2025

Simplifying Employee Scheduling Through Automation

ROSTERREADY

KACPER AGATOWSKI

Author: Kacper Agatowski

Student Number: N00212272

Supervisor: John Montayne

Second Reader: Mohammed Cherbatji

Date: 30/04/2025

Frontend Code: https://github.com/KacperA02/rosterready1

Backend Code: https://github.com/KacperA02/RosterReadyBackend

Year 4 2024/25

DL836 BSc (Hons) in Creative Computing

Abstract

The aim of this project was to construct an application which addressed the challenges around complex scheduling, especially in sectors with high staff turnovers and many part time workers. Scheduling employees is essential in nearly every industry, which can become frustrating or time consuming for the employer. An application which automates their schedules can solve issues around complex scheduling and save time for employers.

The system was developed using a full-stack architecture, incorporating a SQL database, a FastAPI backend, and React.js frontend. The core functionality resided in the backend which integrated a constraint satisfaction problem (CSP) solver, which inherited search algorithm techniques and accepted constraints such as availability of employees. Using the business data provided it would find a solution by assigning employees to shifts without violating any rules.

The CSP became challenging when dealing with a lot of variables such as shifts and employees. Optimisation techniques were introduced to overcome these challenges, reducing computational time and offering an optimal solution to the employer. Additionally, if the solution did not satisfy the employer's needs, the employer could regenerate the solution with certain assignments locked to match their ideal schedule.

The final application resulted in an efficient, scalable and reliable scheduling system capable of generating optimal scheduling solutions within seconds, offering a solution to the process complex scheduling.

Acknowledgements

Firstly, I would like to give a big thank you to my supervisor John Montayne for guiding me through this project. Mr Montayne kept me motivated and positive during the weekly meetings and was very quick at responding to urgent questions I had with the project. Mr Montayne also pushed me outside my comfort zone, to try implement features that I thought were out of my scope. John Montayne believed I can learn more, which became an asset in gaining more experience in software development.

Secondly, I would like to thank all the lecturers which taught me all the valuable code needed for creating this project. Two lectures specifically, Tim McNicholas and John Dempsey for introducing me to python and the fundamentals of the language. I would also like to give a big thank you to Mohammed Cherbatji for the advanced JavaScript tutorials and lectures, which he provided throughout the course. Mr. Cherbatji's teachings led me to become proficient in JavaScript and frameworks such as React.

The incorporation of material without formal and proper acknowledgement (even with no deliberate intent to cheat) can constitute plagiarism.

If you have received significant help with a solution from one or more colleagues, you should document this in your submitted work and if you have any doubt as to what level of discussion/collaboration is acceptable, you should consult your lecturer or the Course Director.

WARNING: Take care when discarding program listings lest they be copied by someone else, which may well bring you under suspicion. Do not to leave copies of your own files on a hard disk where they can be accessed by other. Be aware that removable media, used to transfer work, may also be removed and/or copied by others if left unattended.

Plagiarism is considered to be an act of fraudulence and an offence against Institute discipline.

Alleged plagiarism will be investigated and dealt with appropriately by the Institute. Please refer to the Institute Handbook for further details of penalties.

The following is an extract from the B.Sc. in Creative Computing (Hons) course handbook. Please read carefully and sign the declaration below

Collusion may be defined as more than one person working on an individual assessment. This would include jointly developed solutions as well as one individual giving a solution to another who then makes some changes and hands it up as their own work.

DECLARATION:

I am aware of the Institute's policy on plagiarism and certify that this thesis is my own work.

Student : Kacper Agatowski

Signed [illegible]

Failure to complete and submit this form may lead to an investigation into your work.

Table of Contents

1 Introduction

Employers often struggle to create a conflict-free employee schedule for their business. The project aims to simplify scheduling by generating schedules using an optimised Constraint Satisfaction Problem (CSP) solver based on employer and employee needs. The project consists of creating a full stack application to handle this scheduling process, which utilises MySQL, FastAPI, and React as its technical stack.

The project requires thorough management due to its complexity, therefore, scrum methodology is applied to divide the project phases into iterative sprints. The research phase involves the investigation of search algorithms and CSPs to support the proposed scheduling solution. The requirements section includes gathering data through surveys, interviews and competitor analysis to model the functional and non-functional requirements. The design phase contains the program design and user interface design, providing a structured base for further development. The implementation phase translates the design and requirements into working code, which is organised according to the technical stack. Finally, the testing phase evaluates the functionality and user experience through a series of tests to ensure reliability and usability.

2 Research

2.1 Introduction to Research

Scheduling in various industries is a complex and essential task. Efficiently assigning variables to values without breaking constraints is crucial for finding successful scheduling. However, challenges may arise if the problem itself is complex.

The following sections explore search algorithms and constraint satisfaction problems solving (CSPs) to give an understanding of computational techniques and how they can be used to tackle certain scheduling aspects. By combining search algorithms within the CSP framework, allows to efficiently navigate through the search space and find a solution.

The approach of understanding search algorithms and CSPs becomes very powerful when tackling scheduling challenges. Allowing for different approaches, depending on the scheduling problem that needs to be tackled.

2.2 Search Algorithms

2.2.1 Introduction to Search Algorithms

Search Algorithms (SA) are a type of algorithm used within Artificial Intelligence (AI) to find the best or most optimal solution by exploring the state space provided from the problem. A search problem consists of a state space, successor function, start state, goal state and a solution (Webcast Departmental, 2018, August 28). The algorithm begins its search at the start state, it has a set of possible states known has the state space. A successor function is created to generate the next state from the current state (Webcast Departmental, 2018, August 28), using this function the goal state can be found by checking each current state with the goal state. The solution is the sequence of actions it took from the start state to the goal state; this is also known as a plan (GeeksforGeeks, n.d).

For example, the puzzle ‘Sudoku’ has the search problem of filling the entire 9x9 grid with numbers such that each row, column, and box contains only one of the nine numbers. The state space would be a set of possible configurations of the grid. The start state includes the configurations of the cells that are already filled within the grid. The successor function generates new states by assigning a number to an empty cell which doesn't break any of the search problem rules. The goal state is to have a fully filled grid which satisfies the search problem. The solution or plan is the actions or steps it costs to complete this puzzle.

2.2.2 Types Of Search Algorithms

SA's within AI will be categorised into two main types: Uninformed and Informed. Each of these categories play a significant role in solving the search problem addressed. Choosing a type of category is important to connect it with the type of problem trying to be solved. In the following section, the two categories will be discussed in detail, including how they work and examples of the types.

Common Components of Uninformed and Informed SA

To understand uninformed and informed search, the common components must be first addressed. Each algorithm will contain a *problem graph*, which contains the start node (S) and a goal node (G). A *node* represents a single state. A *solution plan* is the sequence of nodes from S to G. (GeeksforGeeks, n.d.). A *strategy,* is how the problem graph will be approached to get to G. A *fringe* is a set of leaf nodes waiting to be expanded. (Webcast Departmental, 2018, August 28) A *search tree* is then created, which shows the generated path the nodes took. An *optimal solution* is the solution with the lowest cost among other solutions within the problem graph. The *path cost* would generate an integer for each state; however, some algorithms would have different ways of calculating this cost. (Javatpoint. n.d.)

2.2.2.1 Uninformed Search

Uninformed Search explores the state space without any guidance towards the goal state. This type of search is also known as a blind search as it explores all possible states (Pathak, Patel, & Rami, 2018). It is inefficient in search problems which have a large state space. It falls under several different ways of searching techniques, Depth First Search (DFS), Breadth First Search (BFS), and Uniform Cost Search (UCS).

Depth First Search (DFS)

This algorithm begins at the root node (S), DFS strategy is to explore the depth of the search tree first by following a single branch. The fringe within DFS is that any node that is expanded last is called upon first, meaning that DFS prioritises deep exploration. (Webcast Departmental, 2018, August 28). In relation to Figure 1, The first solution plan is the one on the left (A-B-C-G), at each node it would check if the G node was found. However, if the G node was not found on the left it would use the fringe to then backtrack to the next node in the stack and continue to do so until the G node is reached.

In relation to Figure 1, A human can see that the fastest route would be (S-D-G). DFS would not be the best option when the search space is deep and has many nodes as it would be very time consuming. It would be able to complete or find the solution, but it might not be the most optimal solution. (GeeksforGeeks, n.d.).

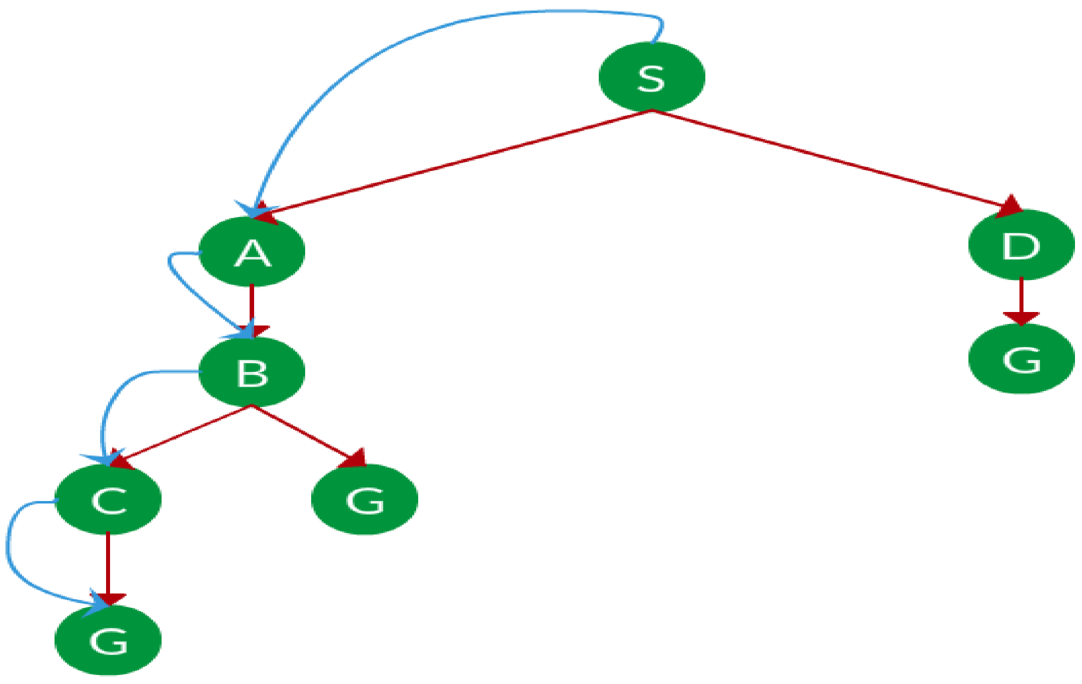


Figure 1, Diagram of Depth First Search (Source: GeeksforGeeks, (n.d))

Breadth First Search (BFS)

BFS begins at the root node (S), BFS strategy takes the shallowest layer first within the search tree. The fringe within BFS is that it holds all the nodes that have been discovered but not explored yet, it would queue the fringe in terms of first in, first out (FIFO). (Pathak, Patel, & Rami, 2018). This would mean the search graph checks each layer before proceeding to deeper layers. In relation to Figure 2, the most optimal solution would be (S-B-G). However, as ‘A’ is the shallowest it put this node in the queue first and places ‘B’ in the queue after. Once ‘A’ is explored, ‘D’ then ‘C’ is placed in the queue, which then the strategy backtracks to ‘B’, due to the FIFO fringe. BFS would have explored all these nodes (S-A-B-D-C-G) before finding the G node.

In relation to Figure 2, BFS could find the optimal solution but would take more time as it would have to go layer by layer. However, if the goal node was ‘K’ and optimality was not the shortest path but the cost of path then BFS would not be seen as optimal. (Webcast Departmental, 2018, August 28).

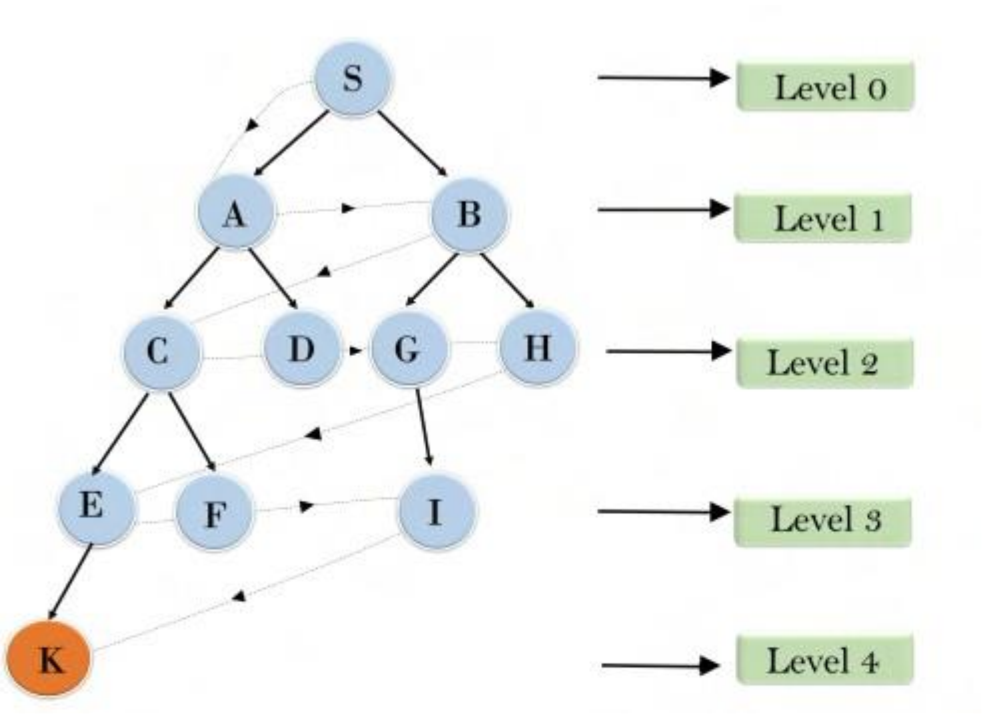


Figure 2, Diagram of Breadth First Search (Source: Javatpoint (n.d))

Uniform Cost Search (UCS)

UCS inherits the same strategy as BFS, however in USC instead of finding the shallowest path, it uses a least cost first strategy (LCF). (Webcast Departmental, 2018, August 28). The fringe within UCS would be a queue of discovered nodes while prioritising the lowest costing nodes. The main goal of the UCS is to find a path which has the least cumulative sum of the cost explored. (GeeksforGeeks, n.d.). In relation to Figure 3, the initial node is ‘A’, and the goal node is ‘G’. ‘A’ discovered ‘C’ and ‘B’, but the strategy and fringe prioritised the ‘B’ node as it cost less than ‘C’.

In relation to Figure 3, The UCS does find the most optimal solution however, a more complex graph which holds negative cost would explore in every direction as it still has no information about the goal state. However, UCS is still better than BFS and DFS in terms of its optimality and completeness under time consumption.

