success mt-2"
value="Login">
        </form>
    </div>
{% endblock %}
```

6.3.2.3.1.6 logout.html

*[Back to file structure: **6.2**]*

```
{% extends "timetable/base.html" %}
{% block content %}
```

```
    <div class="text-center login-width m-auto">
        <h1 class="my-4">Logged out</h1>
        <p>You have successfully logged out</p>
        <a href="/">Log in again</a>
    </div>
{% endblock %}
```

6.3.2.3.1.7 teacher.html

*[Back to file structure: **6.2**]*

```
{% extends "timetable/base.html" %}
{% block content %}
    <div class="container">
        <h1 class="text-center mt-4">Welcome Back</h1>
        <p class="text-center mt-3 mb-4">What would you like to do?</p>
        <div class="text-center">
            <a href="/teacher/timetable" class="w-100"><button class="btn
btn-primary">View your Timetable</button></a>
            <a href="/teacher/scheduled"><button class="btn btn-
primary">Schedule Lessons</button></a>
        </div>
    </div>
{% endblock %}
```

6.3.2.3.1.8 timetable.html

*[Back to file structure: **6.2**]*

```
{% extends "timetable/base.html" %}
{% block content %}
    <ul class="nav nav-tabs nav-fill">
        {% for day, link in weekdays_links.items %}
        <li class="nav-item">
            {% if link == 'disabled' %}
            <a href="#" class="nav-link disabled px-0">{{day}}</a>
            {% else %}
            <a href="?day={{link}}" class="nav-link px-0{% if
weekday_format == day %} active{%endif%}">{{day}}</a>
            {% endif %}
        </li>
        {% endfor %}
    </ul>

    <div>
        {% if not lessons %}
        <p class="text ml-2 mt-2">You have no lessons on this day</p>
        {% endif %}

        {% for lesson in lessons %}
        <div class="timetabled-lesson subject-{{forloop.counter}} card">
            <div class="card-body">
                <h5 class="card-title">{{lesson.0}}</h5>
```

```
                <p class="card-text mb-1">{{lesson.1}}</p>
                <p class="card-text mb-1">{{lesson.2}}</p>
                <p class="card-text">{{lesson.3}}</p>
            </div>
        </div>
        {% endfor %}

    </div>
    {% if user.user_type == 'teacher' %}
        <a class="text-danger ml-2" href="/">Back to Home</a><br>
    {% endif %}
    <a class="text-danger ml-2" href="/logout">Logout</a>
{% endblock %}
```

6.3.2.4 __init__.py

This file is empty

6.3.2.5 admin.py

*[Back to file structure: **6.2**]*

```
from django import forms
from django.contrib import admin
from django.contrib.auth.admin import UserAdmin
from django.contrib.auth.forms import UserCreationForm, UserChangeForm
from django.utils.translation import gettext, gettext_lazy as _

from .models import User, Lesson, Group, Link, Subject


TITLE_CHOICES = [('', "It doesn't matter"),
                 ('mr', 'Mr'),
                 ('mrs', 'Mrs'),
                 ('miss', 'Miss'),
                 ('ms', 'Ms'),
                 ('dr', 'Dr')]
USER_TYPES = [('student', 'Student'),
              ('teacher', 'Teacher')]
YEAR_CHOICES = [(None, 'N/A'),
                ('L6', 'L6'),
                ('U6', 'U6')]


class CustomUserCreationForm(UserCreationForm):
    title = forms.ChoiceField(choices=TITLE_CHOICES, required=False)
    first_name = forms.CharField()
    last_name = forms.CharField()
    user_type = forms.ChoiceField(choices=USER_TYPES)
    email = forms.EmailField()
    year_group = forms.ChoiceField(choices=YEAR_CHOICES, required=False)
```

```
    class Meta(UserCreationForm.Meta):
        model = User
        fields = ('username', 'email', 'first_name', 'last_name',
'password1', 'password2', 'user_type', 'year_group')


class CustomUserChangeForm(UserChangeForm):
    title = forms.ChoiceField(choices=TITLE_CHOICES, required=False)
    first_name = forms.CharField()
    last_name = forms.CharField()
    user_type = forms.ChoiceField(choices=USER_TYPES)
    email = forms.EmailField()
    year_group = forms.ChoiceField(choices=YEAR_CHOICES, required=False)

    class Meta(UserChangeForm.Meta):
        model = User
        fields = ('username', 'email', 'first_name', 'last_name',
'password', 'year_group')


class CustomUserAdmin(UserAdmin):
    add_form = CustomUserCreationForm
    form = CustomUserChangeForm
    add_fieldsets = (
        (None, {
            'classes': ('wide',),
            'fields': ('username', 'email', 'title', 'first_name',
'last_name', 'user_type', 'password1', 'password2'),
        }),
    )
    fieldsets = (
        (_('Authentication'), {'fields': ('username', 'email',
'password')}),
        (_('Details'), {
            'fields': ('first_name', 'last_name', 'title', 'user_type',
'year_group'),
        }),
        (_('Permissions'), {
            'fields': ('is_active', 'is_staff', 'is_superuser', 'groups',
'user_permissions'),
        }),
        (_('Important dates'), {'fields': ('last_login', 'date_joined')}),
    )


admin.site.register(User, CustomUserAdmin)
admin.site.register(Lesson)
admin.site.register(Group)
admin.site.register(Link)
admin.site.register(Subject)
```

6.3.2.6 apps.py

*[Back to file structure: **6.2**]*

