

William Bosley

Registration number 100273515

2022

Student Work Plan Scheduler

Supervised by Richard Harvey



University of East Anglia
Faculty of Science
School of Computing Sciences

Abstract

Many students' academic performances is impacted by poor organisation and time management. Software to generate study schedules for students is not readily available. This paper documents the development of a proof of concept scheduling application which generates a work plan for students as if they had a personal assistant, and applies the plan to their Google calendar. It is concluded that the proof of concept is a success, and that the application brings us a step closer to having easy and effective scheduling for all students.

Acknowledgements

I would like to thank Dr Richard Harvey for his unending support, ideas, feedback, and patience.

Contents

1. Introduction	6
2. Design.	7
2.1. Existing calendaring systems.	7
2.2. Overview of scheduling processes.	9
2.3. Tasks.	10
2.4. Simple events.	13
2.5. Scheduling.	17
2.5.1. checkFreeLocally	19
2.6. Google Calendar events.	22
2.7. Breaks.	24
2.8. Work hours.	26
2.9. Rescheduling.	27
2.9.1. listAllTasks	28
3. Outcome, Impacts and Evaluation.	34
3.1. Testing	35
3.2. Limitations of the design, and how they can be addressed.	39
3.3. Conclusion	41
References	42
A. Requirements: MoSCoW analysis.	44
B. Summary of project notebook.	46

List of Figures

1.	Flowchart showing the sequence of steps the program takes when the user is adding tasks without rescheduling. Some processes have been simplified into single steps for clarity.	10
2.	The second flow chart, showing the sequence of steps the program can take when the user is adding tasks, with the steps needed for rescheduling old tasks visualised. Some processes have been simplified into single steps for clarity.	11
3.	Class diagram showing task, Google Calendar event, simple event and break objects.	12
4.	The form where users can add tasks.	14
5.	The doctors appointment occurs before the break from work in the list of simple events because it's start time is earlier. This could mean that when clashes are checked for previous event (in this case the break from work) and the next event (the sports practice) the clash with the doctors appointment may be missed.	21
6.	The scheduler's form for the user to set custom work hours.	26
7.	The scheduler's UI.	36
8.	The first upcoming week of the user's Google calendar before and after scheduling of new tasks. For clarity, all pre existing events have been coloured green.	38
9.	The user interface after the user has entered their tasks, work hours and breaks, but prior to them pressing the button to schedule their tasks. . .	39
10.	The user interface after the user has selected the option to reschedule, and added their breaks, but prior to them pressing the button to (re)schedule their tasks.	40
11.	The second upcoming week of the user's Google calendar before and after scheduling of new tasks. This week was originally left blank to make it visible how the scheduler spreads out workloads, and makes sure to schedule around breaks.	48
12.	A code snippet from within checkFreeBusy for determining whether a time slot is free or busy.	49
13.	The first part of the table of tests.	50

14.	The second part of the table of tests.	51
15.	The first upcoming week of the user's Google calendar before and after rescheduling of tasks after addition of the drs appointment in red. The user's pre existing events are in green, the doctors appointment in red, and the tasks generated by the scheduler are in blue.	52
16.	The second upcoming week of the user's Google calendar before and after scheduling of new tasks.	53

1. Introduction

A study published this year by the Higher Education Statistics Agency found that 5.3% of young UK students dropped out after their first year of studies(HESA, 2022). One of the primary academic causes for students to drop out of university is lack of preparation. However, strengthening students' academic skills can effectively forestall this(Cuseo, 2011).

Some insights into time management's role as an academic skill were identified within the this project's mid development progress report(Bosley, 2021). Sheffield University was found to state that time management is an important and difficult skill to develop(TUOS, 2021). Information from the University of Cambridge and the Open University was gathered stating the amount of time students should spend studying per week, with Cambridge recommending between 42 and 46 hours a week and the Open University suggesting between 32 and 36 hours per week(Cambridge, 2021; OU, 2021). However, insights were also gained into the real amount of time spent studying by students. A 2019 study of 29,784 UK students concluded that under 11 hours a week were spent studying for 56% of students (Neves, 2019).

It was also noted that there is a correlation between students' academic performance and their perceived control of time (Adams and Blair, 2019). This information, the insights above, and the fact that Computer Science has a drop out rate of 9.8%(Frobisher, 2020; Roberts, 2020), the highest of any course in the UK, suggested that a solution should be found to improve students time management and academic organisation. If the negative impact of poor time management could be circumvented via a digital solution, students being held back by poor time management could be more successful.

A preliminary investigation into time management and organisational software available for students was conducted. Research and testing of the myHomework, Calendly, and iStudiez suggested that the market for student organisational apps was focused primarily on list making and presenting student's university timetables. Although these applications were useful in that they held lots of the user's information in one place, they had no smart abilities that made them more useful and unique than a calendaring app and a to do list.

With the rise of online learning, the lack of organisational tools for students seems like an anomaly. The Covid-19 pandemic has caused a 'paradigm shift' from classroom to online learning(Pokhrel and Chhetri, 2021). Online learning materials were

accessed by 46% of British students in 2020 (Statista, 2020), and the E-Learning market is predicted to have 9.94 billion U.S. Dollars of growth in the UK between 2020 and 2025(Technavio, 2021). A study concluded that online learning should be continued in the years following the pandemic's end as well(Pokhrel and Chhetri, 2021). The current prominence and predicted growth of E-Learning and the lack of digital applications with unique and innovative solutions suggests that there is a lack of developers focusing on a smart, digital aid for student's time management.

Following this research, this project's aim was to create a digital solution to the problem of students' poorly managed time due to incompetently made schedules, with an aim of helping student's academic success. This solution is to create an application which will generate a professional schedule for a student as if they had a personal assistant.

The requirements needed for the project to be a success were defined within a MoSCoW analysis, and the ideal product was defined in an ideal product description(Bosley, 2021). Both were originally written within the project's progress and engagement report, but they are available to view in appendixA.

2. Design.

2.1. Existing calendaring systems.

Calendar software and calendar APIs are already widely used across desktops and smartphones. Although complex, they are not an unexplored field of computer science. The aim of this project is to create a scheduling program. In order to stay focused on the goal, the project needed a calendar API to work with to avoid reprogramming a calendaring system from scratch. Before initial programming, existing calendar systems were reviewed to find the one most appropriate and useful for the system. Four APIs were inspected: Outlook, Google, Nylas and 30Boxes.

30 Boxes had a fairly wide range of functions and usages. However, the documentation seemed dated and limited. It appeared to have a developer forum which could have been useful if it had been chosen, but trying to access the forum lead to an error page saying the website was not found. The lack of documentation and learning resources would have unnecessarily slowed the projects progression, so it was ruled out. It is extremely beneficial that this API was not chosen, as it was recently announced that

the 30Boxes API would be shutting down in June 2022. If this API had been used in the project then a long lifespan would be impossible and demonstration difficult(Boxes, 2022).

Nylas calendar had extensive and easy to use documentation, and appeared to work with Google Calendar and Outlook. A trial developer membership was started for testing and the staff seemed keen to help, and communicated via email asking for the project's use case and offering advice. This project is a proof of concept requiring a quick to use, user-friendly API with a short setup time for small-scale projects. Upon further investigation, Nylas appeared to be aimed towards the corporate market rather than individual developers. The system seemed too in depth for what the project required, and needed working knowledge of the Google Calendar and Outlook API's before development with Nylas could even take place.

The online networking accounts for members of the University of East Anglia use the Outlook calendar API to present the users' upcoming events to them. This made the API seem like a good fit for the project, as it could have allowed for the scheduling algorithm to be tested in real world scenarios using University of East Anglia accounts. Upon further investigation, however, it became apparent that the university does not allow access to Outlook developer tools from its accounts. The documentation looked extensive and simple to follow, however, experimentation with Outlook took place alongside experimentation with Google's API, which had several advantages.

The Google Calendar API had extensive documentation, free developer accounts with plenty of features for small scale developers, and easy to follow tutorials for setting up a Google Developer account and setting up initial applications. Google's tutorials made it simple to set up a web app which could log in to a Google account and access calendar information. The documentation was clear, extensive, and was targeted at small-scale developers, which made the creation of a simple prototype easy. It has many benefits compared to its competitors, which made it the clear choice for the project. Google's quickstart system allowed for the quick creation of a prototype project, where it was simple for a developer to learn how to access and manipulate information on a user's Google Calendar.

The Google Calendar API was chosen for development of the project. Out of the compatible languages available for programming the application Java, JavaScript and Python were the primary options due to their versatility. Having the project as a web application would mean that it could be run on almost any device without prior down-

loads, if it was developed to completion and accessible via a web address. Java and Python applications would require the user to download them as executables before being able to run. This project is aimed towards students, and providing a web application which they could access via a link would mean easy and instantaneous access to the application without a download. By choosing to develop a web application, any final iterations of the project could be accessible on students' home computers, laptops, or library computers without downloading any files to their file explorers. For these reasons, it was decided that the project would be a web app programmed in HTML and JavaScript.

2.2. Overview of scheduling processes.

The finished scheduling system is a simple system made up a series of complex parts. It functions in two different ways, depending on whether the user is choosing to reschedule calendar events previously generated with the application.

The simplest processes the program can do is if the user wants to add new tasks to their Google Calendar without rescheduling any old tasks. The system downloads information from the user's Google Calendar and creates a master list of their upcoming events which will be used to keep track of which time periods new events can be scheduled. It then loops through each task that the user has inputted, and finds free slots in their schedule for new events to be created. It then schedules new work events into these slots, to define time periods for the user to work on their projects. Once this schedule has been created, all the new events are uploaded to the user's Google Calendar. This is visualised in the flowchart in Figure 1. If the user has previously generated a schedule, but their plans change and this results in removal of and/or changes to the future events on their calendar, they can request the system to reschedule any tasks that are in progress around their updated Google Calendar. The system checks which of their tasks are in progress, and calculates how many hours of work the user has already completed on them. This information is found so that the system can reschedule the remaining amount of work to the user's Google Calendar. The flow diagram in Figure 2 shows the updated process. When describing the system as a whole, these processes are simple. However, the sub-processes which are required for the functioning of each step in the flow chart present many complex problems.