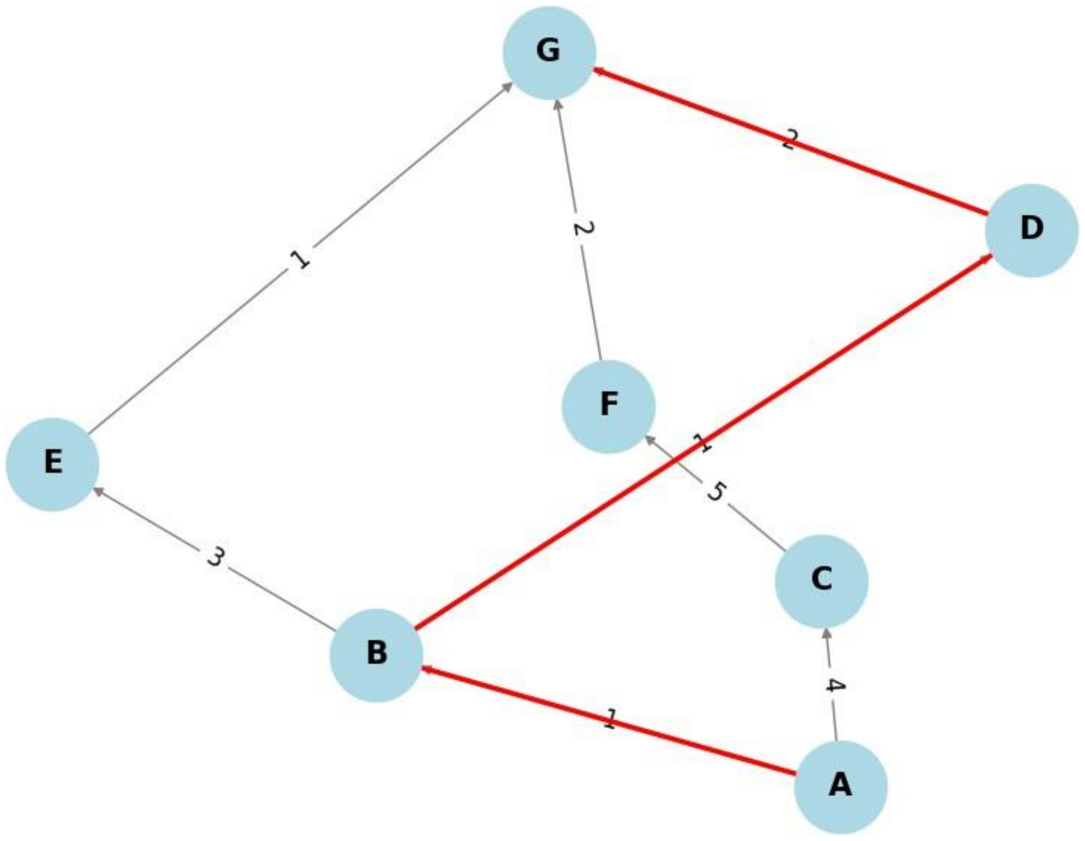


Figure 3, Diagram of Uniform Cost Search (Source: GeeksforGeeks (n.d))

2.2.2.2 Informed Search

An informed search algorithm has information about the goal state, which helps it be more efficient when searching. These algorithms use a heuristic, which is a function to estimate how close the current state is from the goal state. (Webcast Departmental, 2018, August 30). A different heuristic would be used within different informed search algorithm (SA). The solutions that come back are more optimal than the solutions from uninformed SA. A* search and Greedy search are two types of informed SA which will be discussed.

Greedy Search

In greedy search, the algorithm expands the closest node to the goal node based on the heuristic function *h(n) (*GeeksforGeeks, n.d.). The heuristic in greedy search is measured by how close the distance is from the goal and priorities nodes with the smallest heuristic value, which is determined by the distance from the goal, also known as Manhattan distance (Webcast Departmental, 2018, August 30)

In relation to Figure 4, the initial node is 'S', and the goal node is 'G'. From the start the algorithm has a choice to make between h(A)= 9 and h(D)=5. The algorithm would choose 'D' as it cost function is lower than both its current and 'A'. The solution plan would be (S-D-B-E-G) and the path cost would be 19, however the algorithm is solely focusing on the next h(n) and ignores the total cost of the path. Relating to Figure 4, the algorithm might have found the goal, but it wasn't the most cost-effective path in total, showing that prioritises speed for optimality.

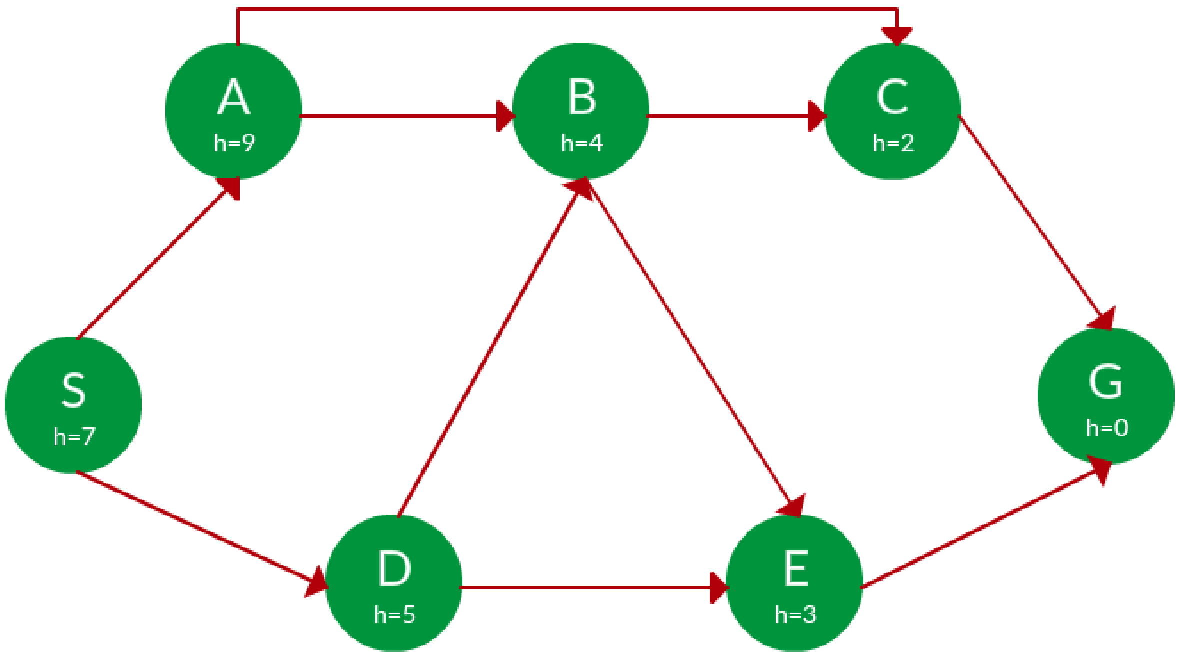
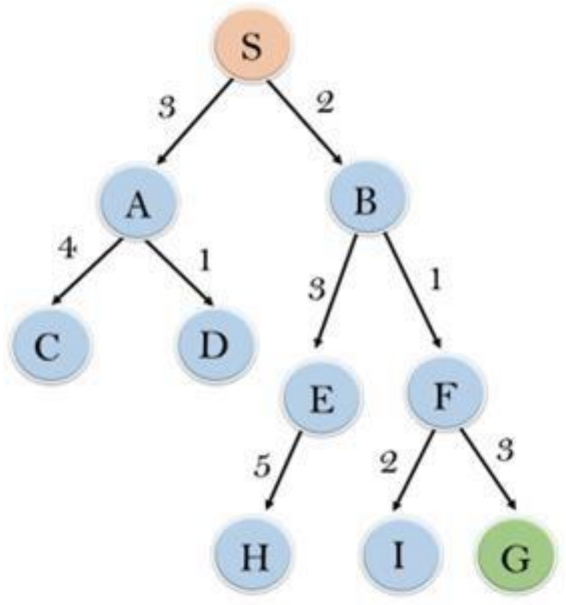


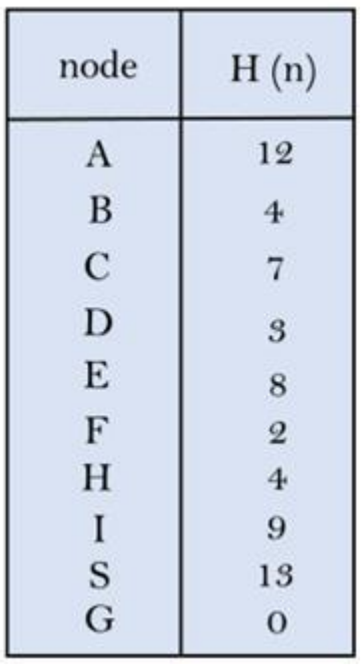
Figure 4, Diagram of Greedy Search (Source: GeeksforGeeks (n.d))

A* Search

A* search combines both UCS and Greedy algorithms, orders by the sum of forward cost and backward cost. In a formula it would look like this: *f(n) = g(n) + h(n)*. (Webcast Departmental, 2018, August 30). This algorithm would combine both the path cost from UCS *g(n)* and the goal distance from greedy *h(n)*. The A* search guarantees optimality if *h(n)* is admissible, which is done if the heuristic never overestimates the true cost of reaching the G node from any node. (Webcast Departmental, 2018, August 30). This formula looks like this *h(n) ≤ h*(n)*, this would mean the heuristic is optimistic as it would never ignore a better potential path. However, if the search problem is large and the heuristic is admissible, it could lead to memory issues as all nodes are explored.

In relation to Figure 5, the initial node is 'S', and the goal node is 'G'. Adding the heuristic as the strategy, it leaves the solution plan to be (S-B-F-G). At the first node it checks *g(B=4) + h(B=2)* output is larger than *g(A=3) + h(A=12).* The algorithm would prioritise B and proceed to then check if the values are less than or equal to the current value, which is the admissibility in the heuristic, *g(B=4) + h(B=2) ≤ g(E=4) + h(E=2) | g(F=4) + h(F=2)*.





node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0

Figure 5, Diagram of A Search (Source: Goseeko (n.d))*

Efficiency

The efficiency of an algorithm solely depends on the search problem. In the case of scheduling, the challenge of exploring a wide range of state spaces and dealing with rules or constraints assigned by the schedule such as availability, shifts and other scheduling related rules. As there are rules or constraints the scheduling search problem becomes more of an informed search rather than an uninformed search.

Creating heuristics which match the scheduling problem would increase efficiency, while also using techniques or fringes from uninformed search can benefit the accuracy of the scheduling search problem. This could be managed with a structured framework like Constraint Satisfaction Problem Solving (CSPs) to define these constraints and implement both informed and uninformed search techniques.

2.3 Constraint Satisfaction Problem Solving (CSPs)

2.3.1 Introduction to CSPs

Constraint satisfaction problems solving (CSPs) are unlike search algorithms (SA) but inherit some of their techniques. In SA the goal state can be any function within the state, however in CSPs there is structure which defines a set of variables, domains and constraints. (Webcast Departmental, 2018, September 4). The goal state for CSPs isn't a single state but it's completing the problem by assigning the domains to variables without breaking the constraints provided, which is also known as a solution. (Barták, Salido, & Rossi, 2010).

Variables and Domains

Variables are the entities within the problem that need to be assigned values. (Webcast Departmental, 2018, September 4). In terms of scheduling, they can be, time slots, tasks or employees. Each variable would be assigned a singular domain. There are two types of variables, discrete and continuous. (Webcast Departmental, 2018, September 4). Discrete variables can be assigned to finite or infinite domains. Continuous variables can take any domain within a continuous range.

Domains are a set of possible values for a variable. (Barták, Salido, & Rossi, 2010). In the case of scheduling, they can be shift times. Domains can be set as any type of value, these include finite values such as days of the week, and infinite values such as all positive integers, to store how many minutes the employee worked.

Constraints

Constraints are a set of rules given to the variables that cannot be broken. If a variable is assigned a domain and doesn't break any constraint it is known as partial instantiation. (Barták, Salido, & Rossi, 2010) If all the variables are assigned domains that don't break any of the constraints indicates that a solution was found, and instantiation is completed. As discussed in Webcast Departmental, 2018, September 4, there are a variety of constraints which are broken down into three categories, unary, binary and higher-order constraints. Unary constraints involve a single variable. In terms of scheduling this could be, 'employee1' cannot work 'Tuesday middle shift'. Indicating that the variable 'employee1' cannot be assigned the domain of 'Tuesday middle shift'. Binary constraints involve two variables, this could be that two shifts (variables) cannot overlap each other. Higher order constraints involve more than two variables, this could involve three different employees (variables) cannot work together on the same shift (domain).

As constraints are rules created not to be broken. Creating constraints which can be broken are called soft constraints. (Webcast Departmental, 2018, September 4). In the case of scheduling, an employee would prefer to have the weekend off. This constraint could be broken if no solution was found with this constraint fixed. Limiting the number of constraints is important as if there are too many constraints that cannot be broken also known as hard constraints, there might not be a solution for the problem.

2.3.2 Solving

Once the variables, domains and constraints are set in the CSP, selecting the right solving strategy depends on the problem that is being solved. Solving CSPs is done by assigning values to variables (instantiating them) which doesn't break constraints. Integrating various methods and techniques would improve and reduce the search space within the CSP. The steps involved in solving a CSP will be discussed to understand how to ensure correct solutions.

2.3.2.1 Backtracking Search

Search is the main algorithmic technique for solving CSPs (Rossi, 2008). Search algorithm for a CSP can be either complete or incomplete. Complete search algorithm comes with a guaranteed solution and possible optimal solutions. An incomplete search algorithm would indicate the problem does not have a solution. An algorithm which is seen as complete is backtracking search.

Backtracking search method also known as chronological backtracking is the backbone of CSPs (Barták, Salido, & Rossi, 2010). It explores the search space and assigns a value from the domain to a variable at each state space then checks if a constraint was broken or not. If a constraint was not broken the search would continue just like Depth-First search and continue assigning values to variables. If a constraint was broken the algorithm would backtrack to the previous state space and assign a new value to the variable.

2.3.2.2 Constraint Propagation

Constraint propagation is a key technique within CSPs to reduce the search space by enforcing consistency between variables in a constraint network. As discussed by Bessiere (2006), the goal of constraint propagation is to remove any pairs (domains and variables) to reduce the number of possibilities which are invalid as they would break a constraint in future states. Techniques such as forward checking, path consistency, and arc consistency are used depending on the structure and constraint category with the search problem.

Forward Checking

Forward checking is one of the most common techniques used within constraint propagation. (Barták, Salido, & Rossi, 2010). Forward checking keeps track of domains, once an assignment is made this technique filters the possible domains for future assignments. It would filter out the domain that was previously assigned from other neighbours (Webcast Departmental, 2018, September 4). A neighbour variable refers to a variable which is directly connected to another variable through a constraint. If the filtered domain list becomes empty, it would indicate that the assignment was not valid, and the backtracking method would begin.

Arc Consistency

Arc consistency would generally be enforced during forward checking. Instead of the just checking the neighbours of the variable like forward checking, it checks all variables at a global level. An arc is consistent if there is still at least one domain present in all variables (Barták, Salido, & Rossi, 2010). It does this by checking the pair of variables bidirectionally, checking that X (head) and Y (tail) for every value in the domain of X, there is still value in the Y that doesn't break a constraint (Webcast Departmental, 2018, September 4). This technique can be used during the search process just like forward checking or be used as a preprocessing technique. This technique works the same as A* search, as it checks all nodes before the search and reduces the search space.