```
from django.apps import AppConfig


class TimetablingAppConfig(AppConfig):
    name = 'timetable'
```

6.3.2.7 fields.py

*[Back to file structure: **6.2**]*

```
from django.forms import ModelChoiceField

from .models import User


class CustomModelChoiceField(ModelChoiceField):
    def label_from_instance(self, obj):
        example_student =
User.objects.filter(link__group_id__id__exact=obj.id,
user_type='student')[:1]
        if example_student:
            example_student = example_student.get()
        if example_student and example_student.year_group:
            text = f"{example_student.year_group} - {obj.name}"
        else:
            text = obj.name
        return text
```

6.3.2.8 forms.py

*[Back to file structure: **6.2**]*

```
from django.contrib.auth.forms import AuthenticationForm, UsernameField

from django import forms
from django.utils.safestring import mark_safe

from .fields import CustomModelChoiceField
from .models import Group, User


class LoginForm(AuthenticationForm):
    username = UsernameField(widget=forms.TextInput(
        attrs={'class': 'mb-3',
               'placeholder': '',
               'id': 'username-input'}),
```

```
        label='Username')
    password = forms.CharField(widget=forms.PasswordInput(
        attrs={
            'class': 'mb-2',
            'placeholder': '',
            'id': 'password-input',
        }),
        label='Password')


class ScheduleForm(forms.Form):

    def __init__(self, *args, **kwargs): # this is required to receive the
request object
        self.request = kwargs.pop('request')
        # initialise the form
        super(ScheduleForm, self).__init__(*args, **kwargs)
        # modify the group field to include all the relevant groups
        self.fields['group'].queryset =
Group.objects.filter(link__user_id__id__exact=self.request.user.id)

    group = CustomModelChoiceField(widget=forms.Select(
        attrs={
            'id': 'group-select'
        }),
        label='Class:', required=True,
        queryset=Group.objects.filter(link__user_id__id__exact=None),
        to_field_name='name', empty_label=None)
    topic = forms.CharField(widget=forms.TextInput(
        attrs={'class': 'w-100 mb-2',
               'placeholder': 'What will the lesson be about? (optional)',
               'id': 'topic-input'}),
        label='', max_length=128, required=False)
    buttons_html = ''
    durations = ['40', '80', '100', '120']
    for duration in durations:
        buttons_html += f'<button type="button"
onclick="document.getElementById(\'duration-input-
number\').value={duration};" class="btn-secondary btn px-1 py-
0">{duration}</button>'
    duration = forms.DurationField(widget=forms.NumberInput(
        attrs={
            'class': 'mb-2',
            'id': 'duration-input-number',
            'step': '5',
            'value': '60'
        }),
        label=mark_safe(f"Duration (minutes): {buttons_html}"))
```

6.3.2.9 models.py

*[Back to file structure: **6.2**]*

```
from django.contrib.auth.models import AbstractUser
from django.db import models
from django_project import settings


class User(AbstractUser):
    title = models.CharField(max_length=8, default='')
    first_name = models.CharField(max_length=16)
    last_name = models.CharField(max_length=16)
    user_type = models.CharField(max_length=8, default='admin')
    year_group = models.CharField(max_length=4, null=True, default=None)


class Subject(models.Model):
    name = models.CharField(max_length=16)
    abbreviation = models.CharField(max_length=4)


class Link(models.Model):
    user_id = models.ForeignKey(
        settings.AUTH_USER_MODEL,
        on_delete=models.CASCADE,
    )
    subject_id = models.ForeignKey(
        'Subject',
        on_delete=models.CASCADE
    )
    group_id = models.ForeignKey(
        'Group',
        on_delete=models.CASCADE
    )


class Group(models.Model):
    name = models.CharField(max_length=8)


class Lesson(models.Model):
    group = models.ForeignKey(
        'Group',
        on_delete=models.CASCADE
    )
    duration = models.DurationField()
    topic = models.CharField(max_length=128, default='', null=True)
    start = models.DateTimeField(null=True, blank=True, default=None)
    fixed = models.BooleanField(default=False)
```

6.3.2.10 tests.py

This file was not modified during development

6.3.2.11 timetabling.py

*[Back to file structure: **6.2**]*

```
import copy
import datetime
import math
import random
from typing import List, Optional, Tuple

from celery import Celery
from celery.schedules import crontab

from .models import Lesson, User, Group

app = Celery()


@app.task(run_every=crontab(hour=20, minute=0))
def schedule_lessons(iterations=10, look_ahead_period=14):
    """Creates a timetable using the unscheduled lessons from the
database"""
    base_day =
datetime.datetime.now(datetime.timezone.utc).replace(minute=0, hour=0,
second=0)
    for x in range(look_ahead_period):
        day = base_day + datetime.timedelta(days=x)
        end_of_day = day.replace(hour=23, minute=59, second=59)
        if day.weekday() <= 4:  # a weekday
            if not Lesson.objects.filter(start__gte=day,
start__lte=end_of_day):
                print('-----------')
                print(f"SCHEDULING {day}")
                best_result: Optional[Timetable] = None
                for x in range(iterations):
                    print(f"Iteration: {x+1}")
                    population = Population(year_start=base_day,
first_day=base_day + datetime.timedelta(days=x))
                    output: Timetable = population.start()
                    if best_result is None or output.get_cost() <
best_result.get_cost():
                        best_result = output

                print(f"Adding best result (cost:
{best_result.get_cost()})")
                best_result.add()

                for lesson in best_result.lessons[0]:
                    new_lesson = Lesson()
                    new_lesson.group_id = lesson.group_id
                    duration = random.randint(6, 24)
                    new_lesson.duration =
datetime.timedelta(seconds=duration * 300)
```