The rules needed to create a schedule are primarily consisting of three things: The

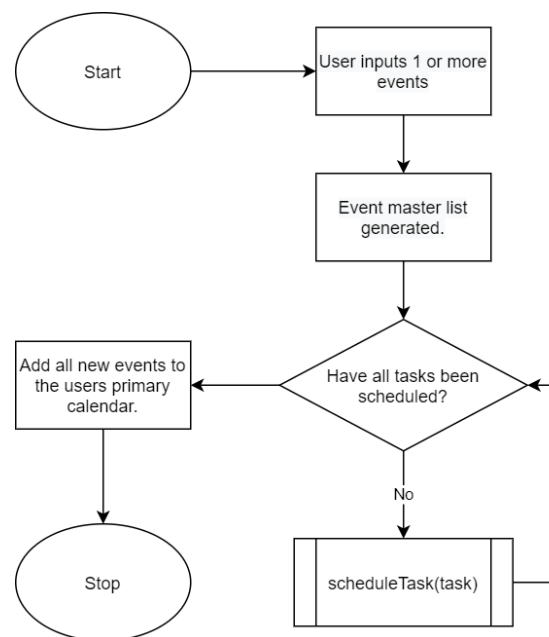


Figure 1: Flowchart showing the sequence of steps the program takes when the user is adding tasks without rescheduling. Some processes have been simplified into single steps for clarity.

tasks that need to be completed, the events/study sessions that need to be worked through in order to achieve the goals of the task, and the time periods which are breaks from work. From these ideas three new specifications of JSON object were developed and used alongside Google's specification for a calendar event. This application has Task objects which comprise of the rules needed to define a task, SimpleEvent objects which are used to track when the user will be working as well as whether new events can be scheduled in, and Break objects which contain the rules needed to schedule a user's breaks. The class diagram in figure 3 shows how these three types of object link with each other and Google Calendar events.

2.3. Tasks.

When a student is set an academic assignment, they are given the assignments name, the task/question to complete and it's due date, and the type of work it will require (e.g. programming, essay). These common prerequisites for a student to begin an assignment were the basis of Task objects, a way of describing an academic task which needs

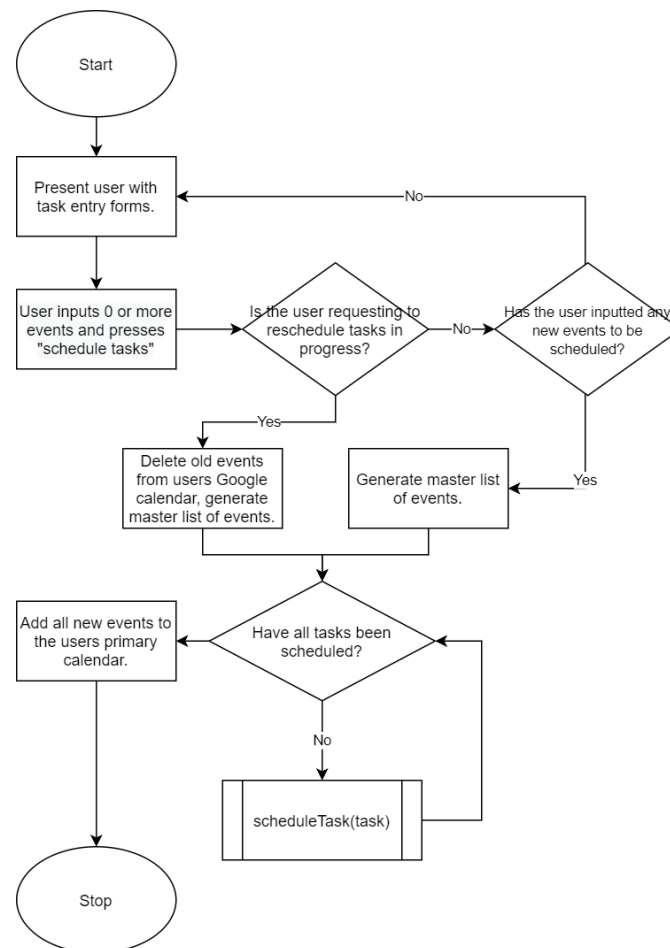


Figure 2: The second flow chart, showing the sequence of steps the program can take when the user is adding tasks, with the steps needed for rescheduling old tasks visualised. Some processes have been simplified into single steps for clarity.

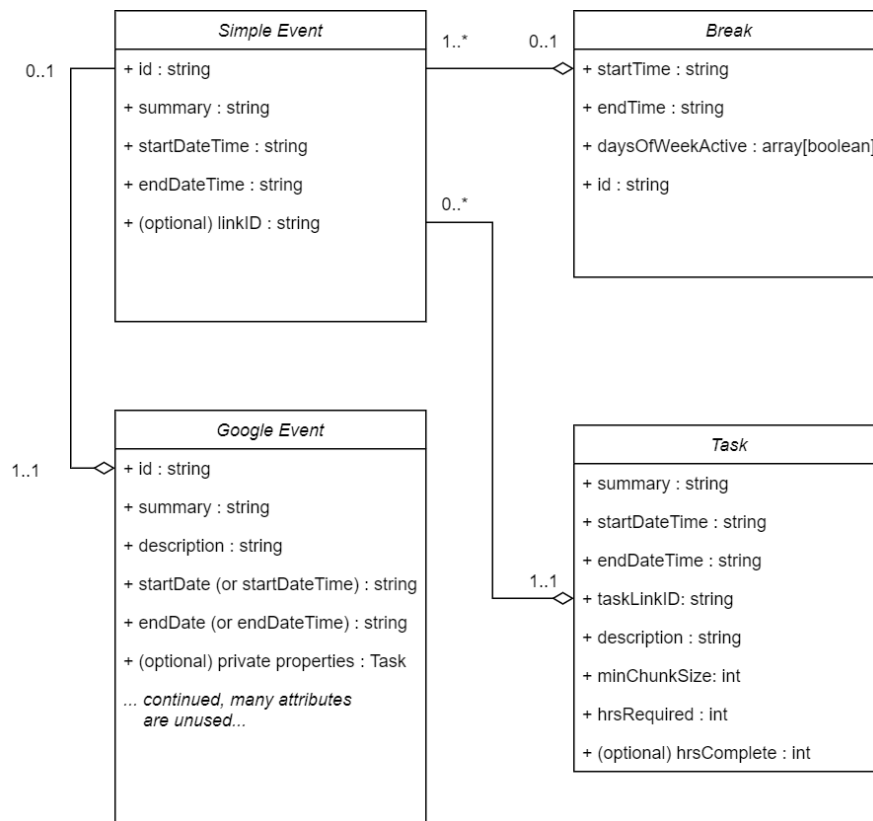


Figure 3: Class diagram showing task, Google Calendar event, simple event and break objects.

scheduling to a computer program.

If a student were to schedule a task as chunks of work on their calendar, they will first need to know the time periods in which they are scheduling. They need the lower bound, the date they should start scheduling work (usually the date the assignment is to be set or the present day), and the upper bound, the assignment's deadline. The student can tell from the type of work the task will require (e.g. programming, essay) the minimum length of time a study session would need to be to complete work in for this task. The student could schedule chunks of work of this length into free slots on a calendar. They'd also need to know how many hours to schedule for them to complete the task. With this information, a student could populate their own calendar if they wanted to, but it would be laborious to look at every day over potentially months of time to find free time, and exhausting to undo all of their work and reschedule things if

their plans were to change.

The same information can be used to describe a task to a computer program, which would significantly reduce the time required to schedule a task compared to a student. This information is the basis of this project's Task objects. Task objects have 7 attributes (and one additional optional attribute). Each task contains a summary (the Task's name), a startDateTime, and endDateTime, a description, a taskLinkID, a minChunkSize and an hrsRequired value. Additionally, some tasks may have the optional attribute hrsCompleted. StartDateTime and endDateTime both contain JavaScript's representation of dates and times denoting the day work on the task begins and the day work on the task ends. The minChunkSize attribute denotes the minimum work chunk length in hours of work events for this task, and hrsRequired denotes how many hours of work are required to be scheduled to complete this task. The taskLinkID attribute is an ID for the task.

The optional attribute hrsCompleted was added on later in the programming process to allow the application to reschedule tasks. The number of hours of work to be scheduled for a student to complete a task is stored in the task's hrsRequired attribute. If the application is rescheduling a previously scheduled task, it will calculate the hours of work already completed by the user and store it in the hrsCompleted attribute. This is so that if the task is being rescheduled, the number of hours remaining can be calculated by doing `hrsRequired - hrsCompleted`.

To enter tasks to be scheduled, the user fills in the form for adding a work task, and presses "create task". Upon pressing 'create task', the details are used to make a new Task object. This Task object is then added to the list of Tasks to be scheduled, which is named globalTaskList. The user can fill in and submit the form multiple times to add multiple Tasks to the list. When they press "schedule tasks", the tasks within the globalTaskList are then scheduled. The order that the tasks are entered into the system is the order of scheduling priority: the first task will be scheduled first, the second task second, etc. A screenshot of the form for adding tasks can be seen in Figure 4.

2.4. Simple events.

Originally, most of the processing and storage of the calendar data was going to be done remotely by the google calendar app. The Google Calendar API has built in ways for finding whether time slots which work chunks could be scheduled in are free or busy. Interactions between the client and the Google calendar API are quick, but not as quick

FForm for adding tasks:

Task name:

Start date:

Due date:

Hours required:

Minimum work block size:

Description

☐ reschedule old tasks

Figure 4: The form where users can add tasks.

as a completely client-side interaction. If an hour free time slot in a 9-5 Monday-Friday work week was to be found, it could take up to 40 API requests to find a free slot that week¹. Whilst the possibility of scheduling using Google's tools was being investigated, run times for extremely simple scheduling tasks were taking upwards of 15 seconds to process. If this were to be scaled up to a user trying to schedule a semester's worth of assignments, this task could take minutes, which would be a poor experience for the user.

To solve the issue of slow processing times, the project needed to perform checks of whether a time slot was free or busy locally. In order to do this, the data structures used to store calendar information locally needed to be decided.

To find whether a time period is free, the system needs to check the events closest to the time window and see if they clash with it. This meant that the project needed to include a way of storing a user's schedule.

The first method for storing the user's schedule locally to be investigated was iCalendar files. iCalendar files are a file type designed for the sharing of calendar and scheduling information across different platforms (Desruisseaux, 2009), which Google Calendar supports. In order to experiment with implementation and management of iCalendar files and events locally, a file was exported from a Google Calendar using a Google account. An investigation was lead into possible APIs which could manipulate and edit iCalendar files locally, and several were found: iCal.JS, npm ics, and Chowdhary's calendar slots. iCal.JS and npm ics both had methods for generating iCal strings and creating calendar files. However, their documentation contained no references to the iCal FREEBUSY request² or any sort of methods for checking if a time slot was free (ica, 2022). The ineffectiveness of the APIs lead to the project requiring its own local system for keeping track of the user's events.

If a human were to check if a free slot on a calendar was free, they would look at the calendar and see if the times taken up by other events clash with the slot they are checking. To do this, they need to know the start and end times of their previously scheduled calendar events. If we were to take the basic elements that a computer would need to track when a time slot was free or busy, these would be the start and end times of the other events close to the time slot we are checking. To make these events human readable, only a piece of text/a name describing each event is needed. These are the

¹8 hours a day * 5 days in a week = 40 hour slots to check.