Path Consistency

Path consistency is like arc consistency but works with higher order constraints rather than binary constraints. It checks for a set of three variables, the values in their domains don't break any constraints between them all. (Barták, Salido, & Rossi, 2010) For example, if there are three variables X, Y and Z with constraints between each pair. Path consistency checks if at least one value within all three domains still exists. It would remove any values that don't satisfy the constraints on the variables.

2.3.2.3 Heuristics Approaches

Heuristics are methods used to order variables or values to minimize backtracking while making more informed choices about which variables to assign and which values to consider. Heuristics would prioritize certain decisions that are likely to lead to a faster solution (Webcast Departmental, 2018, September 4). They would be applied after a constraint propagation technique is applied. Heuristics are applied to two different areas, variable selection and value selection.

Variable Selection

Variable selection determines the order in which variables are chosen for the next assignment. Minimum remaining values (MRV) chooses the variable with the fewest legal values left in that variable's domain (Webcast Departmental, 2018, September 4). Choosing the most constrained variables first, reduces the depth in the search tree. This approach would reduce the risk of hitting a dead-end, variable with no values in the domain (Van Beek, 2006). Hitting a dead end would lead to backtracking but this approach deals with the harder variables to assign first.

Value Selection

Value selection determines the order in which values to assign to a chosen variable (Webcast Departmental, 2018, September 4). Least constraining value (LCV) chooses the values that have the least amount of effect on the remaining variables. By choosing LCV, it leaves maximum flexibility for future assignments and reduces the risk of hitting a dead end (Van Beek, 2006).

2.3.2.4 Local Search Techniques

Local Search Techniques involve a different approach to backtracking. Local Search would complete the entire state space without looking at any constraints (Webcast Departmental, 2018, September 6). Once the state space is completed, some of the states may not be breaking any constraints and they would be left alone. The variables which are breaking would be assigned new values to not break any constraints, essentially working backwards. However, this approach could run endlessly if no solution is available. There are many different local search algorithms with different heuristic approaches which are available but would depend on the problem itself.

Hill Climbing

Hill climbing is a type of local search algorithm, which improves your solution by choosing the best neighbour value (Webcast Departmental, 2018, September 6). Using an evaluation function of keeping track of the number of constraints and solving and decreasing till a solution is found. A common problem in local search algorithms is that a solution could be found but it might not be the most optimal solution. Using a technique such as random restarts could help find a more optimal solution. Random restart strategies create multiple search solutions starting from a different value or variable (Van Beek, 2006). This would find more solutions and compare solutions to find the most optimal one.

Iterative Search

Iterative search in relation to local search would allow to make small adjustments to a solution to change the solution to possibly find a more optimal solution. Users being able to lock a specific assignment which then changes the neighbours in relation to the change, so no constraints are broken.

2.4 Solving Scheduling challenges with SA and CSP's

Scheduling employees to shifts can be challenging, especially if the problem itself is complex or involves an imbalance between variables and constraints. Understanding the scheduling problems and the computational difficulties posed by their combinatorial structure allows to make a hypothesis about which search algorithm would be most suitable for tackling these challenges within a CSP framework.

2.4.1 Understanding Scheduling Problems

Scheduling problems come in many different scopes or sizes and as they increase so does the complexity of the problem. Scheduling problems would consist of a set or multiple sets of variables, domains and constraints. For example, a set of variables would consist of employees, and possibly departments, roles, tasks and any other the depending on the industry. Domains would hold the shift times for each day. A single variable or multiple variables from one set would be assigned to a domain or multiple domains depending on the problem itself. A constraint would consist of hard and soft constraints. Hard constraints would be legal labour laws, employee availability, the number of variables assigned to each domain, and due dates (Fox, 1990). A soft constraint would consist of shift preferences. As there are many different types of scheduling problems, understand the vast majority would allow to create a flexible algorithm.

Creating a shift schedule would be done on a static method meaning the shifts are created in advance and not in real time (Rossi, 2008). However, with a static method, scheduling problems seem to arise when a variable, for example employee cannot work the following day as they are sick. This means a dynamic method would need to be used in real time to update a part of the current solution and find a new solution to satisfy the new problem, for example a shift repair method (Renke, 2021). Using only dynamic methods to update any scheduling problem constantly would use a lot of computational power and increase time complexity.

A specific method could be assigned to a problem depending on the number of variables, domains and constraints. For example, within a simple case of coffee shop problem, a dynamic method can be applied as a coffee shop wouldn't have as many variables, domains and constraints compared to the nurses scheduling problem (Cheng, 1997). Understanding the complexity of each problem will be important when assigning a method.

2.4.2 Combinatorial nature of scheduling challenges

In combinatorial scheduling problems would generally have many variables, domains, and constraints, making the search space large and creating a complex solution space. These problems are nondeterministic polynomial time hard (NP-Hard) problems. In relation to Larksuite (n.d), within the computational complexity theory, NP problems are classified depending on the how quick the problem can be solved. This means that scheduling problems are classified as hard due to the uncertainty in polynomial time it takes to find a solution.

As each industry is different within the real world, problems can be simpler than others. Industries might have only one set of variables such as employees. This becomes a single resource problem, as the algorithm only needs to assign one set of variables over multiple domains (Fox, 1990). For example, X = Employees and Y = time slots. If $X = Y$ that would mean, there are exactly $Y!$ (factorial of time slots) possible solutions. If $X > Y$, it would mean that there is extra flexibility and X*Y solutions. This would then require optimization to find the most optimal solution.

In larger or more complex industries they may require multiple resources (Fox, 1990). For example, there could be three sets of variables, employees, equipment, and locations. This would mean that a single domain is assigned to three separate variables which could have constraints between each set of variables. This makes it a highly computational problem as the search space would be larger and time complexity would be higher, which may result only with one solution or even none.

Cheng’s research on the nurses scheduling problem (1997) gave valuable insight to multiple case studies which there were 27 nurses and 13 different shift types. The theoretical complexity of this search space would be 27 to the power of 13, meaning that the search space for a solution is quite expediential. Problems like these would require the right optimization techniques to come up with any solution within the CSP. Within Cheng’s (1997) first case study, they explore preferences from nurses, 17 out of 27 nurses request a specific shift. Some of the requests clashed as some shifts would be more popular than others, which poses difficulty for the model of choosing which employee gets the popular shift.

Optimization techniques within scheduling would need to be implemented efficiently. In cases like the nurses scheduling problem, creating a table to keep track of which employee should get the next preferred shift would keep fairness within the problem, and implementing dynamic scheduling methods to patch any changes in real time.

2.4.3 The correct search algorithm for CSP in scheduling

Understanding how each search algorithm works is essential when implementing these algorithms into a CSP around scheduling. The complexity of the scheduling problem would influence on the decision of implementing a search algorithm within the CSP. Creating a flexible algorithm to check the complexity of the problem would allow to implement the right algorithm for each specific problem.

2.4.3.1 Smaller scheduling problems

As mentioned, there are a many different search algorithms to choose from within the CSP. Backtracking algorithms could be used within smaller problems to give optimal solutions using various techniques. Constraint propagation techniques such as forward checking to reduce the unnecessary exploration in the search space. Arc and path consistency would help reduce the number of conflicts encountered and improve overall efficiency (Cheng and Smith, 1997).

2.4.3.2 Larger scheduling problems

For larger and more complex scheduling problems, such as involving multiple variables. Local search techniques such as hill climbing would fill the problems search space randomly and change any hard constraint violation relations. This would find any solution, which may not be the most optimal but still finds a solution which satisfies all hard constraints (Bartak, 2010). By choosing computational time over quality would satisfy the problem but may not fully satisfy the company. This is where iterative search would become a key technique to satisfy the company.

2.4.3.3 Implementing iterative search

Implementing iterative search to both smaller and larger scheduling problems would be beneficial for the user. By locking certain assignments from the solution to create a more beneficial solution for the business. As a solution may be seen as an optimal solution, however it might not meet the right needs for the business.

2.5 Conclusion on research

The research section was vital in understanding the how to tackle scheduling problems, which is crucial for the development of an automated scheduling application. Addressing the challenges through the combined use of Search Algorithms (SA) and Constraint Satisfaction Problems (CSPs) offers a versatile and scalable approach to both simple and complex scheduling problems. By defining the structure of the problem with variables, domains and constraints, allows to break down the problem itself and maximise the efficiency and effectiveness of the algorithm.

Using search algorithmic techniques, such as backtracking, constraint propagation, and iterative search within the CSP provides an optimal solution by exploring the search space systematically and eliminating invalid options early on. The choice of techniques solely depends on the size of the scheduling problem to provide an optimal solution. Smaller problems could use complete algorithms such as backtracking then be enhanced by constraining propagation techniques like iterative search. Larger problems which include a deeper search space can become computationally expensive, approaching these with local search techniques like hill climbing or iterative search can offer a more optimal solution in theory.

The integration between both SA's and CSP's enable more structured approach to the problem being solved, offering efficient scheduling solutions across a wide range of industries and use cases. By leveraging these techniques dependant on the size of the problem offers scalability and flexibility in the development. These strategies allow to understand, how the solutions are being found and what requirements are needed to develop this automated scheduling application.

3 Requirements

3.1 Introduction

The process of requirement gathering, modelling, and feasibility analysis is fundamental in the development of a software system, ensuring that the final product aligns with the needs and expectations of the target audience. This chapter details the requirements for the automated scheduling system, outlining how they were identified, refined, and structured.

To establish a solid understanding of how existing scheduling applications function, an in-depth analysis of two similar applications, ConnectTeam and HomeBase, was conducted. These applications were evaluated based on their functional and non-functional aspects, providing insight into industry standards and usability considerations.

Primary data collection was carried out through interviews and a user survey to gain insights into the challenges business owners face in scheduling their workforce. Interviews were conducted with two local business owners to understand their scheduling workflows and the difficulties they encounter. Additionally, a survey was distributed to business owners across Wicklow and Dublin Counties, covering topics such as current scheduling techniques, workforce management, and openness to adopting new scheduling software.

The collected data was analysed to develop personas, user requirements, technical requirements, functional and non-functional requirements. These requirements formed the basis for the development of a Use Case diagram, which visually represents user interactions with the system. Additionally, two use cases were developed, reflecting real-world scenarios based on the personas.

The chapter concludes with a feasibility analysis, which assesses the project's technical, operational, economic, and managerial viability. This evaluation ensures that the proposed system is achievable, identifies potential challenges, and outlines mitigation strategies.

3.2 Requirements gathering

3.2.1 Similar applications

This section will cover two popular applications for employee management: *ConnectTeam* and *HomeBase*. Both are designed to simplify scheduling, communication, and task management. Each application offers unique strengths, with a wide range of functional features and distinct non-functional advantages.

3.2.1.1 ConnectTeam

ConnectTeam is a comprehensive employee management platform built to streamline productivity and communication in the workplace. With an intuitive, user-friendly interface, *ConnectTeam* makes managing employees easy through features like drag-and-drop scheduling, real-time chat, and GPS-based attendance tracking. While the dashboard is feature-rich, its information density may take some time to navigate. Despite this, *ConnectTeam* remains a popular choice for managers seeking an all-in-one tool to efficiently manage and support their workplace.

In the following sections, the app's functional and non-functional aspects will be examined to get a better understand its features and how they contribute to user experience.

<u>Main Functional Aspects</u>

ConnectTeam provides a wide range of features that support the essential functions of employee management, including:

Employee Communication

Managers can set up group chats and communicate with their teams in real-time, allowing for rapid announcements and immediate feedback. This functionality enhances coordination across various departments, ensuring all team members are well-informed about new information. In addition, managers can also send global messages, ensuring that urgent information or changes reach the correct people instantly, we can see this feature in figure 6 below.

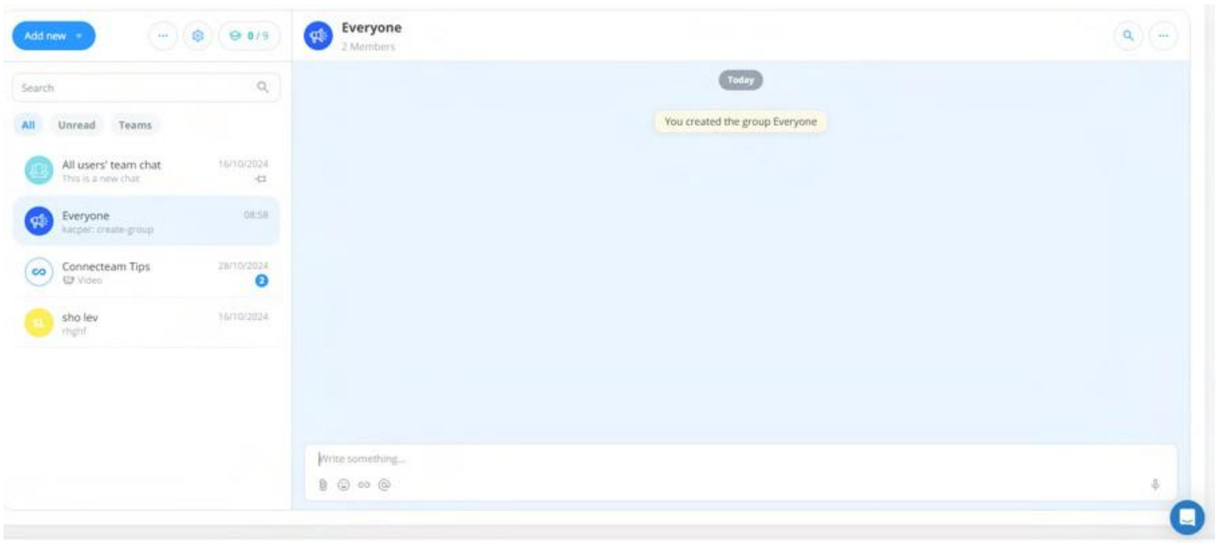
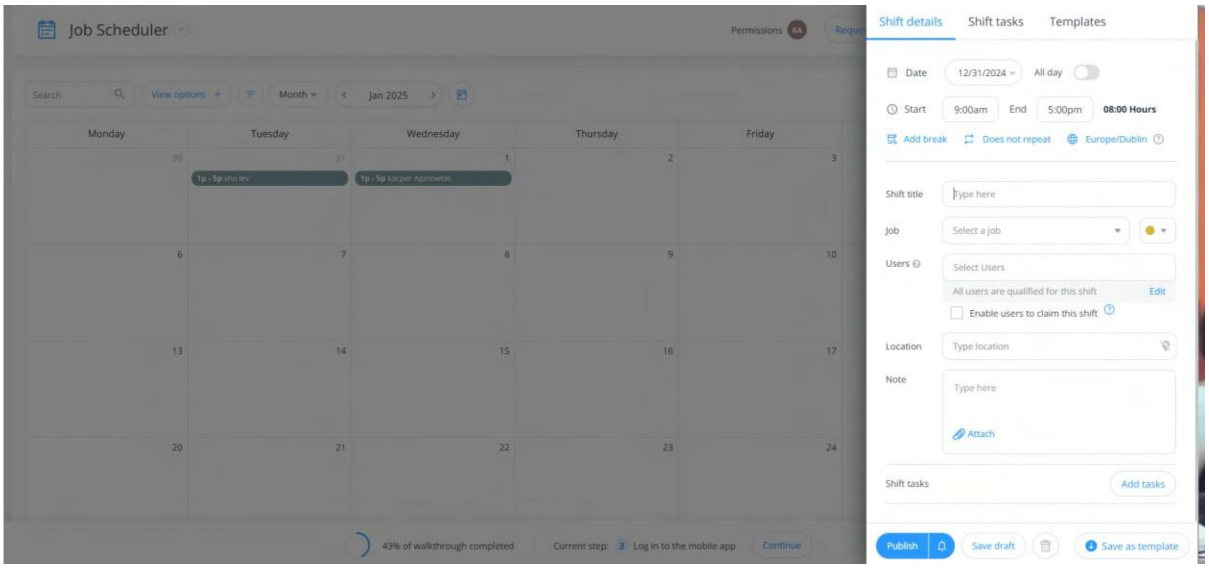


Figure 6, Employee Communication (ConnectTeam (n.d))

Task Management and Scheduling

Employers can create and assign tasks based on specific roles, ensuring that the right employees are responsible for the right duties. Once tasks and schedules are set, employees are notified through the app or email, keeping them informed instantly. The platform also allows employees to request schedule changes, which are easily managed and updated by the employer. Automated notifications are sent to remind employees of upcoming shifts, changes, or new tasks, ensuring employees are notified from the primary source. This functionality ensures a smooth workflow by delivering timely reminders and promoting clear communication between managers and employees. We can see these how these schedules are implemented in Figure 7 and how tasks are implemented in Figure 8.