```
                    new_lesson.topic = f"Automatically generated while
timetabling"
                    new_lesson.fixed = False
                    new_lesson.save()


def should_stop(current_population, iterations):
    if iterations >= 100:
        return True
    else:
        return False


def get_unscheduled_lessons(first_day, days=1, seconds_per_unit_time: float
= 300):
    unscheduled_lessons = []
    per_class = {}
    for unscheduled_lesson in
Lesson.objects.filter(fixed=False).exclude(start__lte=first_day):
        unscheduled_lesson = PotentiallyScheduledLesson(unscheduled_lesson,
seconds_per_time_unit=seconds_per_unit_time)

        if unscheduled_lesson.group_id in per_class:
            per_class[unscheduled_lesson.group_id] += 1
        else:
            per_class[unscheduled_lesson.group_id] = 1

        if per_class[unscheduled_lesson.group_id] <= days:  # this helps to
reduce the possibilities to consider
            unscheduled_lessons.append(unscheduled_lesson)

    return unscheduled_lessons


def get_group_data():
    group_data = {}
    for group in Group.objects.filter():
        previous = None
        time_allocated = 0
        for lesson in Lesson.objects.filter(group_id__id__exact=group.id,

start__lte=datetime.datetime.now(tz=datetime.timezone.utc)).order_by(
            'start'):  # oldest first
            previous = lesson
            time_allocated += lesson.duration.total_seconds()
            days_since_previous =
(datetime.datetime.now(datetime.timezone.utc).replace(hour=0, minute=0,
second=0)
                                    -
previous.start.replace(tzinfo=datetime.timezone.utc, hour=0, minute=0,
second=0)).days
        group_data[group.id] = [time_allocated, days_since_previous]

    return group_data
```

```
def get_year_start():
    first_lesson = Lesson.objects.order_by('start')[:1].get()
    year_start = first_lesson.start
    if not year_start:  # if no lessons in database
        dt = datetime.datetime.now(datetime.timezone.utc)
        if dt.month < 9:  # jan - aug
            # previous september
            year_start = datetime.datetime(year=dt.year - 1, month=9,
day=1, tzinfo=datetime.timezone.utc)
        else:
            # in current year
            year_start = datetime.datetime(year=dt.year, month=9, day=1,
tzinfo=datetime.timezone.utc)

    return year_start


def schedule(*args, **kwargs):
    """A helper function to generate the best timetable using a genetic
algorithm"""
    population = Population(*args, **kwargs)
    return population.start()


class Population:
    """Represents a population of timetables for use in the genetic
algorithm"""

    def __init__(self, timetable_init_kwargs=None,
desired_allocations=None,
                 popsize=200, num_parents=50, num_offspring=100,
                 mutation_amount=3, mutation_chance=0.7,
guaranteed_parent_survival=5,
                 stopping_condition=should_stop,
random_lesson_skip_probability: float = 0.2,
                 first_day: Optional[datetime.datetime] = None, days: int =
1,
                 time_per_day: int = 114, seconds_per_unit_time: float =
300,
                 desired_lessons: int = 44,
                 day_start=datetime.timedelta(hours=8, minutes=30),
year_start=None):
        """
        :param popsize: The population size
        :param stopping_condition: A function taking in:
         - the current population (of type Population)
         - previous population (THIS WILL BE NONE ON THE FIRST ITERATION)
         - the number of iterations / generations
         and returning True to stop and False to continue
        """

        if desired_allocations is None:
```

```
            self.desired_allocations = {}
        else:
            self.desired_allocations = desired_allocations
        if timetable_init_kwargs is None:
            timetable_init_kwargs = {}
        self.timetable_init_kwargs = timetable_init_kwargs
        if first_day:
            self.first_day = first_day
        else:
            self.first_day = datetime.datetime.now(datetime.timezone.utc)
        self.days = days
        self.time_per_day = time_per_day
        self.seconds_per_unit_time = seconds_per_unit_time
        self.day_start = day_start
        self.desired_lesson_time = desired_lessons

        self.popsize = popsize
        self.num_parents = num_parents
        self.num_offspring = num_offspring
        self.guaranteed_surviving_parents = guaranteed_parent_survival
        self.stopping_condition = stopping_condition
        self.generations = 0
        self.mutation_amount = mutation_amount
        self.mutation_chance = mutation_chance
        self.random_lesson_skip_probability =
random_lesson_skip_probability

        self.unscheduled_lessons = get_unscheduled_lessons(self.first_day,
self.days, self.seconds_per_unit_time)
        self.group_data = get_group_data()
        self.all_students = []
        for group_id in self.group_data:
            for student in User.objects.filter(user_type='student',
link__group_id__id__exact=group_id):
                if student not in self.all_students:
                    self.all_students.append(student)
        if year_start:
            self.year_start = year_start
        else:
            self.year_start = get_year_start()

        if self.num_parents > self.popsize:
            raise ValueError("Number of parents cannot be greater than size
of population")
        if self.guaranteed_surviving_parents > self.num_parents:
            raise ValueError("Surviving parents cannot be more than the
number of parents")
        if self.popsize < 1:
            raise ValueError("Population must be positive")

        for group_id in self.group_data:
            # default desired allocations in case it is not provided
            if group_id not in self.desired_allocations:
                self.desired_allocations[group_id] = 1
```