²This is a method for checking whether a timeslot is free or busy in an iCal file.

ideas behind the creation of the class `SimpleEvent`. Simple events are made using the details of a Google calendar event, but stored in a simpler way with only the basic information stored. This object initially only had 4 attributes:

- `Summary` - The name of the task that this event stems from / the name of it's associated Google Calendar event.
- `ID` - The ID of the Google Calendar event that was converted to create it. (If created via conversion from a Google Calendar event)
- `startDateTime` - The date and time this event started at, stored in the ISO8601 format³.
- `endDateTime` - The date and time of the end of this event, also in ISO8601 format.

Google Calendar stores information about it's events as JSON objects. These JSON objects containing event information are referred to as Google Calendar events throughout this paper. Google Calendar events have two ways of storing their start and end times. For example, a start time can either be stored as the attribute `start.date` (Which is just a date in the ISO8601 format) or as the attribute `start.dateTime` (Which is a date and a time combined in the ISO8601 format). When processing a Google calendar event, both of the possible attribute names need to be checked for a date in case one is missing. This can cause difficulty during sorting/comparison processes because two variable names need to be checked, and the dates may be in slightly different version of the ISO8601 date format. Simple events have their start and end times stored as date and time combinations, in attributes named `startDateTime` and an `endDateTime` attribute. When a Google Calendar event is being converted into a simple event, the possibility of the dates being stored under alternative attribute names in slightly different formats is accounted for. Any dates are converted into date time combinations with the time being the earliest time during that day, 00:00. This simplifies any sorting and comparison operations.

As stated before, a computer would only need access the start and end times of events within a time window to see if a time slot was free. It is simple for humans to see if a

³This is a way of formatting a time and a date into a computer and human readable string (Newman and Klyne, 2002). This format is used by JavaScript's date objects as well as the Google Calendar API's calendar objects. (Mozilla, 2022a, 2021).

time slot on a calendar is free because the events appear in a chronologically ordered list, and we can look at the correct point in time and see if there are events clashing with our time slot. This same idea can be applied to a computer system. By storing simple events in a chronologically ordered list, the project can use the `startDateTimes` and `endDateTimes` of the different events in the list to track whereabouts in time it is inspecting. These ideas are what lead to the creation of the `globalEventManagerList` data structure, a global list of `SimpleEvents` kept in chronological order. When trying to find a free time slot, this list can be searched to find the events neighbouring the time slot to be checked. Once the closest events before and after the time slot had been found, the `startDateTimes` and `endDateTimes` of the events and the time slot are checked and compared to decide whether a time slot is free or not. The method to find a free slot, `checkFreeLocally`, will be expanded upon later.

2.5. Scheduling.

Scheduling begins once the user has entered their tasks into the system and pressed "Schedule tasks". Prior to scheduling, the program works out the number of work chunks that need to be entered onto the calendar for each task, and finds free slots spread across the `globalEventManagerList` for them to be scheduled.

The number of work chunks to be scheduled onto the calendar for each Task needs to be calculated. To work out the number of chunks needed for one Task, the number of hours required is divided by the task's minimum chunk size. Calculations are also done to check for a remainder. If a remainder is present, then during the scheduling a small work chunk the length of the remainder will be placed onto the user's calendar to ensure that all the hours they need to work are present on their calendar.

The `scheduleTask` function takes a task and the number of work hours which need to be scheduled for it, and schedules these onto the user's calendar. Within the function, it repeatedly works out which date it needs to schedule a work chunk on next, and then finds a free time slot within that day in which to place the work chunk.

For a schedule to be generated, an algorithm must be created to find dates for work chunks to be placed on. A system was developed which could evenly spread out a user's work events between their task's start date and their task's deadline. The pseudocode below was written to work out the number of days gap needed between work chunks for the chunks to be evenly spread out across a user's calendar.

Let x = number of work chunks required.

Let y = number of days between start date and deadline.

// g will equal the number of days gap between work chunks needed for even spread of work.

$$g = (Y - X) / X$$

Errors did occasionally occur as the outcome of the equation could be a decimal. These decimals needed to be rounded to a whole number, as this formula is for keeping track of whole days rather than fractions of days. Rounding down the number would cause build ups of work chunks during the beginning of a semester, whereas rounding up would build up work towards the end. The decision made was to use JavaScript's `math.floor` function on the output to round it down. This makes work chunks more tightly packed at the beginning of the task rather than the end, to reduce pressure on students during deadline season. This also ensures that all work chunks are definitely scheduled before the deadline, as it could be disastrous for students if work chunks were incorrectly spaced out and not enough work had been done before deadlines arose.

When a day has been decided for a chunk to be placed on, the application jumps through the working day's time slots hour by hour until it finds a free slot. The number of jumps that need to happen in a day is calculated by adding 1 to the minimum chunk length of the task, and subtracting this from the hour the user starts work, and then subtracting that from the hour the user stops work.

A design decision had to be made concerning what would happen if a chunk could not be entered on a specific day. If a day was full of other events, and the system were to add a chunk the next day and then move forwards g more days, then the next chunk would be scheduled a day later than it should be. If this happened 10 times in a row then the final chunk would be scheduled 10 days after the chosen deadline. The solution was to save the day each chunk was **supposed** to be scheduled on. Once a chunk had been scheduled, even if it had been scheduled a day earlier or later, we add g days onto the day scheduling was supposed to occur and attempt to place the next chunk there. This means that the search for a free slot can be wider than the specific days chunks should ideally be placed on, whilst making sure that everything is scheduled before the deadline.

There are many ways of spreading out or clustering chunks of work in a schedule, and it would be extremely interesting to expand this project and add multiple scheduling systems. However, a decision needed to be made for the project as to how the spreading out of work chunks would be done. The decision was made to spread work out over time rather than have blocks of a single type of work for days at a time as it is more keeping with a student's usual schedule, and ensures they have a range of work to do rather than a single task. Anecdotes from peers have revealed that some prefer to have the majority of work on a task finished towards the beginning of the time available. An investigation should be conducted into student's preferences for managing their own time. This information could be incorporated into a new, adjustable algorithm or into multiple new algorithms tailored to different studying styles.

2.5.1. checkFreeLocally

For the system to be able to schedule in events on a specific day, it needed a method for checking whether a time slot was free or busy. This fact meant that a function to perform this task needed to be created, that was able to find free slots using the data structures already implemented. `checkFreeLocally()` uses the `SimpleEvent` and `globalEventManagerList` data structures to check if a time slot is uninterrupted during a period of time on the user's calendar. Three arguments are entered into it: `localEventList`, `startTime` and `endTime`. `startTime` and `endTime` are the start and end dates and times of the time window to be checked. The argument `localEventList` is an array of simple events representing a schedule. `globalEventManagerList` is inputted here in this project.

To develop this method, blocks of time were hand drawn on a timeline, and this was used to work out all the possible configurations of events and free time slots, and what rules should be implemented to work out whether a slot was free or busy. To keep track of busy slots, events that are taking up time (events already in the user's Google Calendar, and events that have just been scheduled by the program) are stored in chronological order in the array `globalEventManagerList`. By having previously scheduled events stored in an array in chronological order, it is easy to keep track of which events precede and succeed each other.

A free time slot will (in most cases) have an event prior to it and an event succeeding it. If the end time of the preceding event is less than or equal to the start time of the time slot we are checking, and the start time of the succeeding event is greater than or equal

to the end of the time slot we are checking, then this is a free slot.

Because events are stored in chronological order in the array, a linear scan is performed through the array until the first event with a start time after the start of the time slot being checked is found. The aim of this is to find the two events that will proceed and succeed (or block) a free time slot. By looking for the first event with a start time **after** the slot for checking's start time, the system knows that this event directly succeeds (or blocks) the time slot being inspected, and therefore the previous event precedes (or blocks) the free slot. The first event with a start time after the slot for checking's start time will either be the first event succeeding the free time slot, or an event with a start time between the slot we are checking's start and end time: For the latter, the slot is not free, and false is returned. If the succeeding event does not clash with the time slot, then the end time of the previous event is compared with the start time of the time slot being checked. If the previous event's end time precedes the start time of the time slot, then this time period is definitely free.

The function cycles through each event in the `globalEventManagerList`. Each time it moves forward in time to the next event, this event is stored in a variable named `nextEvent` and the previous event is stored in a variable named `prevEvent`. This way, `nextEvent` can continuously be checked to see if its `startTime` is after the start time of the time slot to be checked. When this is found to be true, it is also true that `prevEvent`'s start time directly chronologically precedes the start time of the time slot we are checking. A code snippet showing of `checkFreeBusy` demonstrating this can be seen in the Appendix in Figure 12.

There are some other cases that needed to be accounted for when finding a free slot. If the `localEventList`'s length is 0, then the slot is free as the calendar is empty. If the `localEventList`'s first element's start time is greater than the slot we are checking's end time, then the slot is free (the free slot precedes all calendar events). As well as this, if no calendar event has a start time greater than the time slot's start time, then we check if the final event in the list's end time is less than the time slot's start time. If this is true, then the slot is free, and after every event in the list.

Another case to be accounted for is if an events time boundaries are completely within the time boundaries of another event. Events are stored in chronological order according to their start time. If an event with a `startDateTime` after the start time to be checked is found (stored in `nextEvent`), the previous event in the `globalEventManagerList` (stored in `prevEvent`) is inspected to see if it clashes with the time slot. If the previous event

in the list is completely surrounded by another event with an earlier start time, the time succeeding it may be marked as free because the preceding event in the globalEventMasterList's large time window is not taken into account. An image visualising this can be seen in Figure 5.

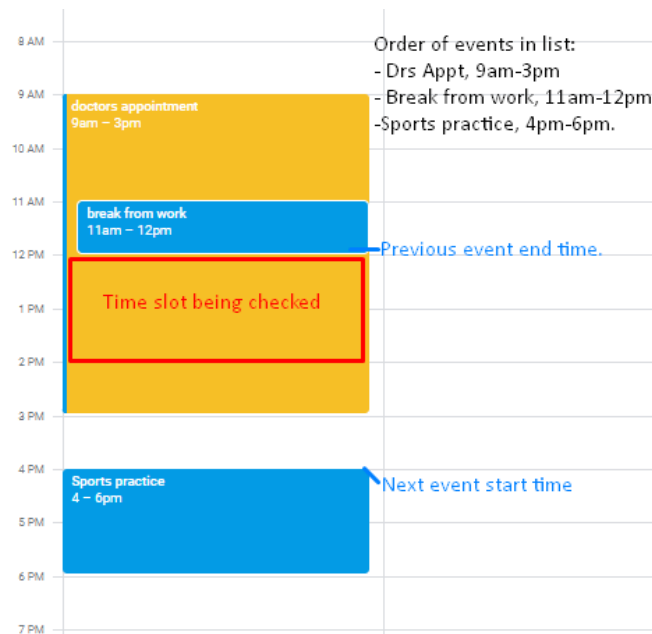


Figure 5: The doctors appointment occurs before the break from work in the list of simple events because it's start time is earlier. This could mean that when clashes are checked for previous event (in this case the break from work) and the next event (the sports practice) the clash with the doctors appointment may be missed.