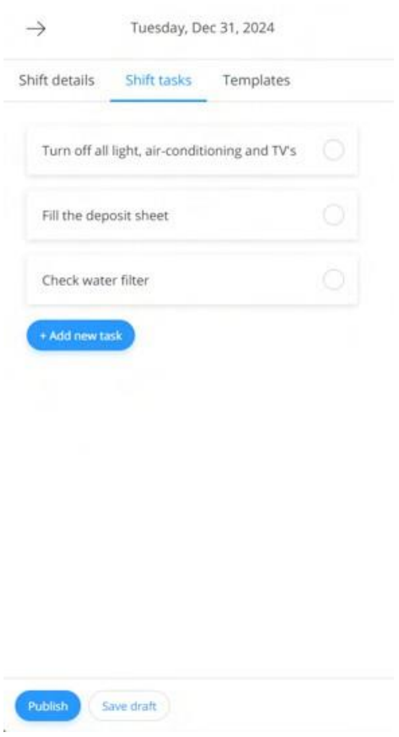


Date 12/31/2024 All day

Start 9:00am End 5:00pm 08:00 Hours

Does not repeat

Figure 7, Scheduling (ConnectTeam (n.d))



Turn off all light, air-conditioning and TV's

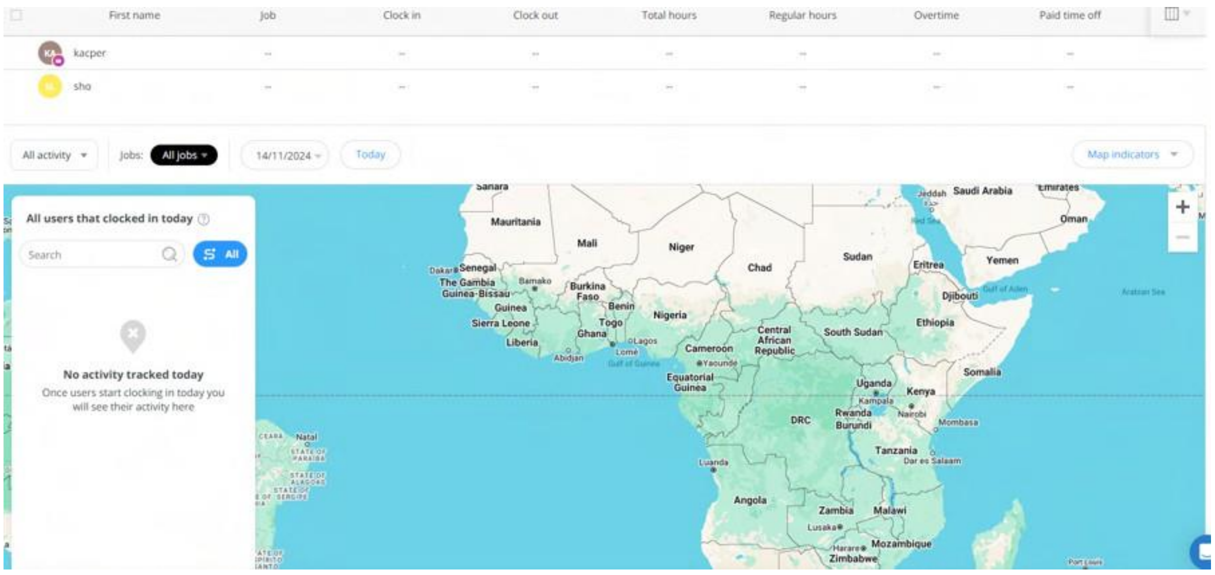
Fill the deposit sheet

Check water filter

Figure 8, Task Management (ConnectTeam (n.d))

Tracking Attendance

A GPS-enabled time clock feature allows employees to clock in and out from authorized locations, this feature can be seen in Figure 9. This data automatically integrates with timesheets for easy management access. This makes it easy for employers and employees to access and manage attendance records for payroll. This functionality increases the accuracy by verifying the user’s location when they clock in and out. However, GPS tracking can raise privacy concerns for employees.



First name	Job	Clock in	Clock out	Total hours	Regular hours	Overtime	Paid time off
kacper	–	–	–	–	–	–	–
sho	–	–	–	–	–	–	–

All activity · Jobs: All jobs · 14/11/2024 · Today · Map indicators

All users that clocked in today

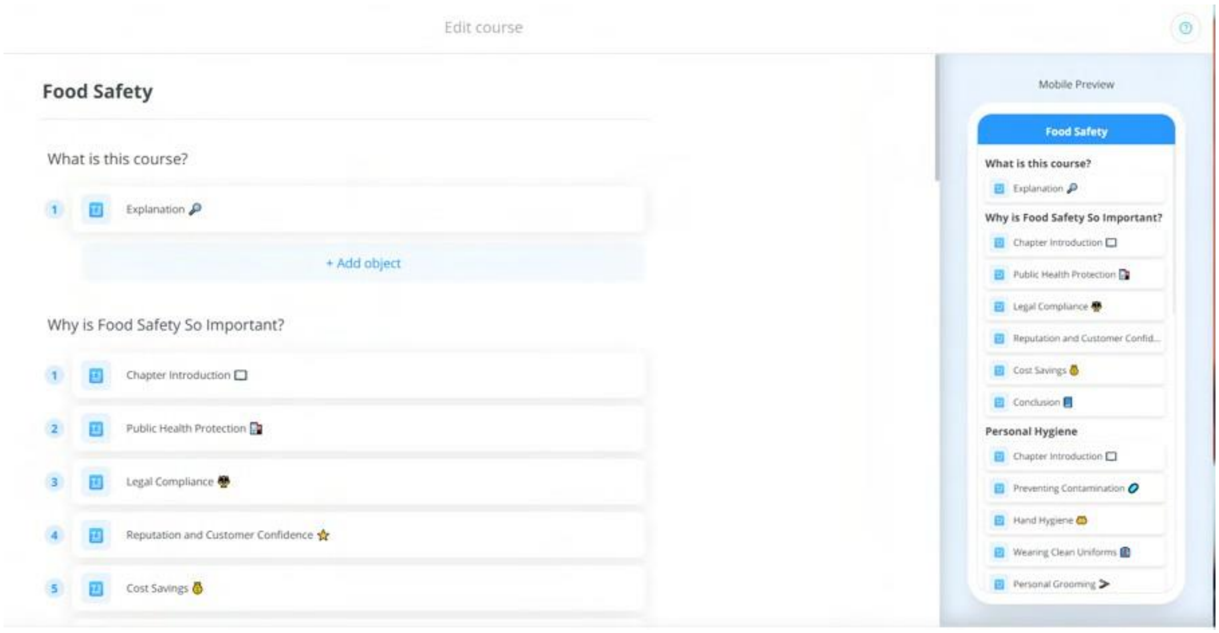
No activity tracked today

Once users start clocking in today you will see their activity here

Figure 9, Tracking Attendance (ConnectTeam (n.d)

Training

Managers can create customized training programs that include videos, quizzes, and assessments, which can be seen in Figure 10. Upon completion, employees receive certificates to acknowledge their achievement. The functionality also tracks progress, allowing managers to monitor progress of each program by employees. This feature supports continuous learning and helps improve employee skills. However, this method of learning may not fit each employee.


Edit course

Food Safety

What is this course?

1 Explanation

+ Add object

Why is Food Safety So Important?

1 Chapter Introduction

2 Public Health Protection

3 Legal Compliance

4 Reputation and Customer Confidence

5 Cost Savings

Mobile Preview

Food Safety

What is this course?

Explanation

Why is Food Safety So Important?

Chapter Introduction

Public Health Protection

Legal Compliance

Reputation and Customer Confid...

Cost Savings

Conclusion

Personal Hygiene

Chapter Introduction

Preventing Contamination

Hand Hygiene

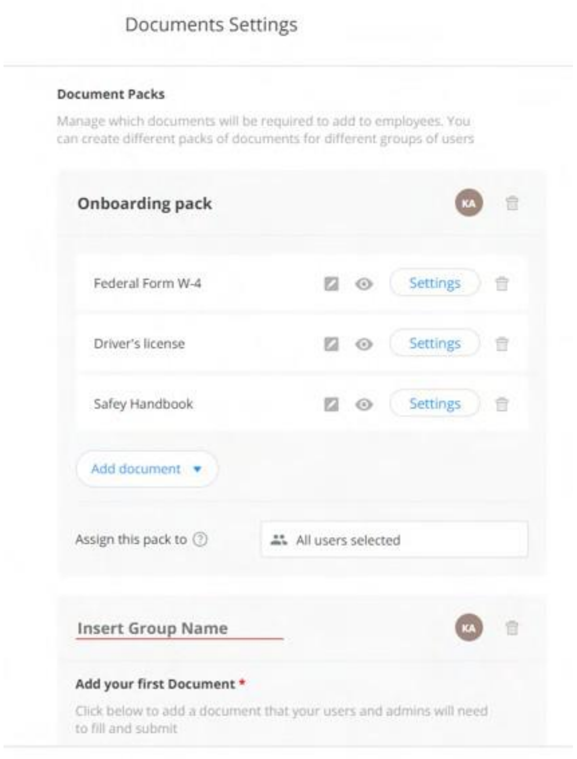
Wearing Clean Uniforms

Personal Grooming

Figure 10, Training (ConnectTeam (n.d))

Document sharing

Managers can upload essential documents such as employee handbooks, company policies, and custom forms, allowing employees to access, complete, and submit them directly through the app. This feature centralizes important resources, making it easier for employees to find necessary information and complete required paperwork without needing printed copies. This feature can be seen in Figure 11.


Documents Settings

Document Packs

Manage which documents will be required to add to employees. You can create different packs of documents for different groups of users

Onboarding pack

Federal Form W-4 Settings

Driver's license Settings

Safey Handbook Settings

Add document

Assign this pack to: All users selected

Insert Group Name

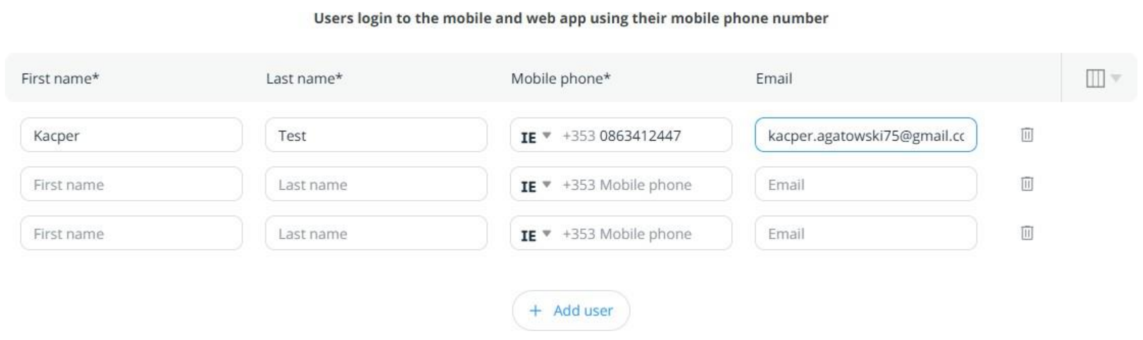
Add your first Document *

Click below to add a document that your users and admins will need to fill and submit

Figure 11, Document Sharing (ConnectTeam (n.d))

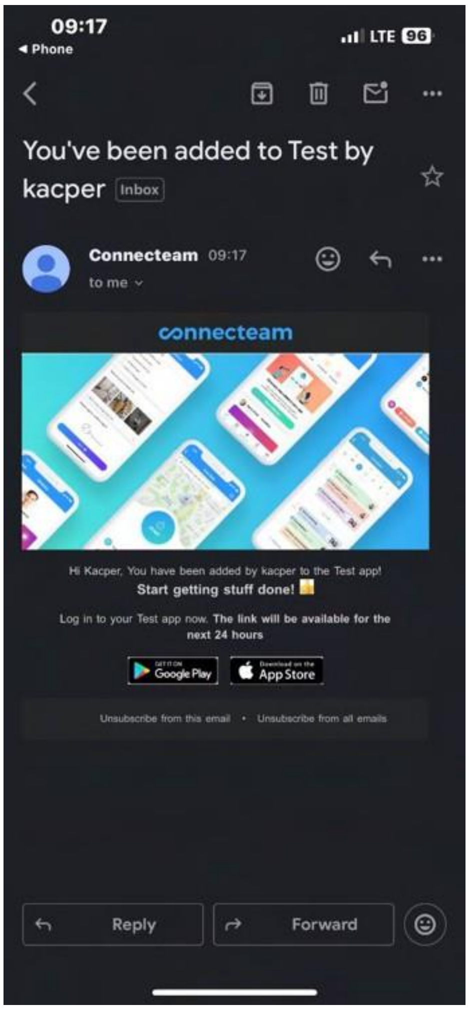
Notification API

New employees receive an automated invitation link via email to set up their accounts. This allows managers to assign roles, teams, and shifts even before the employee's first login, enabling new hires to access schedules and responsibilities immediately. The view of the employer and how they send the invitation link can be seen in Figure 12 and the received email by the employee can be seen in Figure 13.


Users login to the mobile and web app using their mobile phone number

First name*	Last name*	Mobile phone*	Email
Kacper	Test	IE +353 0863412447	kacper.agatowski75@gmail.cc
First name	Last name	IE +353 Mobile phone	Email
First name	Last name	IE +353 Mobile phone	Email

Figure 12, Notification API Sent (ConnectTeam (n.d))


You've been added to Test by kacper

Connecteam 09:17
to me

Hi Kacper, You have been added by kacper to the Test app!
Start getting stuff done!