```
                print(f"Warning: Group {group_id} had no desired
allocation. Set to 1")

        self.population: List[Timetable] = []
        for x in range(self.popsize):
            self.population.append(Timetable(first_day=self.first_day,
days=self.days, time_per_day=self.time_per_day,

seconds_per_unit_time=self.seconds_per_unit_time, day_start=self.day_start,

desired_allocations=self.desired_allocations, group_data=self.group_data,

random_lesson_skip_probability=self.random_lesson_skip_probability,

desired_lesson_time=self.desired_lesson_time,
all_students=self.all_students,
                                                year_start=self.year_start,
**timetable_init_kwargs).random())

    def start(self):
        """Iterate over the solution until self.stopping_condition returns
True
         self.stopping_condition should have a fallback condition on the
number of iterations to prevent an infinite loop"""

        self.population = self.evaluate_all_costs(self.population)
        while not self.stopping_condition(self, self.generations):
            self.iterate()
            self.generations += 1

        best = self.select_best_solution()

        return best

    def iterate(self):
        """Performs one iteration of the genetic algorithm on the current
population"""

        parents = self.choose_parents()
        offspring = self.generate_offspring(list(parents))
        candidates = list(self.population + offspring)
        self.population = self.choose_new_population(candidates)
    def select_best_solution(self):
        """Chooses the best solution, re-evaluating the cost function for
all"""

        best = min(self.population, key=lambda t: t.get_cost())

        return best

    def choose_new_population(self, candidates):
        """Chooses the new population from the list of candidates
        The value of the cost function should be either the correct value
or infinity if not evaluated"""
```

```
        new_population = []
        previous_highest_cost = max(candidates, key=lambda t:
t.get_cost()).get_cost()

        # carry forward the best solutions from the previous iteration
        for x in range(self.guaranteed_surviving_parents):
            best_individual = min(candidates, key=lambda t: t.get_cost())
            new_population.append(best_individual)
            candidates.remove(best_individual)

        if len(candidates)+self.guaranteed_surviving_parents <=
self.popsize:
            return new_population + candidates

        # choose the remaining solutions randomly, with a probability
proportional to the cost function
        # sorting the list based on cost would be ideal however is too
expensive
        while len(new_population) < self.popsize:
            i = random.randint(0, len(candidates) - 1)
            candidate = candidates[i]

            # choose with probability (roughly) proportional to the value
of the cost function
            #   - this will be incorrect if an uncalculated cost is greater
than that of the previous iteration
            p = candidate.get_cost() / previous_highest_cost
            # higher cost is worse, so this gives the probability that it
will fail
            if p < random.uniform(0, 1):
                new_population.append(candidates.pop(i))

        return new_population

    def evaluate_all_costs(self, population):
        """DEPRECATED: Evaluates the cost function for every timetable in
the given population"""
        return population

    def choose_parents(self):
        """Chooses the best parents from the population to mate"""

        candidates = list(self.population)
        parents = []

        # choose the best parents
        for x in range(self.num_parents):
            best_parent = min(candidates, key=lambda t: t.get_cost())
            parents.append(best_parent)
            candidates.remove(best_parent)

        return parents

    def generate_offspring(self, parents):
```

```
        """Generates all the offspring"""

        offspring = []
        for x in range(self.num_offspring):
            parent1 = random.choice(parents)
            parent2 = random.choice(parents)
            child = self.crossover(parent1, parent2)
            child =
child.mutate(mutate_lessons_per_day=self.mutation_amount)
            offspring.append(child)

        return offspring

    def crossover(self, parent1, parent2):
        """Returns one offspring containing half information from each
parent"""

        new_lessons = {}
        for day in range(self.days):
            new_lessons[day] = []
            added_ids = []
            potential_new_lessons = (parent1.lessons[day] +
parent2.lessons[day]).copy()
            random.shuffle(potential_new_lessons)

            for x in range(len(potential_new_lessons) // 2):
                if potential_new_lessons:
                    lesson: PotentiallyScheduledLesson =
potential_new_lessons.pop(0)
                    if lesson.id not in added_ids:
                        new_lessons[day].append(lesson.copy())
                        added_ids.append(lesson.id)

        timetable = Timetable(lessons=new_lessons,
first_day=self.first_day, days=self.days,
                              time_per_day=self.time_per_day,
seconds_per_unit_time=self.seconds_per_unit_time,
                              desired_allocations=self.desired_allocations,
group_data=self.group_data,
                              day_start=self.day_start,
desired_lesson_time=self.desired_lesson_time,

random_lesson_skip_probability=self.random_lesson_skip_probability,
                              all_students=self.all_students,
unscheduled_lessons=self.unscheduled_lessons,
                              year_start=self.year_start,
**self.timetable_init_kwargs)

        return timetable  # the cost function may not be needed, so it does
not need to be executed here

    def mutate(self, offspring):
        """Randomly changes the offspring slightly"""
```