The solution to this was to add a while loop ensuring that the `endTime` of `nextEvent` is always later than the `endTime` of the previous event. If the `nextEvent`'s `endTime` is earlier than the `prevEvent`'s `endtime`, then the `nextEvent` is replaced with the event after it in the `globalEventMasterList`, while the `prevEvent` remains unchanged. This sequence of checking and replacing is repeated until the `nextEvent`'s `endTime` is later than the `prevEvent`'s end time.

When a free slot for an event is found, a new `SimpleEvent` is created using the task being scheduled's name, as well as the start and end time of the free slot that has been found. This simple event is then appended to the `globalEventMasterList`, which is then

sorted using a custom sorting function, `compareLocalSimpleEvent`. Adding the new event into the `globalEventManagerList` and sorting it means that the local system for finding free slots has an up to date source of information regarding free slots in the user's calendar, ensuring that a second new event cannot be accidentally scheduled in at the same time as another one which was not previously on the user's Google Calendar.

At this point in the project there were two paths that could have been followed. Either the project could have been developed into a minimum viable product, or the scheduling algorithm could have been refined. By making the project more practical and by allowing it to cope with changes in peoples scenarios and adding adjustable work hours, etc, a minimum viable product could be created. The success of the project, whether it can manage time for students, can be assessed if a minimum viable product is created. Refining and adding more algorithms is possible, however it is not entirely possible to tell whether the problem has been captured properly or not without constructing a system that tackles then realities of life, rather than an more advanced algorithm which in theory could solve the user's problems if an application was constructed around it. The restrictions encountered by creating a minimum viable product help to define the problems that an advanced algorithm needs to solve. Without a minimum viable product the tasks to be performed and the extremities to be handled by a more advanced algorithm are not clear. For this reason, development of the project now focused on making the system practical and useful.

2.6. Google Calendar events.

Google Calendar events' have an incredibly extensive list of optional attributes (Google, 2022b). The user's schedule is initially downloaded using Google's `calendarList.list` method (Google, 2022c). When the user's schedule is initially downloaded it is saved as an array of Google Calendar events with a wide variation of information, much of which is unnecessary for scheduling. The Google Calendar events' summaries, ids, start times and end times are used to generate the initial sorted array of simple events representing the user's calendar locally, `globalEventManagerList`.

The Google Calendar API has a function for adding Google Calendar events to a user's calendar. Google calendar events can be passed to Google's `events.insert` function alongside information about the calendar it is to be added to. They will then be uploaded to the user's calendar (Google, 2022a).

In order for new event's to be added to the user's calendar in the project, Google Calendar events need to be generated locally when a free slot is found. This prompted the creation of the function `createLocalEvent`, which takes the details of the task being scheduled and the free slot that has been found as arguments. It returns a Google Calendar event with a name matching the task, and a time matching the free slot, ready to be uploaded through the API. These local Google Calendar events that are to be eventually uploaded to the user's calendar are stored in a list called `globalEventToBeAddedList`, and are uploaded all at once at the end of the scheduling process.

One of the Google Calendar event's many optional attributes is an attribute called extended properties. Extended properties are a way of hiding extra information as key-value pairs within a Google Calendar event's attributes. Extended properties can either be stored as private or shared (Google, 2022d). The ability to store shared extended properties is valuable for the project. It is valuable because information specific to the scheduling app could be stored secretly alongside each Google Calendar event's other attributes, and accessed and edited any time by the scheduling application itself.

The first action that was enacted when this information was identified was to adjust the `createLocalEvent` function. When a Google calendar event is created using `createLocalEvent`, the `linkID` of the associated task is stored within the Google Calendar Event's extended properties as a key value pair.

The `globalEventToBeAddedList` stores Google calendar events that are to be added to the user's Google Calendar using the Google Calendar API. When a free slot for an event is found, the event is stored as a Google Calendar event ready for sending to the API within the `globalEventToBeAddedList`⁴. All the events in the `globalEventToBeAddedList` can be added to the user's calendar in one go at the end of the scheduling process. This means that if the user is simply scheduling task(s), (as opposed to rescheduling old tasks as well), there are only two times the program communicates with the API: At the beginning of the process to generate the `globalEventManagerList`, and at the end of the process to upload the Google Calendar events it has generated in the correct places.

⁴As well as a simple event in `globalEventManagerList`. See section 2.5.1

2.7. Breaks.

The information needed to describe a regular break can be described as a set of rules. All that is needed to describe a break is the time period it takes up and the days of the week it is active. Multiples sets of rules describing breaks can easily be combined to quickly describe all the weekly breaks of a regular schedule. These ideas about the descriptions of regular breaks are the backbone of this project's Break objects. A break object has four attributes: An ID, a start time, and end time, and an array of 7 booleans representing the days of the week it is active.

A simple event being in the `globalEventManagerList` guarantees that nothing in the future will be scheduled during it's time window. This information influenced the design decisions deciding how a break would create a gap in the user's schedule. A break object has time in hours for a break to start, a time in hours for the break to end, and a list of the days of the week which it is active on. This, combined with the timespan that work is being generated in, is all the information required to generate simple events linked to the break that would populate the `globalEventManagerList` prior to tasks being scheduled. By populating the `globalEventManagerList` with simple events representing breaks, the time periods represented by the breaks are marked as busy if an attempt is made to schedule an event during the break's time. For example, if the user wanted to have breaks between 1pm and 2pm throughout Monday to Friday, and the timespan of their tasks was two weeks, then prior to task scheduling 10 simple events are added to the `globalEventManagerList` outlining breaks with start times of 1300 hours and end times of 1400 hours on the dates of Monday through Friday for those two weeks. After sorting the `globalEventManagerList` and beginning the scheduling of the user's task(s), these time windows would be noted to be busy if `checkFreeLocally` tried to check if a time window clashing with a break was free. This would mean that when tasks were scheduled they would not clash with the time the user had set aside for breaks.

The function `getSimpleBreakEventsList` was programmed to generate the simple events needed to represent all the breaks defined by the rules in a Break object.

the `getSimpleBreakEventsList` works as such. First, a break object and two dates representing the range of time the breaks will be spanning is inputted. Then, the dates of the Mondays of the weeks that cover the start date and end date are calculated, and stored in `startOfWeekAtStartTime` and `startOfWeekAtEndTime`. The calculation to find the first day of the week that another date falls into happens in a function custom for the

project named `getDateOfStartOfWeek`, which takes a date object as an argument and returns the date of the Monday of the week that date fell within.

Next, the number of days difference between these two dates is calculated using the custom function `getAbsDayDifferenceOfTwoDates()`. This number is then divided by 7 to get the number of weeks spanning over the time period. The number of weeks in the time period is stored in a variable named `weekRepetitions`.

Following this, the first week of breaks is calculated using the function `genFirstWeekOfSimpleBreakEvents`, and stored in the variable `prevWeek`. In order to make break events for the entire timespan of `startOfWeekAtStartTime` up to and including `StartOfWeekAtEndTime`, the events in `prevWeek` are duplicated `weekRepetitions` times, increasing the dates of each event by 7 for each time we loop to generate a new week. This is demonstrated in the pseudocode below.

```
prevWeek = genFirstWeekOfSimpleBreakEvents
```

```
simpleBreakEventList = new empty list
//This list will eventually contain all the break events
//needed to put this break in the schedule.
```

```
Loop 1( i in range weekRepetitions):
```

```
Create an empty array called newWeek.
```

```
//(looping through all the break events in the previous week)
Loop 2(j in range prevWeek.length):
```

```
Take the jth break event in prevWeek, and add 7 to the date,
and push it to the array newWeek.
```

```
// (This means by the end of loop 2, we created a new week
//of break events that are at the same times as the
//previous week but a week forward in date.)
```

```
Concatonate the newWeek array onto simpleBreakEventList to get
all the breaks for the weeks genarated so far.
```

```
Set the value of prevWeek to be newWeek, so that when the loop
```

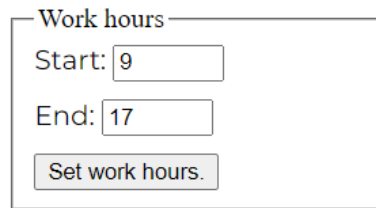
Form for setting work hours:A screenshot of a web form titled "Form for setting work hours:". The form is enclosed in a rectangular border. Inside the border, the text "Work hours" is at the top left. Below it, there are two input fields: "Start:" followed by a text box containing the number "9", and "End:" followed by a text box containing the number "17". At the bottom of the form, there is a button labeled "Set work hours.".

Figure 6: The scheduler's form for the user to set custom work hours.

restarts it will generate a week of break events using the most recently generated week.

Return `simpleBreakEventList` now that all the weeks of break events have been added.

2.8. Work hours.

The default work hours that events are scheduled within are 9am-5pm. Many students will have work hours differing from the default. For this reason, an additional form was added to the website named 'Form for setting work hours'. Users can enter their desired start and end times for the work schedule to fit between here. An image of this form can be seen in Figure 6.

Once the user has filled in their desired hours and pressed the 'Set work hours' button, the function `addWorkHrsToGlobals` saves these times into two global variables, `globalWorkHrsStart` and `globalWorkHrsEnd`, which keep track of the user's work hours across the application. For example, if the user worked from 8am to 4pm, the value of `globalWorkHrsStart` would be 8 and the value of `globalWorkHrsEnd` would be 16. These numbers default to 9 and 17. When the user presses the 'schedule tasks' button, these work hours will be taken into account, and work will only be scheduled between these hours.

2.9. Rescheduling.

In the project's previous state, the project had the functionality to schedule multiple tasks at once, schedule breaks and have adjustable work hours. It could not be considered truly a time management system, merely a tool to add events to a user's Google Calendar. For the project to be a practical product, users must be able to reschedule tasks that have previously been created by the program. If a user has had changes made to their schedule, they would have to manually delete all the items from their Google Calendar and then manually retype every task that they were doing. They would have needed to enter new hours requirements, and start dates, which they would have had to have calculate themselves. For this reason, the next step for the project was to provide the user with the ability to reschedule events the project had previously scheduled around unexpected changes a user has made to their Google Calendar. An example of a scenario like this would be if a user had previously generated a work schedule, and then found out later that they had a doctor's appointment stopping them from completing a day's work tasks. The program would need to regenerate the user's schedule to fit in work events around their new calendar.