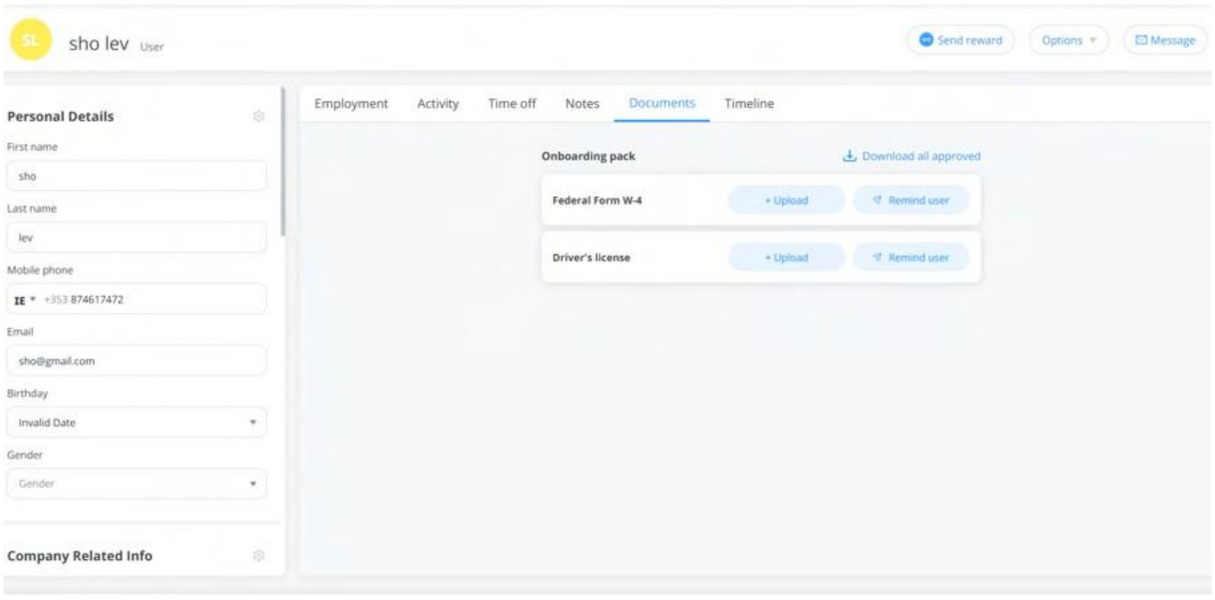
Log in to your Test app now. **The link will be available for the next 24 hours**

09:17
Phone
LTE 96
Inbox
connecteam
GET IT ON Google Play
Download on the App Store
Unsubscribe from this email • Unsubscribe from all emails
Reply
Forward

Figure 13, Notification API Received (ConnectTeam (n.d))

HR Tools

ConnectTeam provides a centralized space for storing key employee information, including contact details, wage data, and other relevant records, which can be easily accessed by managers and HR staff. The implementation can be seen on Figure 14. This feature simplifies record-keeping and ensures all critical employee data is up to date and readily available. It also allows employees to report incidents directly through the app, informing HR or employers immediately.



sho lev User

Personal Details

First name: sho

Last name: lev

Mobile phone: IE +353 874617472

Email: sho@gmail.com

Birthday: Invalid Date

Gender: Gender

Company Related Info

Onboarding pack

Federal Form W-4

Driver's license

Figure 14, HR Tools (ConnectTeam (n.d))

Non-Functional Aspects

ConnectTeam provides several non-functional benefits that improve user experience, scalability, and application reliability. These benefits include:

User-Friendly Interface

ConnectTeam features a visually intuitive layout with drag-and-drop scheduling. Key functions are organized for quick access, allowing managers to efficiently navigate tasks such as scheduling, communication, and tracking. While the dashboard may feel crowded due to the cluster of information displayed, its design prioritizes functionality by placing essential tools and features within easy reach, helping managers save time and improve productivity. By offering a customisable dashboard layout would allow businesses to tailor their business needs within the application.

Cross-Platform Compatibility

The application is accessible on both desktop and mobile devices, allowing users to access their data anytime and anywhere. Ensuring flexibility on all platforms makes it more convenient for users to use any functionality and allows for users to stay connected and coordinated. The full functionality of the desktop version can also be obtained within the mobile app, which means the user is not punished for using a smaller device.

Scalability

ConnectTeam support various business sizes. The application offers a hybrid pricing model, freemium plan for any small business with limited features. If the business size is increased to ten employees or more the freemium plan expires, and employers are pushed into purchasing a monthly plan which prices also depend on the number of users. Once you are a paying a subscription certain features are included.

Custom Options

The application allows you to view schedules based on a range of factors such as roles, departments, shifts or users. These custom options all for employers to view their schedules in relation to their needs by applying their own custom options.

Technical Support

ConnectTeam offer various ways of technical support including, customer support via email or live chat, tutorials and resources to help users understand and use certain features. This keeps users satisfied by maximizing productivity and resolving issues quickly. However, in relation to support via email, ConnectTeam is based in Israel and responses are automated and it may take a longer time for a response due to the global time difference.

3.2.1.2 HomeBase

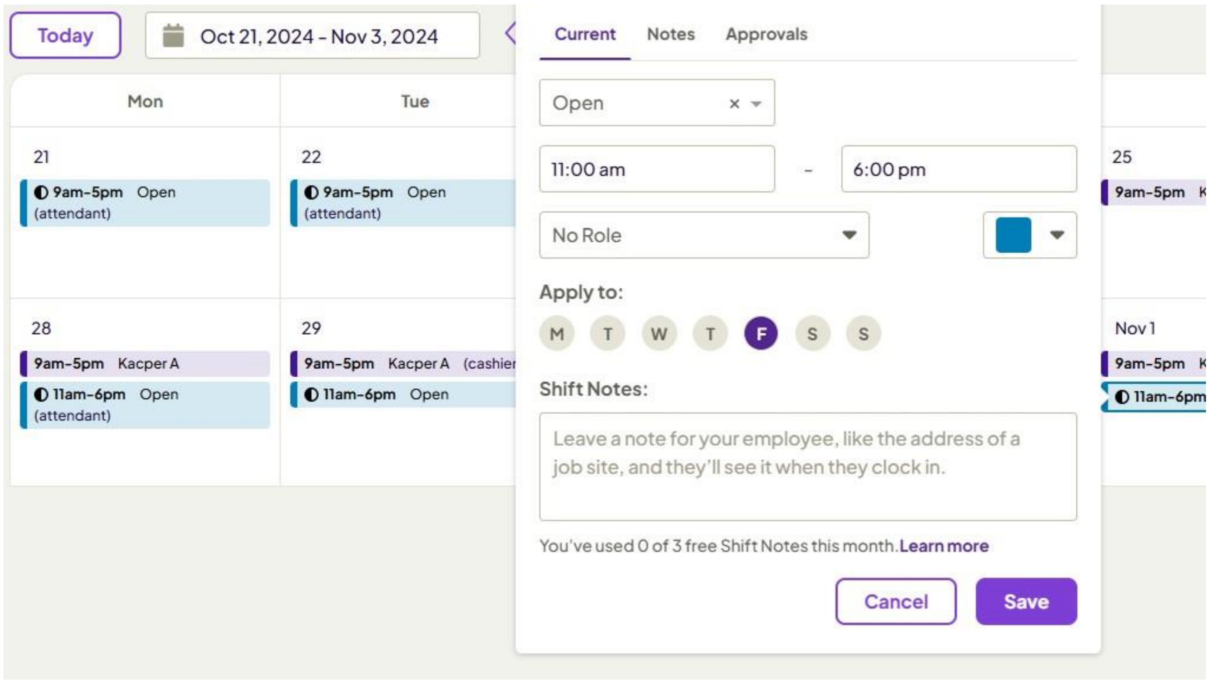
HomeBase is an employee management platform within the US that simplifies workforce management, including scheduling, attendance tracking, payroll, communication, and training. Designed for both employers and employees, it offers an intuitive interface and features that enhance efficiency and communication. Suitable for small teams or larger businesses with multiple locations, HomeBase helps organizations stay organized and comply with labour regulations.

Main Functional Aspects

Homebase provides a comprehensive set of features designed to support core aspects of employee management, such as:

Scheduling

Homebase's scheduling feature enables managers to create, edit, and assign shifts on a daily, weekly, or monthly basis, this can be seen in Figure 15. It includes options for setting repeating shifts, which saves time when creating rosters. Employees can input their availability and submit time-off requests directly within the application, and managers can approve or deny these requests in real-time, which can be seen in Figure 16. After completing schedules, the platform notifies employees of any updates, minimizing the risk of miscommunication and ensuring employees have the latest schedule information.



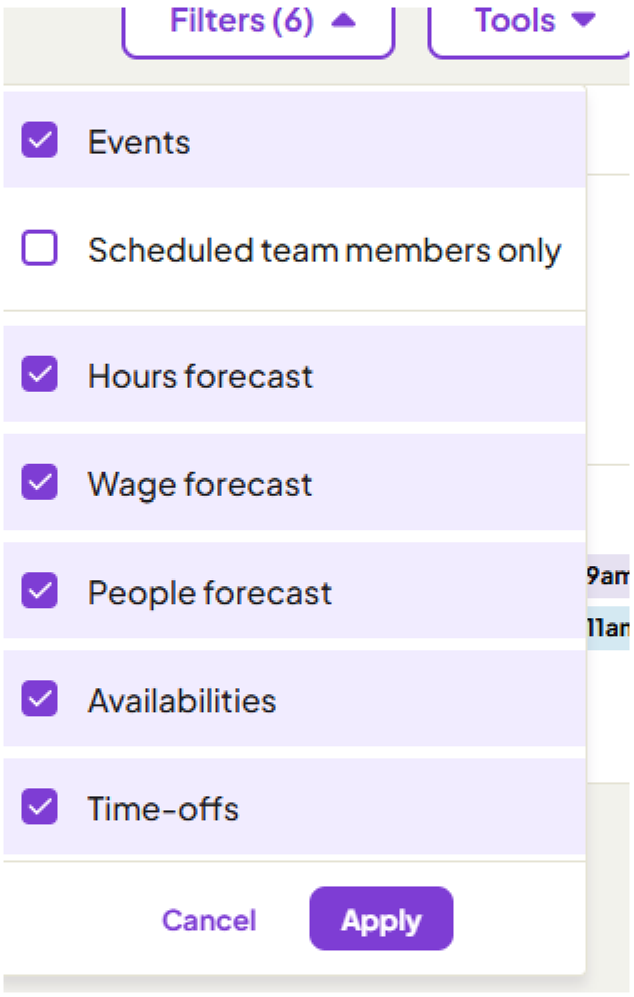
Oct 21, 2024 - Nov 3, 2024

Mon	Tue
21	22
9am-5pm Open (attendant)	9am-5pm Open (attendant)
28	29
9am-5pm Kacper A	9am-5pm Kacper A (cashier
11am-6pm Open (attendant)	11am-6pm Open

25
9am-5pm K
Nov 1
9am-5pm K
11am-6pm

Open
11:00 am - 6:00 pm
No Role
Apply to: [✓ F]
Shift Notes:
Leave a note for your employee, like the address of a job site, and they'll see it when they clock in.
You've used 0 of 3 free Shift Notes this month. Learn more

Figure 15, Shift Scheduling (HomeBase (n.d))


Filters (6)

Tools

☑ Events

☐ Scheduled team members only

☑ Hours forecast

☑ Wage forecast

☑ People forecast

☑ Availabilities

☑ Time-offs

Cancel

Apply

Figure 16 Request Filters (HomeBase (n.d))

Attendance Tracking

HomeBase offers a GPS-enabled timeclock that allows employees to clock in and out from specific locations. This feature helps managers verify employee attendance, including the time an employee arrives and leaves the location of work (Figure 17). The system also tracks breaks, allowing for accurate calculation of hours worked. This feature would be more beneficial for larger businesses to reduce time-theft. However, employees may find this as an invasion of their privacy.

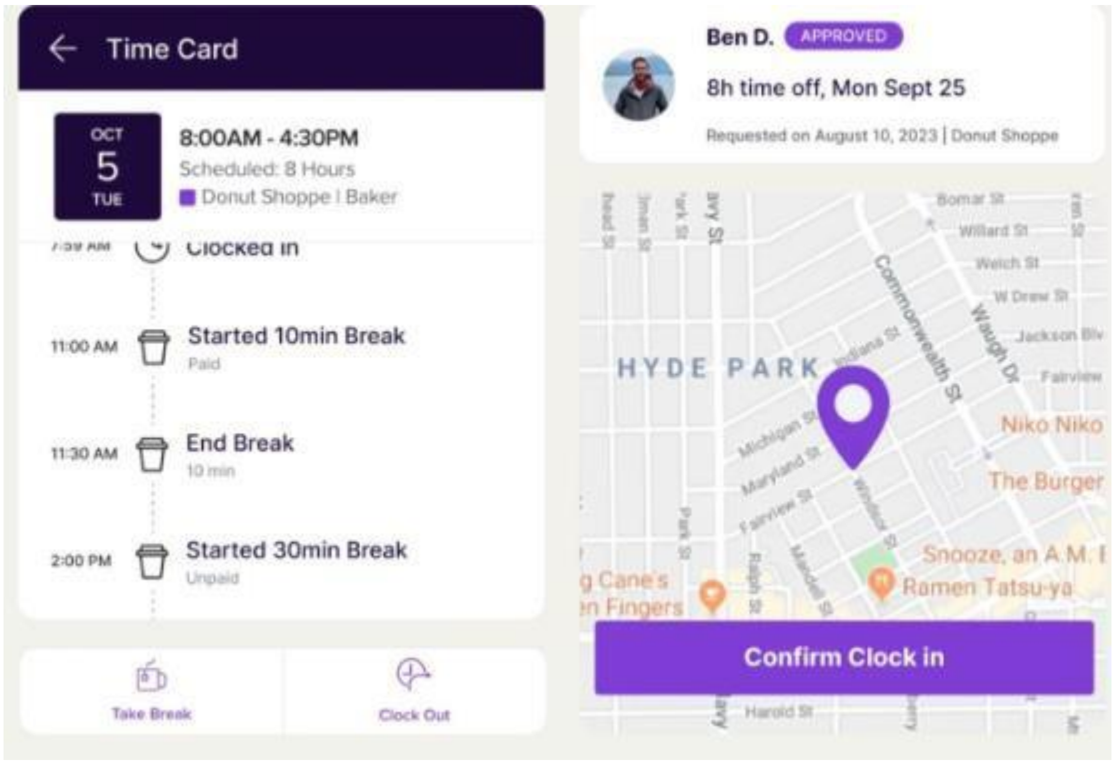
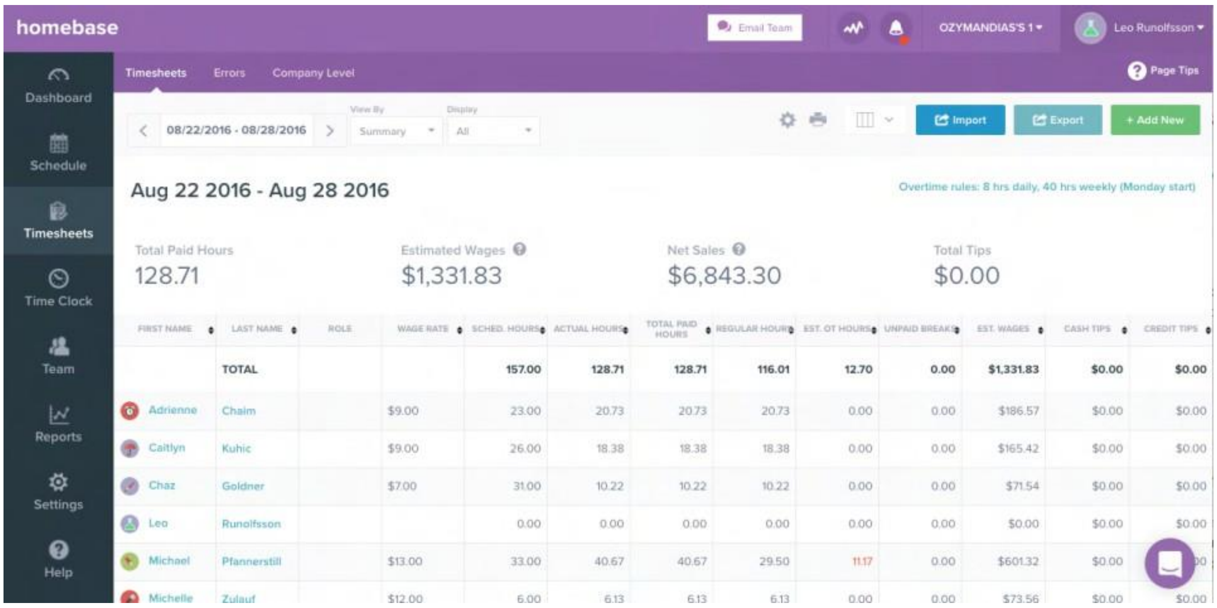


Figure 17, Attendance Tracking (Homebase (n.d))

Payroll Integration

The payroll integration feature calculates employee hours, including any overtime and break periods. This would relate to the attendance tracking feature; however, the employer can also manually input any hours. *HomeBase* integrates with popular payroll providers within the US, automating wage calculations and reducing the time required to process payroll, (Figure 18). This feature streamlines payroll processing and reduces human error, making it an asset for companies looking to simplify their payroll processes.