```
        for timetable in offspring:
            if random.uniform(0, 1) < self.mutation_chance:
                timetable.mutate()

        return offspring


class Timetable:
    """Represents a potential timetable for a given period of time"""

    def __init__(self, first_day: Optional[datetime.datetime] = None, days:
int = 1, time_per_day: int = 114,
                 seconds_per_unit_time: float = 300,
day_start=datetime.timedelta(hours=8, minutes=30), year_start=None,
                 unscheduled_lessons=None, group_data=None,
desired_allocations=None, lessons=None,
                 desired_lesson_time=44, random_lesson_skip_probability:
float = 0.2, all_students=None):
        if first_day:
            self.first_day = first_day
        else:
            self.first_day = datetime.datetime.now(datetime.timezone.utc)
        self.days = days
        self.time_per_day = time_per_day
        self.seconds_per_unit_time = seconds_per_unit_time
        self.day_start = day_start
        self.desired_lesson_time = desired_lesson_time
        self.random_lesson_skip_probability =
random_lesson_skip_probability
        if desired_allocations:
            self.desired_allocations = desired_allocations
        else:
            self.desired_allocations = {}
        self.cost = float('inf')
        self.modified = True

        if year_start:
            self.year_start = year_start
        else:
            self.year_start = get_year_start()

        if unscheduled_lessons:
            self.unscheduled_lessons = []
            for lesson in unscheduled_lessons:
                self.unscheduled_lessons.append(lesson.copy())
        else:
            self.unscheduled_lessons =
get_unscheduled_lessons(self.first_day, self.days,
self.seconds_per_unit_time)
        random.shuffle(self.unscheduled_lessons)

        if group_data:
            self.group_data = group_data
        else:
```

```
            self.group_data = get_group_data()

        self.all_students = all_students

        if not lessons:
            self.lessons = {}  # format {day: lesson} of type {int:
PotentiallyScheduledLesson}
            for d in range(self.days):
                self.lessons[d] = []
        else:
            self.lessons = lessons

    def __eq__(self, other):
        if isinstance(other, Timetable):
            return self.lessons == other.lessons
        else:
            return self == other

    def random(self, threshold=10, true_random_min=None,
true_random_max=None):
        """Generates a random solution
        This algorithm makes some attempt to minimise teacher clashes while
being quick to execute"""

        if true_random_min and true_random_max:
            for x in range(random.randint(true_random_min,
true_random_max)):
                lesson = random.choice(self.unscheduled_lessons)
                lesson = PotentiallyScheduledLesson(lesson)
                latest_end = self.time_per_day - lesson.relative_duration
                lesson.relative_start = random.randint(0, latest_end)
                self.lessons[0].append(lesson)

        else:
            counter = 0
            for lesson in self.unscheduled_lessons:
                lesson: PotentiallyScheduledLesson
                teacher = self.get_teacher(lesson)
                day = random.randint(0, self.days - 1)
                gaps = self.get_gaps(user_id=teacher.id, days=[day],
random_order=True, boundaries=True)

                for gap_start, gap in gaps:  # for each (random) gap...
                    if random.uniform(0, 1) <
self.random_lesson_skip_probability:
                        continue
                    if gap > lesson.relative_duration + 1:  # if there's
enough space for a lesson (need at least 1 unit either side)...
                        if gap < lesson.relative_duration * 1.5:  # if
there's not much space...
                            lesson.relative_start = gap_start + 1  #
schedule for start of gap (plus 1 unit break)
                        else:
                            latest_end = gap_start + gap - 2
```

```
                            lesson.relative_start =
random.randint(gap_start,

latest_end - lesson.relative_duration)  # allocate to random position
                        self.lessons[day].append(lesson)
                        break
                    else:
                        continue
                else:  # this triggers if the end of the loop is reached
without a break statement
                    counter += 1
                    if counter > threshold:
                        break

        self.modified = True

        return self

    def get_teacher(self, lesson: 'PotentiallyScheduledLesson'):
        """Gets a teacher who teaches the given lesson
        Teachers are cached, so the cached version will be returned upon
any future calls"""
        if not hasattr(lesson, 'teacher'):
            lesson.teacher =
User.objects.filter(link__group_id__lesson__id__exact=lesson.id,

user_type__exact='teacher')[:1].get()
        return lesson.teacher

    def get_gaps(self, user_id, days=None, random_order=False,
boundaries=False) -> List[Tuple[int, int]]:
        """Gets a list of the duration of every gap on the given day, in
time units

        Days should be a list of numbers, each indicating the number of
days since first_day
        By default, only gaps between lessons are returned. However, if
boundaries = True, then the gaps between the
            start of the day and the first lesson will be returned
(likewise with the final lesson)

        Returns dict of form [(start of gap, length of gap)] of type [(int,
int)]"""
        if not days:
            days = range(self.days)

        previous = None
        gaps = []
        for day in days:
            sorted(self.lessons[day])
            if boundaries:
                previous = 0
            for lesson in self.lessons[day]:
                if previous:
```