If a user were to replan their schedule manually, they would need to work out which tasks were currently in progress, and how many hours of work they have already completed for these tasks. This would allow them to only schedule the remaining hours which they had not completed of in progress tasks into their calendar. There are two pieces of information needed here: the tasks that are in progress, and the user's progress through them. With this information, the project could reschedule the remaining number of hours for each task as events on their calendar, whilst deleting the old outdated events from the schedule that the application had previously made. Functionality was added to the project so that these values could be calculated by the application itself before automatically rescheduling the user's in-progress tasks, if they desire.

In order for the program to know which tasks it had previously scheduled, it needs to retrieve this information from the Google Calendar. The only information originally stored about each Task was its taskLinkID in the Google Calendar event's extended properties. The decision made for the project was to store all the details of a task in the extended properties of a Google Calendar event.

When a free slot for an event is found, the system creates a SimpleEvent and a Google Calendar event. The SimpleEvent is to keep the local schedule up to date, and it is stored

in the `globalEventManagerList`. The Google Calendar event is the object that will be sent to the Google Calendar API for uploading to the user's calendar. This Google Calendar event is created using the function `createLocalEvent`, which originally was passed the tasks name (as the Google Calendar event's name), the time window of the free slot that had been found, and the tasks `taskLinkID`. In order to allow Task objects to be retrieved, the `createLocalEvent` function was updated, so that a Task object was now passed in as an argument. All of the properties of this task object are then stored as attributes in the Google Calendar event's extended properties. To find whether an event was previously created by the application, it would need to use `javascripts.hasProperty` function to check whether the event has a link ID within its shared extended properties. If this returned true, then the application can safely inspect the event's extended properties and use these to retrieve and recreate the task object that was originally used to create the event.

With these changes to the `createLocalTask` function in place, it was possible to look at the extended properties of an event, and find out whether it was created by the application and if so, find the details of the task that created it. This information is then used to create a complete list of the user's unique tasks in progress.

2.9.1. listAllTasks

Development began on creating a function that could find the tasks that created the upcoming event's on the user's Google Calendar to be rescheduled. This function is called `listAllTasks`. The function inspects the users upcoming events, and interprets them to work out find which unique tasks the user entered in previously that made the program create these events. The function communicates with the Google Calendar API to receive a list of all Google Calendar events from the current time onwards. The function checks whether any of the upcoming Google Calendar events have an `extendedProperties` attribute. If an event does, the function then checks to see if this event has a `linkID` extended property, within the shared category. If this is true, then it means that the event being inspected was previously added to the user's Google Calendar by the application. The event's `linkID` is then used as a key, and the rest of the extended properties for the event (which are all the attributes of the original Task object which, in being scheduled, created the event) are used as a value in a key-value pair. This key-value pair is then saved in a map called `tasks`. A map is used here so that no duplicates of the same task are stored. A set was originally used for this data structure, but an error was occurring

which meant that JavaScript interpreted identical Task objects as having differences. As this was meaning there were duplicate tasks within the set, the data structure was changed to a map which only used the Tasks' linkIDs as their unique identifiers, creating? a map of key-value, linkID-taskObject pairs. The function then retrieves all the unique tasks from the map, and returns them in a list.

For Tasks that are in progress to be rescheduled, the number of hours already completed by the user for each Task needs to be calculated, so that the correct number of work events needed for the user to complete the Task is scheduled. The number of hours completed by a user on their task is the number of hours of work events for the Task that exist in the past on the user's calendar. To calculate the number of hours of work that need to be scheduled in the future, the number of hours already completed is subtracted from the original Task's hours required. This prompted the development of the function `getHrsCompleted`.

The `getHrsCompleted` function takes a Task object as an argument, and then checks the user's Google Calendar to see how many hours of this Task have already been completed. A request is made to the Google Calendar API to return a list of all Google Calendar event items, in a time span that starts at the Task's `startDateTime` and ends at the current time and date. This time window contains all events associated with the inputted Task that are in the past. These events are then looped through, and events which have a matching linkID in their extended properties are recorded. The length in hours of each of the recorded events is calculated by subtracting each event's `startDateTime` from its `endDateTime` and converting this value to hours. This number is then added to a running total of hours of work for this task that have previously been completed. This total number of hours already completed for this task is then returned.

Providing the scheduling algorithm with a whole number of hours to schedule provides the best results for the user. If the user is currently in the middle of a task in their timeline when they are trying to reschedule it, the question arises about how to handle the fraction of a work chunk that the user is mid way through. Scheduling in too few hours in the future could lead to a negative impact on work, as the user may not have completed as many hours of work as they aimed to. This could happen if the program were to round up and give the impression that the user had completed more hours than they really had. Rounding down, and thus scheduling in extra hours is more appropriate, as it is better for a user to complete their work early with time to spare rather than to complete their work late and miss a deadline. For this reason the decision was made that

the number of hours completed for a task that is currently in progress should be rounded down to the nearest hour.

When scheduling a task, the `scheduleTask` function needed a way of determining whether the task being scheduled was new, or a previous task being rescheduled. If a previous task is being rescheduled, then schedule task needs to schedule in less hours of work than originally entered by the user. A solution to this could have been subtracting the number of hours already completed (calculated by the `getHrsCompleted` function) from the task's `hrsRequired` value, before using the Task as an argument in `scheduleTask`. If this were to be done, the task being stored in the generated events' extended properties would have information different from the original task that was entered by the user. Problems could arise if attempts to access the details of the original task were made in the future, as the task returned would be the edited version of the task used for rescheduling.

As an alternative to this, changes were made to how task objects are structured. Task objects may now have an optional attribute, `hrsCompleted`. After the Tasks have been retrieved by `listAllTasks`, the number of hours that have already been completed by the user are calculated using `getHrsCompleted`. Before these tasks are added to the `globalTaskList`, these values would then be stored in each Task as a new attribute called `hrsCompleted`. When `scheduleTask` is scheduling a normal task, that is not being rescheduled, the number of hours required to be scheduled is put into a variable called `hoursRequired` by retrieving the value of `task.hrsRequired` (the tasks `hrsRequired` attribute). This variable is then used for calculating the number of chunks to be scheduled. `ScheduleTask` has been updated, and now the JavaScript function `hasOwnProperty(hrsCompleted)` is used to see if this task has an `hrsCompleted` attribute. If the attribute is found, then the task is currently in progress, and the scheduling system needs to schedule a reduced number of events: the `hoursRequired` variable is now assigned the value of `task.hrsRequired - task.hrsCompleted`.

When an event has been rescheduled and a new Google Calendar event is being created, the `hrsCompleted` attribute is not uploaded within the extended properties. This is so that the attributes of the Task stored within this event remain as they originally were when the user first created the task in the past.

During the rescheduling process, the old events which are being rescheduled need to be deleted from the user's Google calendar. There are two reasons for this. The first is that once the new schedule has been created, the old, redundant events need

to be removed as new events have been created to replace them. The second reason is so that the `globalEventManagerList` does not have redundant events in it during the scheduling process. If the old events remained in the `globalEventManagerList` during the rescheduling process, then `checkFreeBusyLocally` would return that time slots that clash with these old events are busy, even though the events are going to be removed.

To delete an event from the Google calendar, the value of its ID attribute is needed. The corresponding event can then be deleted from the user's Google calendar by sending the ID to Google's `events.Delete` method (Google, 2021a). Because of this, in order to delete the old events from the user's Google Calendar, their IDs must be acquired.

To find the IDs of upcoming events linked to a specific task, the function `getEventIDsOfTask` was developed. This method takes a task as an input argument, and returns a list of upcoming Google calendar event IDs of events that have been created to contribute to completing this task. The basic premise of this function is to search through a list of upcoming events and store the IDs of the events linked to the task. To get the upcoming events, a request for all the Google Calendar events within a specific time window must be sent to the Google Calendar API. If the time window specified spans from the present day up until the deadline of the task, then it is guaranteed that all the upcoming events for the Task in question will be within the timespan. This means that the area searched is as small as possible while ensuring no events are missed. For this reason, a request to retrieve all the upcoming calendar events between the present day and the Task's deadline is performed. This list of events is then looped through, and each event is checked to see if there is a link ID attribute within their expanded properties. IF there is a link ID, and it matches the link ID of the inputted task, then the Google Calendar event ID is added to a list named `IDs`. Once all the Google calendar events have been inspected, all the `id`'s of the upcoming Google calendar events linked to this task are stored within the `IDs` array, and the array is returned.

Originally this method was used on each Task in progress to get a total list of the eventIDs to be deleted. Each ID on the list would then be sent to the Google Calendar API one by one for deletion from the user's calendar, so that the updated calendar could be downloaded and stored locally as `simpleEvents` within the `globalEventManagerList`. However, this strategy caused several problems for the project.

Most programming languages are synchronous, meaning that each line of code is executed one after another, with the next line of code only executing when the previous has finished. Fetching data from across the internet can take time, so JavaScript has

many asynchronous functions (Mozilla, 2022b). If one of these functions is run, the succeeding lines of JavaScript may be executed before the function has finished executing. Many of the Google Calendar API methods are asynchronous. A request for the API to provide some data or do an action is made, the request is fulfilled on an external Google server, and then the value from the function is returned. The JavaScript keywords ‘async’ and ‘await’ can be used to ensure that the program will wait for a function to finish running before continuing.

The act of deleting from the Google Calendar is a lengthy and asynchronous process. It requires many communications between the application and Google’s servers, which takes a lengthy amount of time compared to the almost instantaneous calculations that can be performed locally. JavaScript would send the delete requests to the Google Calendar API, and then run the rest of the process while waiting for the results to be returned. Even with the use of async and await, the `globalEventManagerList` would sometimes be generated before all the old events had been deleted. The `globalEventManagerList` has not updated properly, so `checkFreeLocally` still interprets the times taken up by these old events that should have been deleted as busy. The old events would only finish deleting from the user’s Google Calendar after the faulty schedule had been produced and uploaded. This meant that the user’s schedule was left messy and broken, with empty slots remaining in the calendar where the old schedule would have been, with new events messily placed around them.

The deletion of the events is needed for two reasons. To clear the `globalEventManagerList` of redundant events, and to clear the Google Calendar of old event’s so the user doesn’t see them any more after a reschedule. The errors are being caused because the deletion process is too slow for the `globalEventManagerList` to be updated before the local scheduling process begins. If the slowness of the initial deletion process was eliminated then the `globalEventManagerList` would be up to date before the scheduling begins. For this reason, development began on splitting up the deletion process into two parts: the deletion of the old events directly from the `globalEventManagerList` after it has been generated using the user’s Google Calendar data, and the deletion of the events from the users Google Calendar itself at the end of the process. Deleting the events directly from the `globalEventManagerList` removes lengthy API communications at the start of the function, and ensures that the contents of the `globalEventManagerList` will always be correct before the scheduling process begins. Deleting the old events from the user’s

Google Calendar after the rest of the scheduling process is complete means that the main scheduling process works correctly, as it is not dependent on asynchronous activities to be completed fast for it to give a correct outcome. Instead of relying on the potentially slow action of deleting the events from the Google Calendar to ensure our scheduling is correct, the process is shifted to the end and the deletion of unnecessary events from the `globalEventMasterList` is handled locally instead. This change means that old events are always deleted from the schedule, which means it generates correctly.