Aug 22 2016 - Aug 28 2016

Overtime rules: 8 hrs daily, 40 hrs weekly (Monday start)

Total Paid Hours 128.71

Estimated Wages $1,331.83

Net Sales $6,843.30

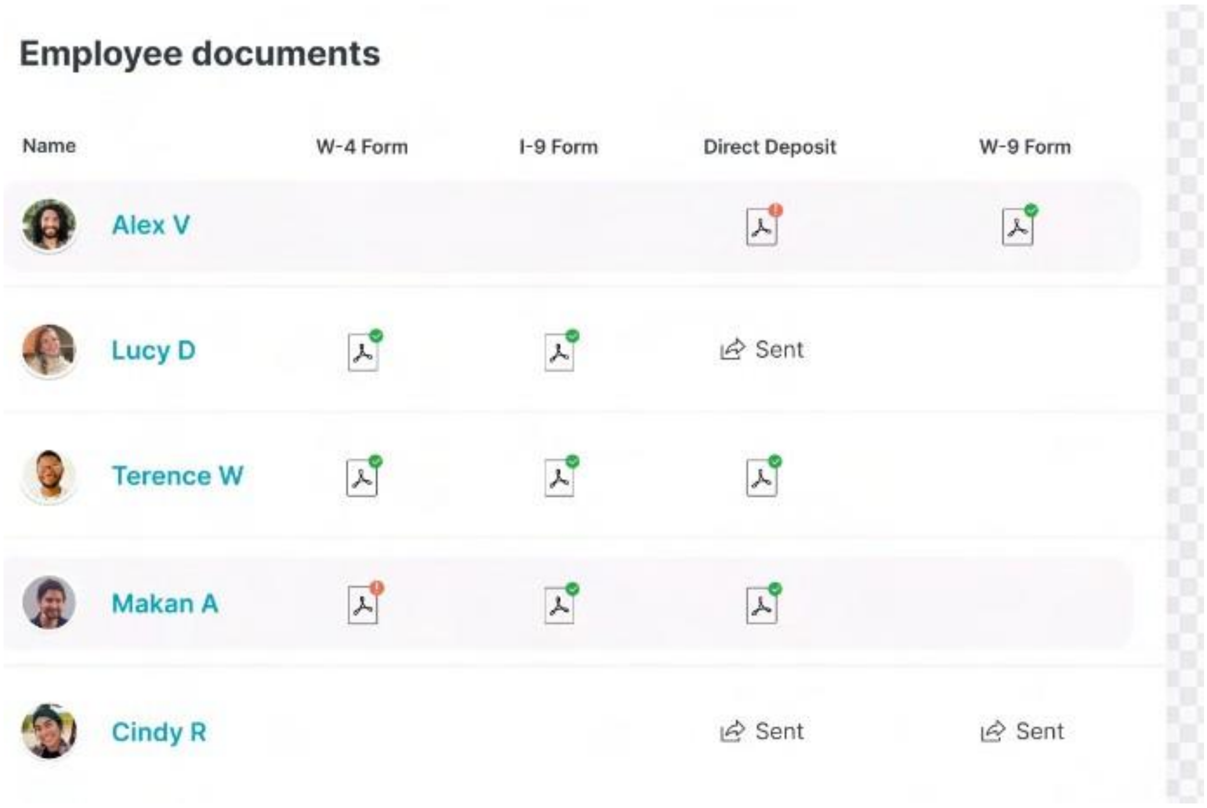
Total Tips $0.00

FIRST NAME	LAST NAME	ROLE	WAGE RATE	SCHED. HOURS	ACTUAL HOURS	TOTAL PAID HOURS	REGULAR HOURS	EST. OT HOURS	UNPAID BREAKS	EST. WAGES	CASH TIPS	CREDIT TIPS
	TOTAL			**157.00**	**128.71**	**128.71**	**116.01**	**12.70**	**0.00**	**$1,331.83**	**$0.00**	**$0.00**
Adrienne	Chaim		$9.00	23.00	20.73	20.73	20.73	0.00	0.00	$186.57	$0.00	$0.00
Caitlyn	Kuhic		$9.00	26.00	18.38	18.38	18.38	0.00	0.00	$165.42	$0.00	$0.00
Chaz	Goldner		$7.00	31.00	10.22	10.22	10.22	0.00	0.00	$71.54	$0.00	$0.00
Leo	Runolfsson			0.00	0.00	0.00	0.00	0.00	0.00	$0.00	$0.00	$0.00
Michael	Pfannerstill		$13.00	33.00	40.67	40.67	29.50	11.17	0.00	$601.32	$0.00	00
Michelle	Zulauf		$12.00	6.00	6.13	6.13	6.13	0.00	0.00	$73.56	$0.00	$0.00

Figure 18, Payroll Integration (HomeBase (n.d))

Employee Training

Employers can upload essential documents, such as employee handbooks, safety protocols, and training materials, to Homebase's centralized document storage. This function allows employees to access critical resources at any time and helps managers track employee certifications and training completion, which can be seen in Figure 19. This feature supports employee development and compliance with workplace safety standards.



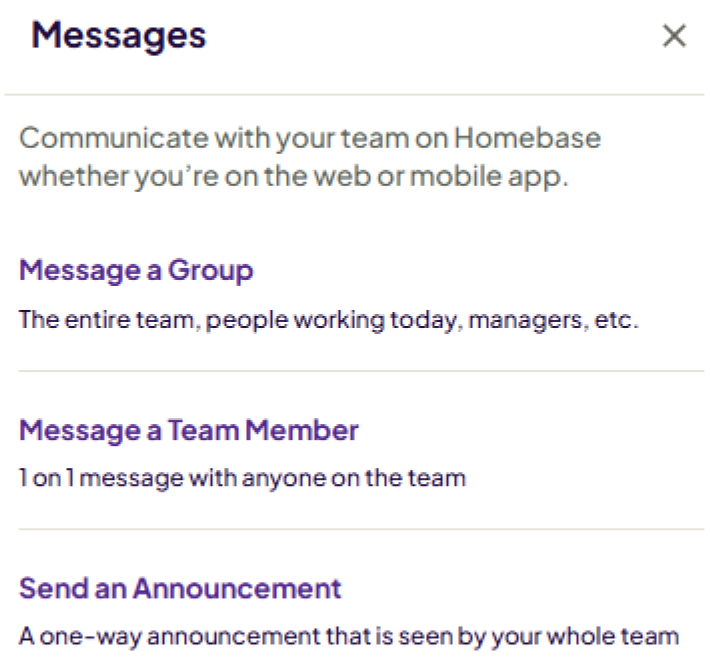
Employee documents

Name	W-4 Form	I-9 Form	Direct Deposit	W-9 Form
Alex V			❗	✓
Lucy D	✓	✓	Sent	
Terence W	✓	✓	✓	
Makan A	❗	✓	✓	
Cindy R			Sent	Sent

Figure 19, Employee Training (HomeBase (n.d))

Communication

HomeBase offers in-app messaging for real-time communication between employees and managers, Figure 20. Users can create group chats by roles or teams and send company-wide announcements. Managers can also start one-on-one chats. This centralized communication fosters a connected workplace and helps ensure important messages aren't missed.


Messages

Communicate with your team on Homebase whether you're on the web or mobile app.

Message a Group

The entire team, people working today, managers, etc.

Message a Team Member

1 on 1 message with anyone on the team

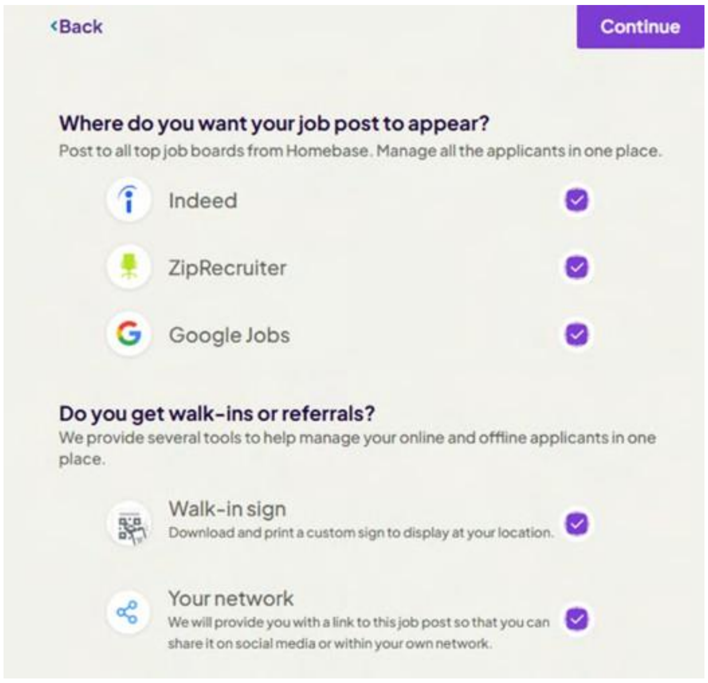
Send an Announcement

A one-way announcement that is seen by your whole team

Figure 20, Communication (HomeBase (n.d))

Hiring and Job Posting

HomeBase supports hiring by enabling managers to create job posts (Figure 22) and distribute them to multiple job boards (Figure 21). Employers can review applications within the platform, facilitating streamlined candidate selection and communication. This functionality aids in attracting and onboarding new talent.



Where do you want your job post to appear?

Post to all top job boards from Homebase. Manage all the applicants in one place.

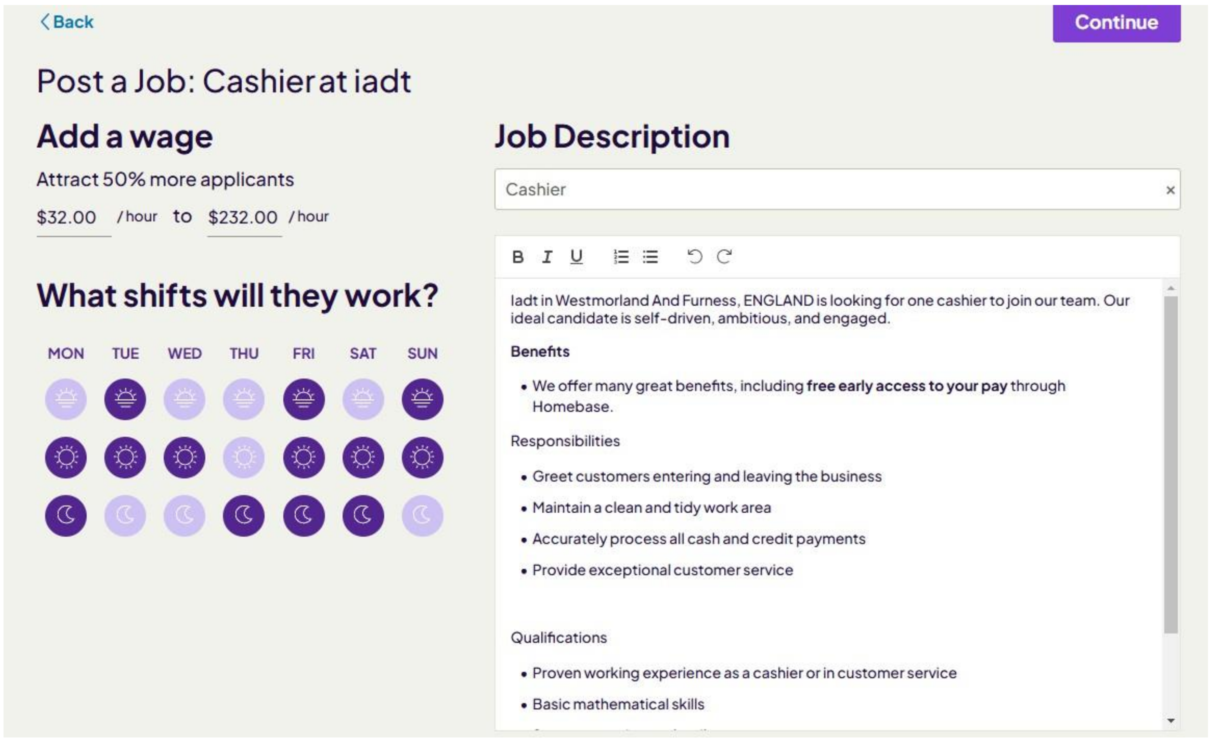
- Indeed ✓
- ZipRecruiter ✓
- Google Jobs ✓

Do you get walk-ins or referrals?

We provide several tools to help manage your online and offline applicants in one place.

- Walk-in sign — Download and print a custom sign to display at your location. ✓
- Your network — We will provide you with a link to this job post so that you can share it on social media or within your own network. ✓

Figure 21, Job Posting (HomeBase (n.d))



Post a Job: Cashier at iadt

Add a wage

Attract 50% more applicants

$32.00 /hour to $232.00 /hour

What shifts will they work?

Job Description

Cashier

Iadt in Westmorland And Furness, ENGLAND is looking for one cashier to join our team. Our ideal candidate is self-driven, ambitious, and engaged.

Benefits

- We offer many great benefits, including **free early access to your pay** through Homebase.