```
                    gaps.append((previous, lesson.relative_start -
previous))
                previous = lesson.relative_start
            if boundaries:
                gaps.append((previous, self.time_per_day - previous))

        if random_order:
            random.shuffle(gaps)

        return gaps

    def get_gap_cost(self, gap_length):
        """Returns the value to add to the cost function for a gap of
length gap_length, in time units
        Gaps should never be negative, but if so this function will return
0"""
        if gap_length == 0:
            return 10
        # 1 is the perfect length -> 0 is returned
        elif gap_length in (2, 3):
            return 5
        elif gap_length == 3:
            return 2
        elif gap_length == 4:
            return 1
        else:
            return 0

    def get_cost(self, debug=False, force=False):
        """Evaluates the cost function for the current solution"""
        POINTS_PER_TEACHER_CLASH = 100
        POINTS_PER_STUDENT_CLASH = 10
        MAX_LOAD_CONSTANT = 23  # max load = c ln c, where c is this value
        EARLY_FINISH_CONSTANT = 10
        EARLIEST_EARLY_FINISH = 48
        EVEN_ALLOCATION_CONSTANT_A = 0.4
        EVEN_ALLOCATION_CONSTANT_B = -8
        EVEN_ALLOCATION_CONSTANT_K = 1000
        WEIGHTING_OF_DIFF = 100
        VARIETY_BASE = 2
        VARIETY_COEFFICIENT = 1
        VARIETY_CONSTANT = 1_000_000
        DESIRED_LESSONS_BASE = 1.2
        DESIRED_LESSONS_MULTIPLIER = 25

        if not self.modified and not force and not debug:
            return self.cost

        total_cost = 0
        debug_info = {}
        for day in range(self.days):
            # constraints 1 & 2: clashes
            teacher_clashes = 0
            student_clashes = 0
```

```
            schedules = {}  # {user_id: [time_slot]}
            user_types = {}  # {user_id: user_type}
            for lesson in self.lessons[day]:
                for user in lesson.get_users():
                    clashes = 0
                    if user.id not in schedules:
                        schedules[user.id] = []
                    for x in range(lesson.relative_duration):
                        time_slot = lesson.relative_start + x
                        if time_slot in schedules[user.id]:
                            clashes += 1
                        else:
                            schedules[user.id].append(time_slot)
                    if user.user_type == 'teacher':
                        teacher_clashes += clashes
                    else:
                        student_clashes += clashes
                    user_types[user.id] = user.user_type
            clashes_cost = POINTS_PER_STUDENT_CLASH * student_clashes +
POINTS_PER_TEACHER_CLASH * teacher_clashes

            # constraint 3: even allocation of lesson times
            def sigmoid(a):
                return 1 / (1 + math.exp(-a))

            weighting = sigmoid(
                EVEN_ALLOCATION_CONSTANT_B + EVEN_ALLOCATION_CONSTANT_A *
(self.first_day - self.year_start).days)
            diffs = 0
            for group_id in self.group_data:
                diff = abs(self.group_data[group_id][0] -
self.desired_allocations[group_id]) / \
                       self.desired_allocations[group_id]
                diffs += diff
            even_allocation_cost = WEIGHTING_OF_DIFF * weighting * diffs /
EVEN_ALLOCATION_CONSTANT_K

            # constraint 3a: how many lessons
            total_lesson_time = 0
            n_students = 0
            if not self.all_students:
                for user_id in schedules:
                    if user_types[user_id] == 'student':
                        total_lesson_time += len(schedules[user_id])
                        n_students += 1
            else:
                for student in self.all_students:
                    if student.id in schedules:
                        total_lesson_time += len(schedules[student.id])
                    n_students += 1
            try:
                average_lesson_time = total_lesson_time / n_students
            except ZeroDivisionError:
                average_lesson_time = 0
```

```
            lessons_scheduled_cost = DESIRED_LESSONS_MULTIPLIER *
DESIRED_LESSONS_BASE ** (self.desired_lesson_time - average_lesson_time)

            # NOTE: constraint 4 is being computed with constraint 7

            # constraint 5: variety of subjects
            variety_cost = 0
            for group_id in self.group_data:
                variety_cost += VARIETY_COEFFICIENT * (VARIETY_BASE **
self.group_data[group_id][1])
            variety_cost /= VARIETY_CONSTANT * len(self.group_data)

            # constraint 6: max daily workload
            daily_workload_cost = 0
            for user_id in schedules:
                # NOTE: It is assumed that if any clashes occur, the user
will miss out on
                # all but one of the concurrent events, so it does not
contribute to their overall workload
                daily_workload_cost += max(math.exp(len(schedules[user_id])
/ MAX_LOAD_CONSTANT) - MAX_LOAD_CONSTANT, 0)

            # constraint 7: gaps
            gaps_cost = 0
            for user_id in schedules:  # NOTE: schedules is only used for a
list of user ids
                gaps = self.get_gaps(user_id)
                for gap in gaps:
                    gaps_cost += self.get_gap_cost(gap)