Changes needed to be made to allow the correct `SimpleEvents` to be deleted from the `globalEventMasterList`. Originally, the `globalEventMasterList` could not be searched through for simple events with matching link IDs because `SimpleEvents` did not have `linkID` attributes. To fix this, an edit was made to the structure of simple events. They could now have an optional attribute, `linkID`. When a simple event is being created through conversion a Google Calendar event during the initial process of downloading the schedule, it is inspected to see if a link ID is in the extended properties. If one is present, then the simple event being created is made with a `linkID` attribute, and this attribute contains the value of the link ID of the Google Calendar event being converted.

However, the `eventIDs` of the events to be deleted still need to be acquired so they can be deleted from the user's Google Calendar. To delete the simple events from the `globalEventMasterList`, the simple events to be deleted are those with link IDs matching the tasks for deletion. When a simple event is selected to be deleted, but before it has been deleted, its Google Calendar event `id` value (which it inherited from the Google calendar event which created it) is saved into an array called `globListAndDeletedEvent.deletedEventIDs`. After all scheduling is completed, these `eventIDs` are passed to the Google Calendar APIs `events.delete` method to be removed from the user's calendar.

In order to connect to the Google Calendar API, this project was ran through a Google Developer account, and was linked to the Google Cloud Platform. Because this project was run using a free developer account, there are limits on the number of API requests that user's can make. Google Calendar API users have a maximum of 10 API requests per second to prevent requests being sent too frequently, (Google, 2021b) and 100 API requests per 100 seconds to prevent there being too high a volume of requests. When too many requests are sent at once, the website will throw a 403 `rateLimitExceeded` error (Google, 2022e). These errors are sent by Google as responses to API requests when the API's usage limits are being exceeded. Google developers can fix this by paying to increase their quotas. Alternatively, Google recommends implementing methods

to slow down the rate requests are sent, such as exponential backoff, a system which periodically retries failed requests, with an increasing wait time between retries every time(Google, 2022f).

Unfortunately, for large scheduling tasks, the high number of requests at the same time was too much for a free Google Developer project. For this reason, I followed a coder's guide to implementing a function called sleep, which pauses the running of JavaScript. Because large scheduling tasks can create more than 10 API requests a second, and this project was being developed on a free account, the rate of requests being sent during the uploading and deleting stages needed to be slowed down. The sleep function was implemented to pause the running of the program for a fifth of a second between each API request. This change means that the scheduler can function correctly without using up its API quotas. This is only a temporary fix that has had to have been implemented due to the limits imposed by Google on free developers.

All of these functions are combined together in the function, `scheduleAndAddArrayOfTasks`. This function performs all of the steps in the flowchart in Figure 2 except for presenting the user with the forms for information entry. This function creates a list of all the tasks the user has entered and any to be resubmitted, generates the `globalEventManagerList`, saves the eventIDs of any events to be deleted, generates and schedules all breaks, schedules each task using `scheduleTask`, uploads the generated Google Calendar events to the user's Google Calendar, and finally deletes any Google Calendar events that are to be deleted from the user's Google Calendar.

A brief summary of my project notebook can be found in Appendix B.

3. Outcome, Impacts and Evaluation.

This project aimed to develop an application which could provide a student with a professional study plan, as if they had a personal assistant. The reasons for this were to help tackle the issue of students' poor time management and poor perceived control of time affecting their grades. All of the Must-have and Should-have requirements within the MoSCoW analysis have been achieved, demonstrating that the project has been a success and has succeeded at not only demonstrating that scheduling work onto a user's calendar is possible, but that a system implementing this can be practical and useful.

The market of student scheduling apps had none with capabilities developed to the level of this project. All were calendaring tools with limited functionality and no au-

tomation or generation of schedules. An impact of this project has been proving that a working solution to the problem of scheduling students' studies is possible. At this stage on the pathway towards full development, the project is a step closer to bringing effective and fast time management to all students.

3.1. Testing

Functional and user input testing was performed. The tables of test cases and their results can be seen in Figure 13 and Figure 14 in the Appendix.

In order to connect to the Google Calendar API, this project was ran through a Google Developer account, and was linked to the Google Cloud Platform. Because this project was run using a free developer account, there are limits to the number of API requests a program can make. These quotas have limited the projects ability to perform well for large volumes of work, because it cannot successfully upload high quantities of event's to the user's Google Calendar. Because of the 10 requests per second limit, a temporary change was made to the sections of code that send requests to the API, halting the running of the program for a fifth of a second between each requests, using a sleep function developed by a user of the alias Energetic Eagle(Eagle, 2019). While working for smaller bursts of API requests needed for most scheduling operations, this allows the program to stay within Google's limits. However, the operational limit for the number of requests the program can make seems to be 70 operations per complete scheduling process.

The aim of this project was to tackle the problems caused by student's poor time management and poor perceived control of time through a scheduling program. The ideal product to match these aims is outlined in the project's MoSCoW analysis and ideal product description, which can be seen in Appendix A. Everything non-UI themed within the ideal product description was realised, and all of the Must-have and Should-have goals within the MoSCoW analysis have been achieved. Following this are screenshots showing proof of the success of the concept, with explanations as to how these screenshots show that the project successfully meets its goals.

Figure 7 shows the user interface of the scheduling system. This is the interface where the user can enter in multiple tasks, and schedule them all to their calendar at the same time. This interface clearly proves the product has an 'Interface to allow users to add commitments to a schedule', as well as the 'ability to add multiple commitments at

Figure 7: The scheduler's UI.

once'; both quotes are two points from the MoSCoW analysis, which is outlined in the appendix.

Below in Figures 8 and 9 are three screenshots demonstrating the projects ability to schedule items to a user's calendar, the results of test 6 in the testing table. There are before and after images showing the contents of a calendar, alongside the filled in UI used to add new tasks with time constraints and breaks. For this demonstration, three tasks were inputted. All three tasks had start dates of Monday 23rd May, and due dates of Sunday 5th June. The first task was 'Write dissertation', which had a minimum work chunk size of 3 hours, and 18 hours of work to complete. The second task was 'Revise for ML exams', which had a minimum work chunk size of 2 hours, and 10 hours of work to complete. The third and final task was 'read mathematics book', which had a minimum work chunk size of 1 hours, and 5 hours of work to complete. The work hours were as between 9am and 5pm, and the breaks applied were a lunch break between 1 and 2 pm for Monday-Saturday, and Sundays as off work completely. The UI after the user has chosen these settings, but prior to the user pressing the 'Schedule tasks' button to schedule them, is shown in figure 9.

Figure 8 show the first upcoming week of the user's calendar before and after scheduling these three tasks. In the appendix, Figure 11 shows the second week before and after scheduling. This week was left blank originally so the schedulers ability to spread out workloads and avoid scheduling tasks in breaks can be seen. This image clearly shows the scheduler staying within the user's 9am to 5pm work bounds, and avoiding scheduling work within their 1pm - 2pm break.

The placement of the events on the calendar and their even spread across the two weeks demonstrates the program meeting the requirements 'Ability to calculate the number of work hours / week that a piece of work will need' as well as 'Ability to place these work hours onto a calendar'. The successful application of events onto a user's Google Calendar after they have logged into the application also shows that the project meets the requirement of 'Outlook / Google Calendar connectivity'. The UI providing the user with an option to set their work hours meets the requirement 'Ability for calendar constraints to be set. E.g. no work after 9pm', and the work chunks placement onto the calendar within these boundaries gives evidence of meeting the requirement 'Ability to apply items to a calendar within according to these constraints' being met. The differing lengths of the work chunks for each task demonstrates the meeting of the requirement 'Ability to adjust length of work chunk depending on the activity', and the order that the requirements are entered into the application being the same as the order they are scheduled onto the user's calendar demonstrates that the application meets the 'Ability to adjust placement of work hours based on priority' requirement.

Figures 15, 16, and 10 contain screenshots which demonstrate the scheduling application's ability to reschedule work around unexpected calendar changes. Figures 15 and 16 can be found in the Appendix. A doctors appointment on Wednesday the 25th has been entered into the user's calendar. Doing this has removed a 'Revise for ML exam' event and a 'Write dissertation' event from the calendar. Figure 10 shows the user interface after the user has entered their breaks and work hours, and selected that they would like to reschedule their tasks. Figures 15 and 16 show the first and second weeks respectively of the user's calendar before and after they have rescheduled their events. Even though events were deleted from Wednesday 25th, the correct amount of events have been rescheduled across the two week period, within the user's time limits and around their breaks.

This rescheduling of the user's commitments around new scenarios demonstrates the requirement 'Users can edit commitments which are on the schedule'. The scheduler's

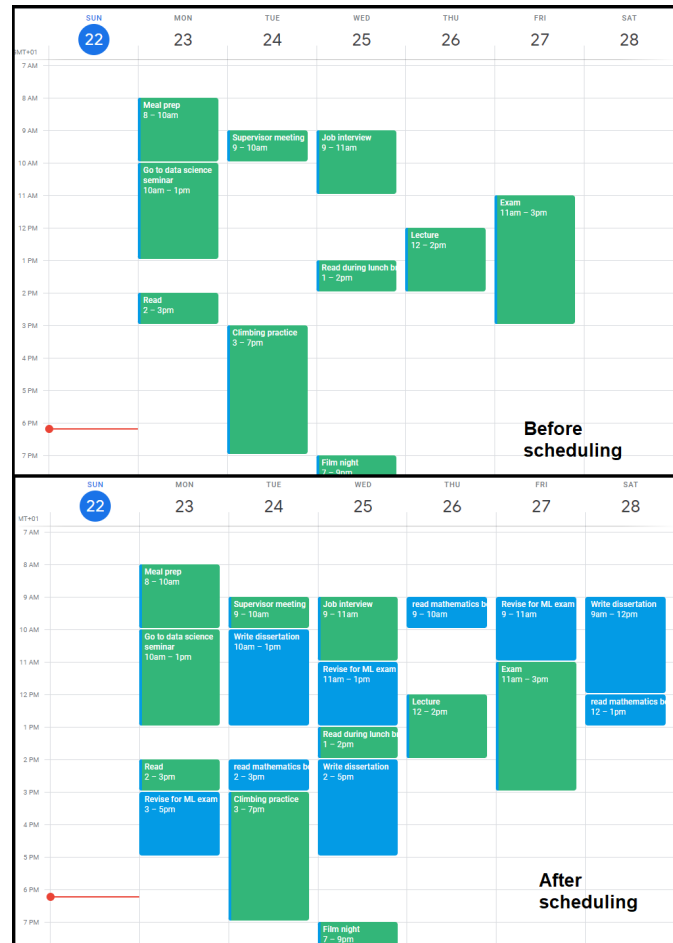


Figure 8: The first upcoming week of the user's Google calendar before and after scheduling of new tasks. For clarity, all pre existing events have been coloured green.