Responsibilities

- Greet customers entering and leaving the business
- Maintain a clean and tidy work area
- Accurately process all cash and credit payments
- Provide exceptional customer service

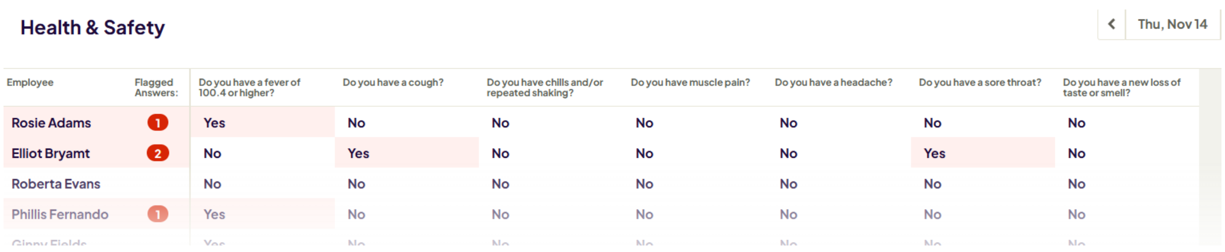
Qualifications

- Proven working experience as a cashier or in customer service
- Basic mathematical skills

Figure 22, Hiring (HomeBase (n.d))

Labour Law Compliance

HomeBase keeps managers updated on relevant labour laws, such as minimum wage requirements and overtime regulations, specific to each U.S. state, this can be seen in Figure 23. This feature is valuable for ensuring businesses remain compliant with evolving regulations and can avoid potential penalties. However, since the platform primarily targets U.S.-based businesses, international users may not benefit from this feature, as it lacks information on non-U.S. labour laws.


Health & Safety

Thu, Nov 14

Employee	Flagged Answers:	Do you have a fever of 100.4 or higher?	Do you have a cough?	Do you have chills and/or repeated shaking?	Do you have muscle pain?	Do you have a headache?	Do you have a sore throat?	Do you have a new loss of taste or smell?
Rosie Adams	1	Yes	No	No	No	No	No	No
Elliot Bryamt	2	No	Yes	No	No	No	Yes	No
Roberta Evans		No	No	No	No	No	No	No
Phillis Fernando	1	Yes	No	No	No	No	No	No
Ginny Fields		Yes	No	No	No	No	No	No

Figure 23, Labour Laws (HomeBase (n.d))

Non-Functional Aspects

HomeBase provides several non-functional benefits that enhance user experience, scalability, and application reliability. These aspects contribute to the platforms overall effectiveness in supporting businesses across different industries. These key non-functional benefits include:

UX Optimisation

HomeBase is designed with a user-centric approach, prioritising navigation and a clean layout. The platforms interface is simple, making it easy for employers and employees to access different features. All the key features are easily accessible, ensuring users can complete schedule related tasks efficiently. However, the initial sign-up process may require users o input detailed business and employee information before proceeding, this could push future customers away. HomeBase could consider implementing a more simplified sign-up process, such that larger business can pass a preexisting excel sheet with users' data and payroll to the system.

Cross-Platform Compatibility

HomeBase offers cross-platform accessibility though both a mobile app and web browsers, making it beneficial for both desk-based employees and teams on the go. This ensures that users can stay connected and manage and view schedules regardless of their location. Real-time updates between platforms allows users to make updates and view them on another device without delay, enhancing productivity and collaboration within the team.

Scalability

HomeBase caters to businesses of various sizes, which can be a small company with ten employees or a large enterprise with multiple locations. The platform offers flexible pricing plans which depend on the size of the business and the features you would like to use. If a business grows, they won’t need to look for a different platform to meet their schedule solution needs. As the platform can handle such large businesses, they also handle many data volume without affecting the performance of the application.

Customization

HomeBase provide customisation within scheduling payroll and task management features. This allows for business to tailor these specific functions to their requirements. Dependant on where the company is based the scheduling and payroll would be automatically implemented to meet the legal requirements, this leaves the employer stress free as the application double checks rules and regulations haven’t been broken.

Security

Given the sensitivity of the data HomeBase handles, payroll information, and personal data. The platform implements a security framework to protect its users. Features such as data encryption, secure login protocols such as twostep authentication (TOTP), and regular security audits are in place against unauthorized access. HomeBase is also compliant with industry regulations such as GDPR, ensuring that user data is handled in accordance with global privacy standards.

3.2.1.3 Conclusion

In conclusion, both *ConnectTeam* and *HomeBase* offer unique non-functional benefits that enhance the user experience, ensuring businesses can streamline their operations and improve efficiency. They provide functional aspects for employee management, each with its own strengths and features that cater to different business needs.

ConnectTeam's main aspects such as communication, customisable management features, and GPS-enabled attendance tracking, making it a great choice for businesses seeking an all-in-one package. Its scalability, cross-platform compatibility, and user-friendly interface make it suitable for organizations of all sizes. However, the crowded dashboard and occasional delays in support responses can be areas for potential improvement.

On the other hand, *HomeBase* focuses on simplifying workforce management, with strong features in scheduling, payroll integration, and labour law compliance. Its user-friendly design and cross-platform accessibility make it an excellent choice for businesses. While its clustered sign-up process might be a not appealing for some users, the platform offers a scalable solution for businesses that require flexibility and robust features as they grow.

3.2.2 Survey

This section covers the reasoning behind the questions asked within the survey, the responses received and the future actions to be considering with the development of the application. The survey was created within Google Forms and was distributed to businesses by email with the link provided or by physically approaching the business with a QR-code which brings the user directly to the survey. The survey was mainly sent to businesses where scheduling becomes a problem, this would include sectors such as hospitality, food service, retail, healthcare and transportation.

3.2.2.1 Reasoning behind questions

The survey was conducted to gather insights in relation to scheduling practices, challenges and interests within a business. The following explains the reasoning behind each question:

1. ***Company Name and General Location?***
   This question identifies the business and its general location. This allows for comparison in areas and labelling the business to the future questions asked.
2. **What is your Industry?**
   Each industry would have their own unique scheduling challenges. This question allows for industry-specific analysis.
3. ***How many employees do you manage?***
   This question provided checkboxes with five options to group the business in relation to their sizes. The complexity of scheduling often correlates with workforce size,

larger businesses generally would require more advanced scheduling solutions. Grouping each business with their specific challenges, allows for understanding the requirements dependant on the size of business.

4. ***How do you currently create employee schedules?***

   Understanding how the business performs their schedules gives an insight into the level of technological adoption and efficiency within the individual business.

5. ***If you use a scheduling software, which do you use? Skip if you do not.***

   If the business currently uses a scheduling software, it provides a new competitor and allows for further analysis to be conducted.

6. ***If you use a scheduling software, what are the best features about it? Skip if you do not.***

   Understanding why this business uses this scheduling software is important and gives more valuable insight for features to consider applying to the application.

7. ***What are the biggest challenges you face in scheduling employees? Multiple Choice***

   This question aims to identify the most common struggles businesses encounter when manging employee schedule. By providing multiple choice options, respondents can select challenges that relate to them and allows for grouping and sorting the answers based on trends.

8. ***What features would be most useful in an automated scheduling system? Multiple Choice***

   Identifying the most valuable functionalities that businesses look for in an automated scheduling system. By providing multiple-choice options, businesses can highlight feature they consider essential for creating schedules for their business.

9. ***How often do you need to adjust the schedule after it's created?***

   This question helps assess the stability of employee schedules within the respondent's business. Understanding the frequency of schedule adjustments allows for designing a system that minimises disruptions and improves efficiency and employee satisfaction within the business.

10. ***Would you be open to trying a new scheduling tool if it saves time?***

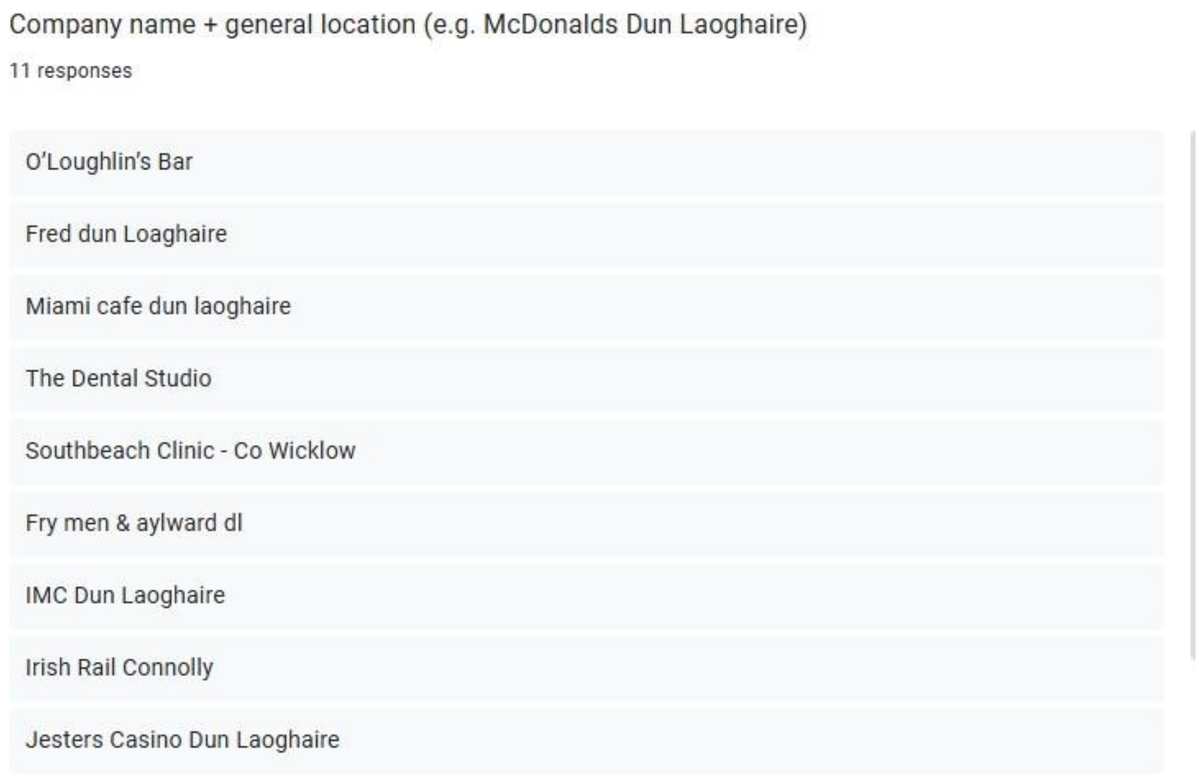
    Assessing the willingness of businesses to adopt a new scheduling solution if it offers efficiency improvements provides valuable insight into the openness of a business to change its current scheduling system.

11. ***What pricing model would be most acceptable?***
    Determining the preferred pricing model businesses would consider when adopting a new scheduling tool is crucial as the wrong pricing model can push potential businesses away from change to a new system.
12. ***Thank you for filling out the survey! If you would like to add any other relevant details, feel free to do so!***
    This final open-ended question allows for additional insight that may have not been covered by the questions. Any additional feedback could be valuable when refining the systems features.

3.2.2.2 Results

The results from the questionaire were underwhelming and did not provide substantial feedback. Many businesses weren't inclined to provide any details on their current schduling process. However, I received enough feedback to get an understanding of certain businesses needs and compare industries.

1. ***Company Name and General Location***
   In relation to Figure 24, the questionnaire reached Dublin and Wicklow County's. Businesses within Dun Laoghaire were more inclined to take the survey.


Company name + general location (e.g. McDonalds Dun Laoghaire)

11 responses

O'Loughlin's Bar

Fred dun Loaghaire

Miami cafe dun laoghaire

The Dental Studio

Southbeach Clinic - Co Wicklow

Fry men & aylward dl

IMC Dun Laoghaire

Irish Rail Connolly

Jesters Casino Dun Laoghaire

Figure 24, Result Q1 (Response from Questionnaire)

2. **What is your Industry?**
   Within Figure 25, the results show that the "Hospitality and Food Service" industry was the primary industry which was taking part within the survey. While "Retail & Customer Service" and "Healthcare & Medical", were joint second in relation to the industry type which took the survey.

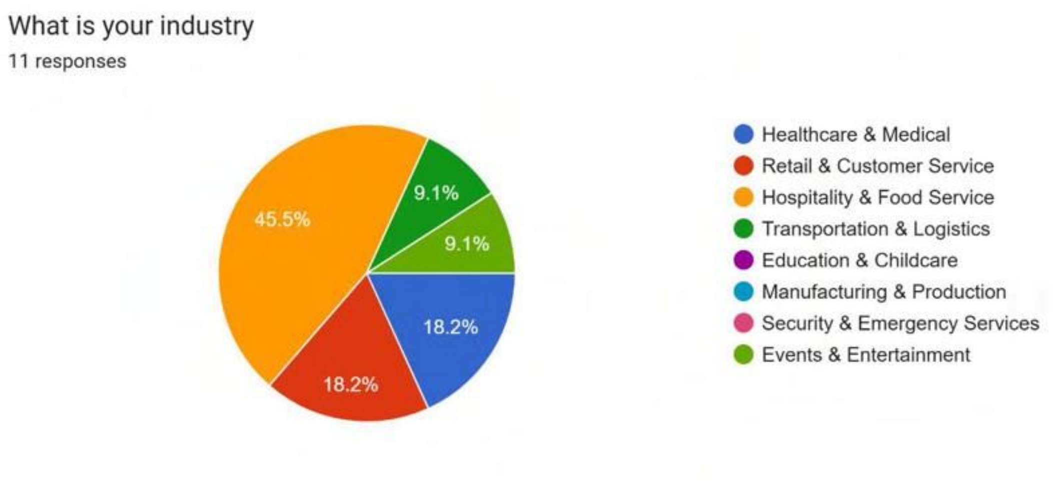


Figure 25, Result Q2 (Response from Questionnaire)

3. ***How many employees do you manage?***

   Smaller businesses participated in the survey, as shown in Figure 26. This suggests that the primary challenges and desired features identified are most relevant to smaller businesses.

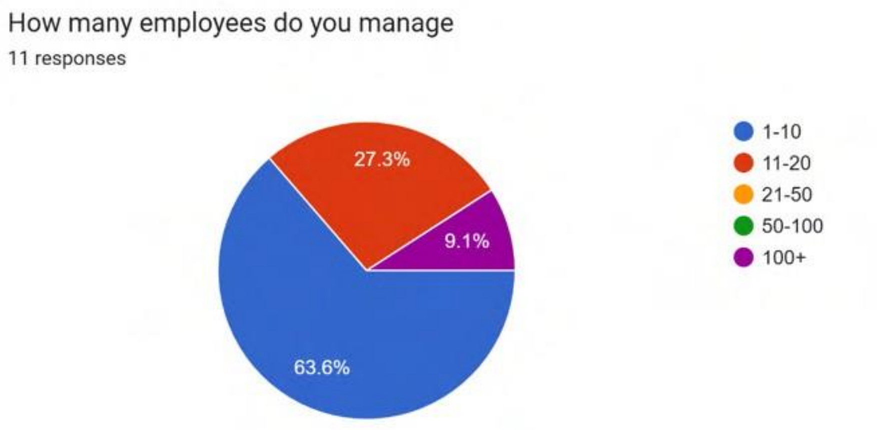


Figure 26, Result Q3 (Response from Questionnaire)

4. ***How do you currently create employee schedules?***

   From the results, we understand that 72.7% of the companies use some form of spreadsheet, such as Excel or Google Sheets, for scheduling. This is illustrated in Figure 27.