            # constraint 8: early finish time
            early_finish_cost = 0
            for user_id in schedules:
                finish_time = max(schedules[user_id]) -
EARLIEST_EARLY_FINISH
                if finish_time > 0:
                    early_finish_cost += finish_time /
EARLY_FINISH_CONSTANT

            total_cost += clashes_cost + even_allocation_cost +
lessons_scheduled_cost + variety_cost \
                          + daily_workload_cost + gaps_cost +
early_finish_cost

            debug_info = {
                'teacher clashes': teacher_clashes,
                'student clashes': student_clashes,
                'clashes cost': clashes_cost,
                'total diffs': diffs,
                'even allocation cost': even_allocation_cost,
                'average lesson time': average_lesson_time,
                'lessons scheduled cost': lessons_scheduled_cost,
                'variety cost': variety_cost,
                'daily workload cost': daily_workload_cost,
```

```
                'gaps cost': gaps_cost,
                'early finish cost': early_finish_cost
            }

            # print(debug_info)

        self.cost = total_cost
        self.modified = False

        if debug:
            return debug_info
        else:
            return max(total_cost, 0)

    def get_fitness(self):
        """Returns the fitness value for a solution
        This is simply the negative of the cost value"""
        return -self.get_cost()

    def mutate(self, mutate_lessons_per_day=2):
        """Mutates the given solution (for use in a genetic algorithm)
        NOTE: The same lesson could be mutated twice (although unlikely)"""

        for day in range(self.days):
            for x in range(mutate_lessons_per_day):
                n = random.randint(1, 3)
                if n == 1:
                    # mutate start time of random lesson
                    if self.lessons[day]:
                        i = random.randint(0, len(self.lessons[day]) - 1)
                        lesson = self.lessons[day][i]
                        latest_time = self.time_per_day -
lesson.relative_duration
                        lesson.relative_start = random.randint(0,
latest_time)
                elif n == 2:
                    # delete a random lesson
                    if self.lessons[day]:
                        lesson = self.lessons[day].pop(random.randint(0,
len(self.lessons[day]) - 1))
                        # add to random position in unscheduled lessons
                        self.unscheduled_lessons.insert(random.randint(0,
len(self.unscheduled_lessons)), lesson)
                elif n == 3:
                    # add a random lesson
                    if self.unscheduled_lessons:
                        lesson =
self.unscheduled_lessons.pop(random.randint(0,
len(self.unscheduled_lessons) - 1))
                        latest_time = self.time_per_day -
lesson.relative_duration
                        lesson.relative_start = random.randint(0,
latest_time)
                        self.lessons[day].append(lesson)
```

```
        self.modified = True

        return self

    def add(self):
        """Update the database to include the start times for all lessons
currently stored within this object"""
        for day in self.lessons:
            for lesson in self.lessons[day]:
                new_lesson = Lesson()
                new_lesson.group_id = lesson.group_id
                new_lesson.duration = lesson.duration  # relative_duration
not required because duration is never modified
                new_lesson.topic = lesson.topic
                new_lesson.fixed = True
                start_time = self.first_day + datetime.timedelta(days=day)
+ self.day_start + datetime.timedelta(seconds=lesson.relative_start *
self.seconds_per_unit_time)
                new_lesson.start = start_time
                new_lesson.save()


class PotentiallyScheduledLesson:
    """A Lesson used as part of a Timetable
    Notably, this abstracts the start time to make computation easier"""

    DESIRED_FIELDS = [
        'id',
        'duration',
        'group_id',
        'topic'
    ]

    def __init__(self, lesson, seconds_per_time_unit: float = 300):
        for field in self.DESIRED_FIELDS:
            self.__dict__[field] = lesson.__dict__[field]
        self.relative_start = None  # contains start time in time units
relative to start of day
        self.relative_duration = math.floor(lesson.duration.total_seconds()
/ seconds_per_time_unit)
        self.users = None

    def __gt__(self, other):
        return self.relative_start > other

    def __lt__(self, other):
        return self.relative_start < other

    def __str__(self):
        if hasattr(self, 'teacher'):
            return f"<Lesson teacher='{self.teacher.username}'
start={self.relative_start}>"
        else:
```

```
            return f"<Lesson start={self.relative_start}>"

    def copy(self):
        return copy.copy(self)

    def __repr__(self):
        return self.__str__()

    def get_users(self):
        """Retrieves all users in this lesson, caching the result for
subsequent queries"""
        if not self.users:
            self.users =
User.objects.filter(link__group_id__lesson__id__exact=self.id)
        return self.users

    @classmethod
    def from_lesson(cls, *args, **kwargs):
        """DEPRECATED: Should not be used"""
        return cls(*args, **kwargs)
```

6.3.2.12 urls.py

*[Back to file structure: **6.2**]*

```
from django.urls import path
from django.contrib.auth import views as django_views
from . import views
from .forms import LoginForm

urlpatterns = [
    path('login/',
django_views.LoginView.as_view(template_name='timetable/login.html',
authentication_form=LoginForm), name='timetable-login'),
    path('logout/',
django_views.LogoutView.as_view(template_name='timetable/logout.html'),
name='timetable-logout'),

    path('', views.login_redirect, name='timetable-login-redirect'),

    path('student/', views.timetable, name='timetable-student'),

    path('teacher/', views.teacher, name='timetable-teacher'),
    path('teacher/timetable', views.timetable, name='timetable-teacher'),
    path('teacher/scheduled', views.teacher_scheduled, name='timetable-
teacher-scheduled'),
    path('teacher/schedule', views.teacher_scheduler, name='timetable-
teacher-schedule'),
]
```

6.3.2.13 views.py

*[Back to file structure: **6.2**]*