The figure shows a web application interface with three main forms and a footer. The first form, 'Form for adding tasks', contains fields for 'Task name' (read mathematics book), 'Start date' (23/05/2022), 'Due date' (05/06/2022), 'Hours required' (5), 'Minimum work block size' (1), and 'Description' (do the maths reading). It also has a 'reschedule old tasks' checkbox and 'Create task' and 'Schedule tasks' buttons. The second form, 'Form for setting work hours', has 'Start' (9) and 'End' (17) time inputs and a 'Set work hours' button. The third form, 'Form for adding breaks', has 'Break time' inputs for 'Start' (13:00) and 'End' (14:00), a list of days of the week with checkboxes (Monday through Saturday are checked), a 'Create break' button, and a 'Schedule break(s)' button. The footer displays the text 'Writing the dissertation', 'ML revision', and 'do the maths reading'.

Figure 9: The user interface after the user has entered their tasks, work hours and breaks, but prior to them pressing the button to schedule their tasks.

ability to retrieve the user's schedule from their Google Calendar so it can be used to reschedule shows the programs ability to have 'Google calendar syncing'. The program does meet the requirement of 'Functionality for allowing the storing of a schedule', but this is through its ability to sync with the Google calendar and retrieve information about it's previously created schedules from there.

The documentation, user guides, technical guides, and most up to date code can be found on this project's Github, at <https://github.com/wbosley/Scheduler>.

3.2. Limitations of the design, and how they can be addressed.

The focus of this project has been to prove that the concept of automated time management for students is a possibility. The focus of development has been on making sure the project is a working solution to the problem, rather than creating a perfect product. For this reason, the program has deficiencies in that it lacks back-end programming and

The figure shows a web interface with three main sections:

- Form for adding tasks:** Contains a 'Sign Out' button, a 'Task name' input field, 'Start date' and 'Due date' (both dd/mm/yyyy with calendar icons), 'Hours required' and 'Minimum work block size' input fields, a 'Description' input field, a checked checkbox for 'reschedule old tasks', and 'Create task' and 'Schedule tasks' buttons.
- Form for setting work hours:** Contains a 'Work hours' section with 'Start' (9) and 'End' (17) input fields, and a 'Set work hours' button.
- Form for adding breaks:** Contains a 'Break time' section with 'Start' (13:00) and 'End' (14:00) input fields with clock icons, a section for 'Which day(s) of the week is this break active?' with checkboxes for Sunday through Saturday (Monday-Saturday are checked), a 'Create break' button, and a 'Schedule break(s)' button.

Figure 10: The user interface after the user has selected the option to reschedule, and added their breaks, but prior to them pressing the button to (re)schedule their tasks.

security. The scheduling calculations, and systems to connect to the Google Calendar API can currently be found in the HTML. This unfortunately means that if the user closes the webpage before the scheduler has scheduled all tasks to the user's Google Calendar, it will fail. Because the focus needed to be on creating a proof of concept that solves the key problems, the handling of the regular problems of security and serverside calculation have been neglected. This limitation could be addressed by having the front end communicate with a back end written in Node.js, which would do the calculations and make the upload requests externally, as well as having information such as API keys hidden from users.

A criticism I have for this project is it's inability to schedule around time periods differing to a whole hour. Pseudocode for a function to do this was created. The function would see by how many minutes a time slot clashed with a calendar event, and try moving the time slot by that many minutes and attempting to schedule again. However, it was never implemented as the importance of making the system able to handle

real world scenarios such as rescheduling outweighed it in importance. To implement support for time periods differing from whole hours, I would continue to develop this pseudocode into a working function.

The project was limited by Google's API request quotas. These quotas have limited the project's ability to perform well for large volumes of work, because it cannot successfully upload high quantities of events to the user's Google Calendar. In future developments of this program, more investigations into being granted quota increases should be made, including the linking of a billing account to the Google Cloud Platform if necessary. If a backend using Node.JS was implemented in the future, exponential backoff could be implemented, and the issues caused by this being a lengthy process, such as that if the user closed the webpage the processing would stop, would be handled by the backend even if the user had closed the web page.

In order to create a minimum viable product, focus has been primarily on creating a practical application rather than a polished appearing one. Because of this, the UI for the program is basic. Future iterations of this program should have aesthetically pleasing UI with accessibility options and mobile compatibility.

3.3. Conclusion

It has been observed how students' academic success is intimately linked to their ability to manage their time, and that the majority of students fail to do enough work per week. There was therefore a pressing need to create a system that would allow for students to manage their time through an automatically generated schedule of work. This project shows that a proof of concept of this system has been implemented successfully, with all of its Must-have and Should-have MoSCoW requirements met. The system has been demonstrated through a table of tests, with screenshots presenting how it meets the requirements. The project has a scheduling system which spreads the user's work out between the work's start date and deadline, but the option to use alternative scheduling algorithms could be implemented in the future. The system integrates with Google Calendar, but could be implemented for Outlook Calendar also. We hope that developing this proof of concept is a small step towards helping students in universities, and maybe all stages of education, have greater productivity.

References

- (2022). 3.2.9. free/busy time type.
- Adams, R. V. and Blair, E. (2019). Impact of time management behaviors on undergraduate engineering students' performance. *SAGE Open*, 9(1):2158244018824506.
- Bosley, W. (2021). Progress and engagement report.
- Boxes, . (2022). 30 boxes api.
- Cambridge (2021). Teaching and learning.
- Cuseo, J. (2011). The "big picture": Key causes of student attrition key components of a comprehensive student retention plan. pages 1–2.
- Desruisseaux, B. (2009). Internet calendaring and scheduling core object specification (icalendar). RFC 5545.
- Eagle, A. (2019). Top answers by energetic eagle.
- Frobisher, A. (2020). Degree dropouts.
- Google (2021a). Events delete.
- Google (2021b). Limits and quotas on api requests.
- Google (2022a). Create events.
- Google (2022b). Events.
- Google (2022c). Events: list.
- Google (2022d). Extended properties.
- Google (2022e). Handle api errors.
- Google (2022f). Retry strategy.
- HESA (2022). Non-continuation summary: Uk performance indicators.
- Mozilla (2021). Date.prototype.toISOString().

Mozilla (2022a). Date() constructor.

Mozilla (2022b). Introducing asynchronous javascript.

Neves, J. (2019). Uk engagement survey 2019.

Newman, C. and Klyne, G. (2002). Date and time on the internet: Timestamps. RFC 3339.

OU (2021). Finding time to study.

Pokhrel, S. and Chhetri, R. (2021). A literature review on impact of covid-19 pandemic on teaching and learning. *Higher Education for the Future*, 8(1):133–141.

Roberts, M. (2020). Universities with highest and lowest dropout rates.

Statista (2020). Share of individuals using online learning materials and doing online courses in great britain in 2020, by age and gender.

Technavio (2021). E-learning market in uk by product and end-user - forecast and analysis 2021-2025.

TUOS (2021). Time management.

A. Requirements: MoSCoW analysis.

MoSCoW analysis.

Ideal product description:

The ideal product would be an application that allowed users to quickly add all their usual commitments to a calendar, either manually or by syncing with a calendar application or by importing an iCalendar file. Then, they can quickly upload all their assignments and work. The application would then use the weightings and deadlines of each task, alongside constraints set by the user (e.g. no work between 6pm and 9am) to populate their calendar accordingly. Urgent and time-consuming tasks would receive priority, whilst also scheduling regular instalments of other work so that progress is still made on other upcoming assignments. The work schedule generates around the users' prior commitments. If a new assignment is added to the schedule after the initial setup, the upcoming schedule should rearrange accordingly.

The users upcoming schedule would be displayed on an aesthetically pleasing, easy to understand calendar. The user could be able to select between viewing a day, a week, or a month of the schedule. The user can colour coordinate calendar items. If a user has not completed a work chunk, they should be able to select the incomplete chunk and the application will add it into their schedule in the future. The system will alert user to upcoming deadlines. The user can also add commitments by click and dragging to mark an area of the calendar to be taken up by this commitment. If they wanted to sync this calendar with another application, they could sync it with their Google or Outlook accounts, or export the calendar as an iCalendar file.

Must have:

Interface to allow user to add commitments to a schedule.

Users can edit commitments which are on the schedule

Ability to calculate the number of work hours / week that a piece of work will need.

Ability to place these work hours onto a calendar.

Ability for calendar constraints to be set. E.g. no work after 9pm.

Ability to apply items to a calendar within according to these constraints.

Should have:

Functionality for allowing the storing of a schedule.

Ability to adjust length of work chunk depending on the activity.

Ability to adjust placement of work hours based on priority.

Ability to add multiple commitments at once

Could have:

Account system / login system.

Easy to read user interface.

Ability to link assignments to modules/classes.

Mobile responsive webpage.

Outlook / google calendar connectivity.

Outlook / google calendar syncing.

Accessibility Settings e.g high contrast mode.

Screen for first use allowing a quick setup.

Ability for user to colour coordinate work chunks.

Ability to display a day, a week or a month of a schedule.

Would have:

Ability to import/export to iCalender Format.

Ability to select area on calendar to add/remove work chunks.

Ability to sync with non-Gregorian calendars for religious festivals / holidays.

B. Summary of project notebook.

- Meeting 24-09-2021 - Started on first draft of project proposal.
- Meeting 08-10-2021 - Got feedback on project proposal.
- Meeting 25-10-2021 - Finished literature search and started literature review.
- Meeting 22-11-2021 - Researched literature review.
- Meeting 26-11-2021 - Created MoSCoW Analysis, wireframes.
- Meeting 03-12-2021 - Chose to use Google Calendar API. Created a test application linked to Google Cal API.
- Meeting 13-12-2021 - Created website that allows users to add a custom calendar event to their Google Calendar using an HTML form and JavaScript, made first draft of progress and engagement report with introduction, gantt chart, and evaluation of progress.
- Meeting 15-12-2021 - Progressed on engagement report.
- Notes on progress engagement report 18-12-2021 - Got feedback from Richard about the PAE report.
- Meeting 09-22-2022 -Added system for linking events using secret fields, made functions for finding events linked to same task.
- Meeting 23-02-2022 - Started on basic scheduling algorithm.
- Meeting 09-03-2022 - Set up free/busy checks, started learning Node.JS in attempt to make backend. Came up with idea for simple events.
- Meeting 17-03-2022 - Fixed scheduling algorithm bugs. Allowed addition of multiple tasks.
- Meeting 11-04-2022 - Created rescheduling system.
- Meeting 24-04-2022 - Transition from writing code to portfolio.

- Explaining `checkDaysNoRemainder` - Creating text...pseudocode versions of functions to help me write the portfolio.
- Issues - interesting things encountered - Listing all design decisions made for use in portfolio.
- Creating events: changes: - Very rough page where not-taking never really started.
- Editing dissertation - Detailed portfolio plan.



Figure 11: The second upcoming week of the user's Google calendar before and after scheduling of new tasks. This week was originally left blank to make it visible how the scheduler spreads out workloads, and makes sure to schedule around breaks.


```
for(let i = 1; i < localEventList.length; i++){

    nextEvent = localEventList[i];
    nextEvent.start = new Date(nextEvent.start);
    nextEvent.end = new Date(nextEvent.end);

    //dealing with if the previous event completely envelops the next event:
    while(nextEvent.end < prevEvent.end){
        i++;
        nextEvent = localEventList[i];
    }

    // Checking we have found the closest event after the start of the event we are trying to fit in
    if(nextEvent.start > startTime)
    {
        //If true, Checking the subsequent event (nextEvent) does not clash with the event we are trying to fit in.
        if(nextEvent.start >= endTime )
        {
            //Checking the preceeding event does not clash with the event we are trying to fit in.
            if(prevEvent.end <= startTime)
            {
                //if no clash, return true
                console.log("\nFree! Pre: " + prevEvent.summary + ", " + prevEvent.start + " - " + prevEvent.end + "\n\n"
                + nextEvent.summary + ", " + nextEvent.start + " - " + nextEvent.end
                + "\n\n thisEvent.start > endTime: " + (nextEvent.start > endTime));
                return true;
            }
            else
            {
                console.log("Busy.");
                return false;
            }
        }
        else
        {
            console.log("Busy. Pre: " + prevEvent.summary + ", " + prevEvent.start + " - " + prevEvent.end + "\n Post: "
            + nextEvent.summary + ", " + nextEvent.start + " - " + nextEvent.end
            + "\n\n thisEvent.start > endTime: " + (nextEvent.start > endTime));
            return false;
        }
        break;
    }

    //set the prevEvent to be the value of the nextEvent before the cycle restarts.
    prevEvent = nextEvent;
}

console.log("Busy at END");

return false;
```

Figure 12: A code snippet from within checkFreeBusy for determining whether a time slot is free or busy.

Test:	Description	Data Type	Expected Result	Pass / Fail	Notes
1	Logging into the web page with a Google account	Typical	The user will be authenticated, and they will be able to use the websites tools.	Pass	
2	Scheduling a new task, total time to complete 6 hours, 2 hour work chunks, with a week time span.	Typical	Three new two hour long events will be added to the user's Google calendar, all within the week time boundary and within the programs default 9am-5pm work hours.	Pass	
3	Schedule 2 new tasks. First task: 1hr work chunks, 5 hrs total work. Second task: 3 hr work chunks, 6 hrs total work. Timespan for both: 1 week.	Typical	5 new 1hr events, 2 new 3 hr events, within the week time boundary and within the programs default 9-5 work hours.	Pass	
4	Repeat of test 3 but with the work hours set between 4pm and 11pm	Typical	Same result as test 2 except all events will be placed between 4pm and 11pm.	Pass	
5	Repeat of test 4 but with a break between 9pm and 10pm added for all days of week	Typical	Same results as test 4 but with no events scheduled between 9pm and 10pm on any day.	Pass	
6	Scheduling 3 tasks. 1 st task: total time to complete 18 hrs, min chunk size 3 hrs. 2 nd task: total time to complete 10hrs, min chunk size 2 hrs. 3 rd task total time to complete 5hrs min chunk size 1 hr. Time span for all tasks, 2 weeks. Work hours 9am-5pm, breaks between 1pm and 2pm and a break for the whole of Sunday.	Typical	6 new 3 hr events, 5 new 2hr events, 5 new 1hr events, all placed between 9am and 5pm, none overlapping with 1pm-2pm, within the 2 weeks.	Pass	See paragraphs for explanation on how this meets the requirements set out in the brief.
7	The events created in task 6 are on the user's calendar. A new Google Calendar event is created which blocks out an entire working day, deleting a 3hr and a 2 hr event. The user re-enters the breaks from task 6, and reschedules their tasks around this by ticking "Reschedule old tasks" button and pressing "Schedule tasks".	Typical	All old events created in test 6 are deleted, and 6 new 3 hr events, 5 new 2hr events, 10 new 1hr events, all placed between 9am and 5pm, none overlapping with 1pm-2pm, within the 2 weeks, are created to replace them. These new events are scheduled around the changes to the user's calendar.	Pass	See paragraphs for explanation of how this met the requirements in the brief.
8	Pressing "schedule task" with no tasks entered.	Erroneous	No changes are made to users google calendar.	Pass	
9	Pressing "create task" with no details entered.	Erroneous	Error message in console. No changes to Google calendar	Pass	
10	Pressing "create task" with no details entered, then pressing "Schedule tasks"	Erroneous	Error message in console. No changes to Google calendar	Pass	
11	Trying to create a task with a number of hours required less than the minimum work chunk size.	Erroneous	Error message to user explaining problem and saying to try entering task again with different details.	Pass	

Figure 13: The first part of the table of tests.

12	Creating a task with a number of hours required or a minimum work block size smaller than 1.	Erroneous	Error message to user explaining problem and saying to try entering task again with different details.	Pass	
13	Scheduling a new task, total time to complete 150 hours, 8 hour work chunks, with a 3 week time span.	Typical	18 new 8 hour long tasks and a 6 hour long task will be scheduled to the user's Google calendar.	Pass	
14	Scheduling a new task, total time to complete 151 hours, 9 hour work chunks, with a 3 week time span.	Erroneous	Error messages saying min chunk size is too larger, also error message saying number of hours required too large.	Pass	
15	Scheduling a new task, total time to complete 150 hours, 1 hour work chunks, with a 4 week time span.	Typical	150 new events scheduled between the 9am to 5pm default work hours over a 4 week window.	Fail	120 new events created, interspersed with lots of error 403 rateLimitExceeded errors. After 74 successful calendar entries errors started appearing. Google does not allow developers more than 100 requests per 100 seconds.
16	Schedule new task with work hrs between 0 and 24, 24 hrs total needed, 1hr minimum work chunk.	Typical	24 new 1hr work events scheduled over the entire day.	Pass	
17	User closes web page before all tasks have been scheduled to Google calendar.	Erroneous	Prediction is that not all tasks will be scheduled, but this should be fixed.	Fail.	Not all tasks are scheduled. This is because website is currently not server based, so all calculations occur while the user's webpage is open.

Figure 14: The second part of the table of tests.

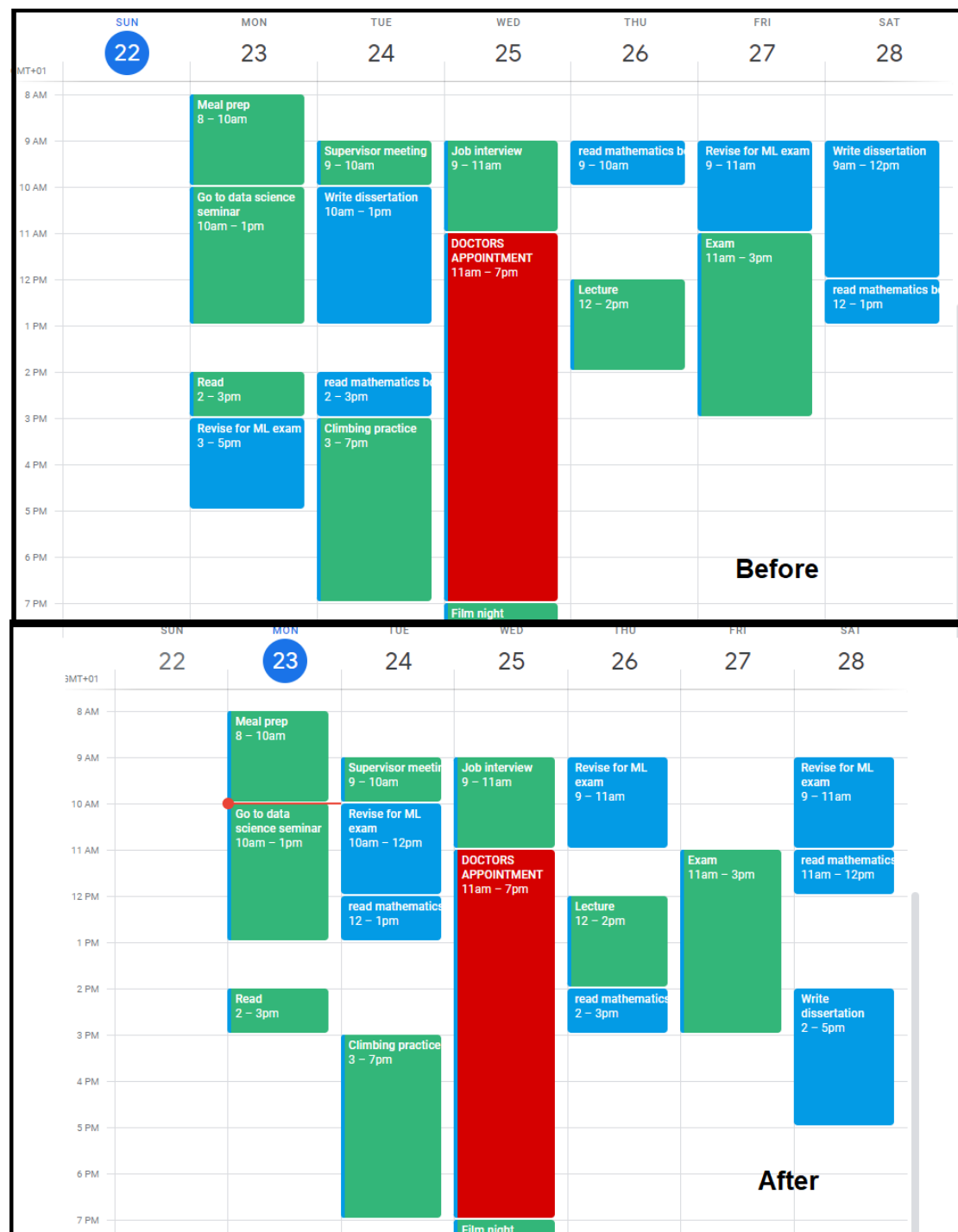


Figure 15: The first upcoming week of the user's Google calendar before and after rescheduling of tasks after addition of the drs appointment in red. The user's pre existing events are in green, the doctors appointment in red, and the tasks generated by the scheduler are in blue.



Figure 16: The second upcoming week of the user's Google calendar before and after scheduling of new tasks.