```
import datetime
import math

from django.contrib.auth.decorators import login_required
from django.contrib.auth.models import AnonymousUser
from django.shortcuts import render, redirect

from .forms import ScheduleForm
from .models import Lesson, Subject, User, Group

MIN_UNSCHEDULED_LESSONS = 3

def login_redirect(request):
    # redirect appropriately depending on user type
    user = request.user
    if isinstance(user, AnonymousUser):
        return redirect('/login/')
    else:
        if user.user_type == 'student':
            return redirect('/student/')
        elif user.user_type == 'teacher':
            return redirect('/teacher/')
        else:
            return redirect('/admin/')


@login_required
def timetable(request):
    user = request.user
    day = request.GET.get('day')

    if day and day.isdigit():  # user could have modified it to not be an
int
        day = int(day)
        current_date = datetime.datetime.utcfromtimestamp(int(day))
    else:
        current_date = datetime.datetime.utcnow()
    weekday: int = current_date.weekday()
    if weekday >= 5:
        current_date = current_date + datetime.timedelta(days=7-weekday)
        weekday = 0
    weekstart = current_date - datetime.timedelta(days=weekday)
    after = current_date.replace(hour=0, minute=0, second=0)
    before = current_date.replace(hour=23, minute=59, second=59)
    weeks_diff = math.floor((current_date -
datetime.datetime.utcnow()).days / 7)

    weekdays = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri']
    if weeks_diff <= -2:
```

```
        weekdays_links = {'<': 'disabled'}
    else:
        weekdays_links = {'<': math.floor((weekstart -
datetime.timedelta(days=3)).timestamp())}
    for i in range(5):
        d = weekstart + datetime.timedelta(days=i)
        if i == weekday:
            key = weekdays[i]+' '+str(d.day)
            weekday_format = key
        else:
            key = weekdays[i]
        weekdays_links[key] = math.floor(d.timestamp())
    if weeks_diff >= 2:
        weekdays_links['>'] = 'disabled'
    else:
        weekdays_links['>'] = math.floor((weekstart +
datetime.timedelta(days=7)).timestamp())

    lessons = []

    for lesson in
Lesson.objects.filter(group_id__link__user_id__username__exact=user.usernam
e, start__gte=after, start__lte=before).order_by('start'):
        lesson_data = []
        if user.user_type == 'student':

lesson_data.append(Subject.objects.filter(link__group_id__lesson__id__exact
=lesson.id)[:1].get().name)
            teacher =
User.objects.filter(link__group_id__lesson__id__exact=lesson.id,
user_type='teacher')[:1].get()
            if teacher.title:
                lesson_data.append(teacher.title.title() + ' ' +
teacher.last_name.title())
            else:
                lesson_data.append(teacher.first_name.title() + ' ' +
teacher.last_name.title())
        else:

lesson_data.append(Group.objects.filter(lesson__id__exact=lesson.id)[:1].ge
t().name)
            topic = lesson.topic
            if len(topic) > 44:
                topic = topic[:42]+'...'
            lesson_data.append(topic)
        lesson_data.append('Room')
        start = lesson.start
        end = lesson.start + lesson.duration
        lesson_data.append(start.strftime('%H:%M') + ' - ' +
end.strftime('%H:%M'))
        lessons.append(lesson_data)

    return render(request, 'timetable/timetable.html', context={'lessons':
lessons, 'weekday_format': weekday_format, 'weekdays_links':
```

```
weekdays_links})


@login_required
def teacher(request):
    return render(request, 'timetable/teacher.html')


@login_required
def teacher_scheduler(request):
    if request.method == 'POST':
        form = ScheduleForm(request.POST, request=request)
        if form.is_valid():
            Lesson.objects.create(duration=form.cleaned_data['duration'],
topic=form.cleaned_data['topic'],
                                  group=form.cleaned_data['group'])
            return redirect('/teacher/scheduled')

    else:
        form = ScheduleForm(request=request, label_suffix='')

    return render(request, 'timetable/scheduling/schedule.html', {'form':
form})


@login_required
def teacher_scheduled(request):
    user = request.user

    lessons = []
    unscheduled = 0
    for lesson in
Lesson.objects.filter(group_id__link__user_id__username__exact=user.usernam
e).exclude(start__lte=datetime.datetime.now()).order_by('id'):

        hours = math.floor(lesson.duration.seconds / 3600)
        minutes = math.floor((lesson.duration.seconds - hours*3600) / 60)

        if lesson.topic:
            topic = lesson.topic
        else:
            topic = '[untitled]'  # this makes more sense than an empty
string

        if not lesson.fixed:
            unscheduled += 1

        lessons.append({'topic': topic, 'duration': str(hours)+'h
'+str(minutes)+'m', 'fixed': lesson.fixed})

    return render(request, 'timetable/scheduling/scheduled_list.html',
{'lessons': lessons, 'enough': unscheduled >= MIN_UNSCHEDULED_LESSONS})
```

6.3.3 db.sqlite3

*[Back to file structure: **6.2**]*

This file contains the contents of the database. It has been excluded

6.3.4 manage.py

*[Back to file structure: **6.2**]*

This file was not modified, so has been excluded

END OF DOCUMENT