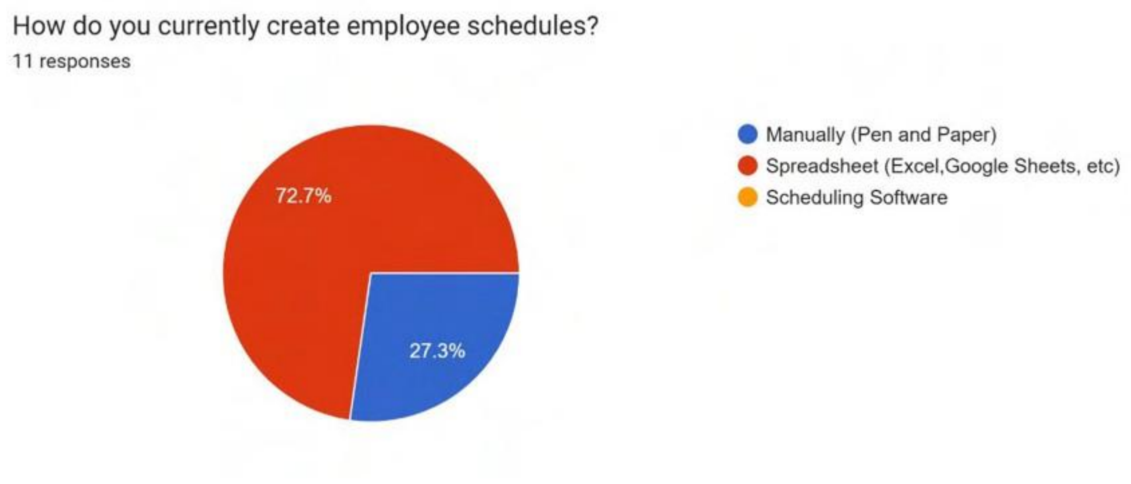


Figure 27, Result Q4 (Response from Questionnaire)

5. ***If you use a scheduling software, which do you use? Skip if you do not.***
   The survey received one response in this section, which was redundant as the business inputted "Sheets", which relates to the previous question. This question could have been more precise about scheduling software.
6. ***If you use a scheduling software, what are the best features about it? Skip if you do not.***
   In relation to this question, there were zero responses. As we can see in Figure 27, there were no scheduling software selected. This would result to this section being empty.
7. ***What are the biggest challenges you face in scheduling employees? Multiple Choice***
   While each challenge received selections, the most reported issues among businesses were last-minute changes (81.8%), employee availability conflicts (63.6%), and managing shift swaps (45.5%), as shown in Figure 28. This highlights key areas where scheduling improvements are most needed.

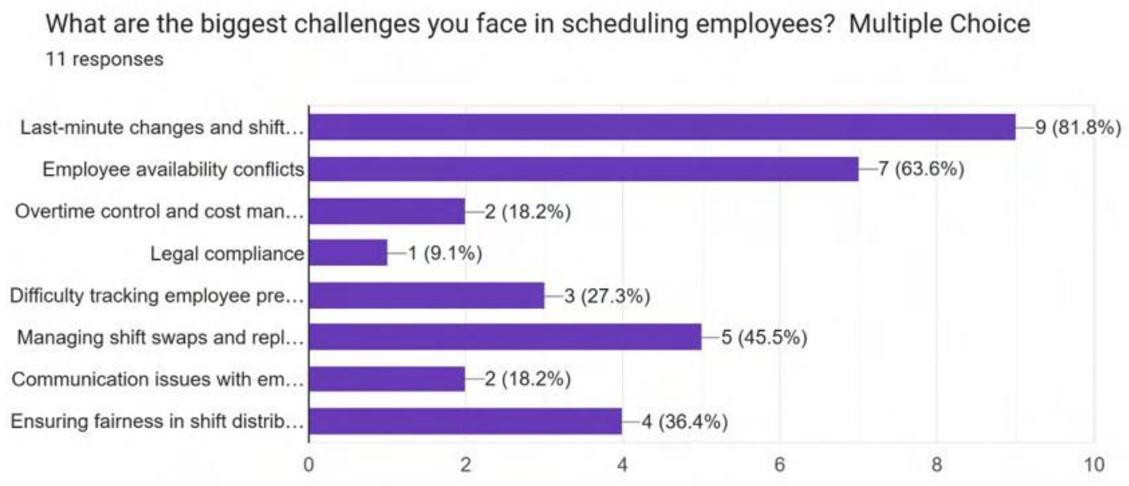


Figure 28, Result Q7 (Response from Questionnaire)

8. ***What features would be most useful in an automated scheduling system? Multiple Choice***

   While all features received selections, the most preferred functionalities in an automated scheduling system were customisable scheduling rules (e.g., max hours, shift preferences for employees) (81.8%), shift swap functionality (54.5%), and real-time updates and notifications for changes and creations (45.5%). As shown in Figure 29, these features highlight the key priorities businesses have for improving their scheduling processes.

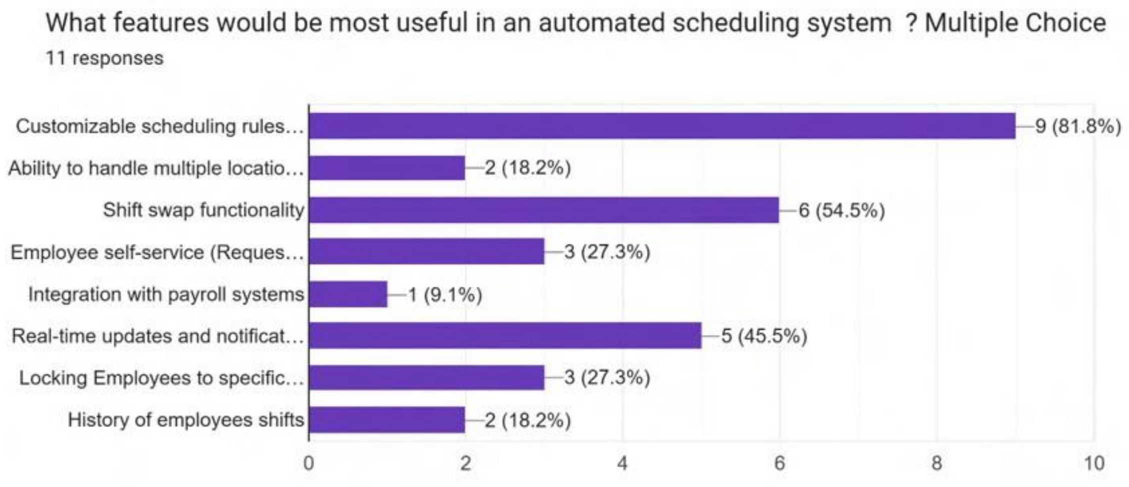


Figure 29, Result Q8 (Response from Questionnaire)

9. How often do you need to adjust the schedule after it's created?

The responses to this question indicate that schedule adjustments are a common occurrence for many businesses. Most responses reported they occasionally adjust their schedule. As Shown in Figure 30, these results suggest that businesses have a relatively stable schedule creation but do experience moderate changes. This highlights the need for a scheduling system in certain businesses.

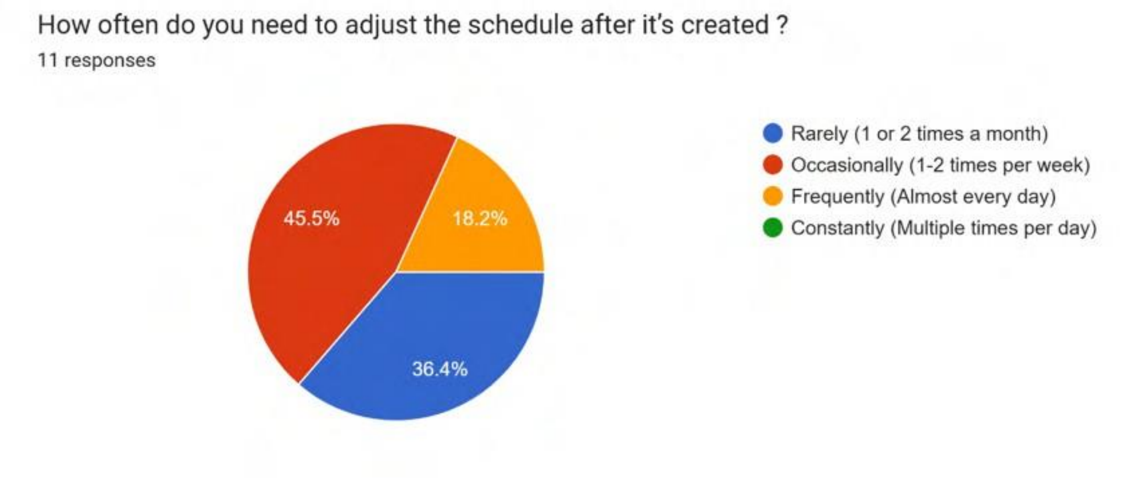


Figure 30, Result Q9 (Response from Questionnaire)

10. Would you be open to trying a new scheduling tool if it saves time?

The responses indicate a strong willingness to consider a new scheduling tool to save the business time. More than half of the respondents selected "Yes" (54.5%), this shows a clear openness for transition to a more efficient system. 36.4% of respondents selected "Maybe", suggesting that while they are not fully committed, they are still open to the idea if the application demonstrates clear improvements. As shown in Figure 31, these results highlight the significant opportunity for the adoption of an automated scheduling tool.

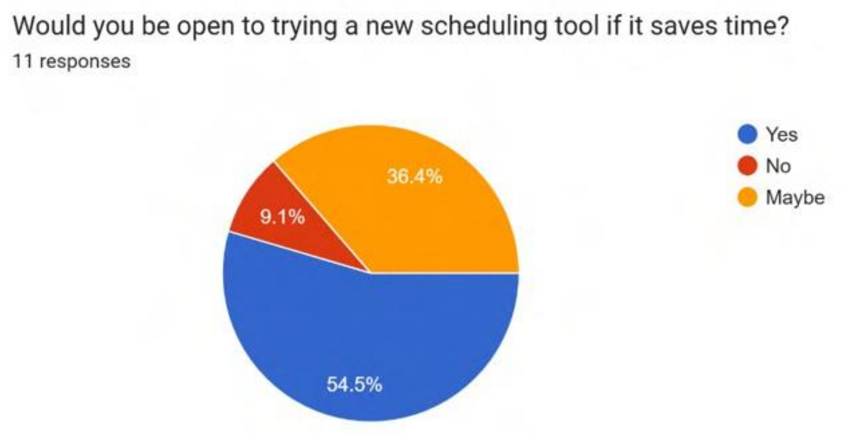


Figure 31, Result Q10 (Response from Questionnaire)

11. What pricing model would be most acceptable?

There was an even split between two preferred pricing models. 45.5% of businesses favoured a freemium model (free with limited features), which suggest that many would like to try the software before committing to a paid plan. Another 45.5% of businesses preferred a one-time purchase (life-time access), indicating that some businesses prefer a long-term cost efficiency over continuous payments. In relation to Figure 32, these results highlight the importance of offering multiple pricing options to cater to different business preferences.

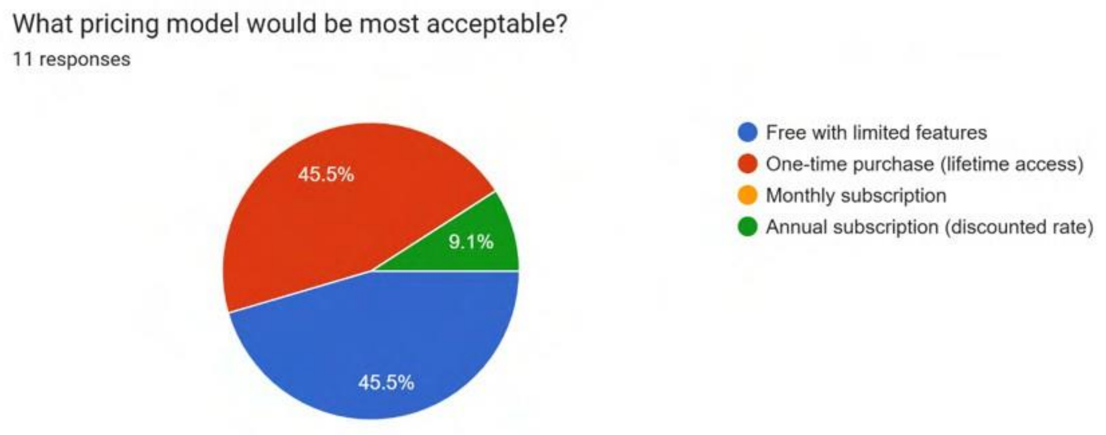


Figure 32, Result Q10 (Response from Questionnaire)

12. Thank you for filling out the survey! If you would like to add any other relevant details, feel free to do so!

This question received zero feedback, while it was disappointing no additional feedback was given, it also suggests that the questions effectively captured the key concerns and insights from the respondents for each business.

3.2.2.3 Actions to consider

Based on the survey results, several key actions should be considered moving forward in the development and requirements of the application. These insights provided valuable information on the challenges businesses face, their preference features and their openness to adopt a new solution to scheduling. Actions to consider would be:

1. ***Targeting smaller Businesses***
   Since smaller businesses were the primary respondents, it would be crucial to develop the requirements and design around smaller businesses especially within the hospitality and food service industries. By addressing this target audience, it provides potential growth for the application in these areas.
2. ***Addressing these scheduling challenges within the application***
   The survey gave insight into common challenges especially in last-minute changes, employee availability conflicts, and shift swaps. These key pain points need to be prioritised into the development and design of the application. This will result in better feedback from future users.
3. ***Implementing the most preferred features***
   By prioritising the popular features, then implementing them into the functional requirements and design, it could benefit the applications future growth and satisfy users' needs when dealing with scheduling.

3.2.3 Interviews

Two interviews were conducted with two separate business owners. Within both interviews, the same questions were asked in person. These questions were to understand their current scheduling process, challenges and potential features they would like to use. Both interviewees provided that they did NOT want to be recorded and asked to remain anonymous. In this case we will call the first interviewee Paul and the second, Fred. The following section includes the questions that were asked, the responses that were given, and an overall analysis.

3.2.3.1 Questions

1. Can you briefly describe your current scheduling process? (e.g. tools)?
2. How much time do you spend on average creating and managing schedules each week?
3. What do you feel is lacking in your current scheduling process?
4. What specific challenges do you face when scheduling employees?
5. How do you manage situations where employees request last-minute schedule changes or call in sick?
6. What issues, if any, have you encountered with managing different types of employees
7. How are you technically? Using computers and applications?
8. Would you consider using an application, which automates your schedules every week?
9. (If yes): What additional features do you think would help you like to see in this application?

3.2.3.2 Results interviewees 1 (Paul)

Paul currently is the owner of a very popular pub in Dublin.

1. "I am old fashioned, pen and paper"
2. "Weekly? I create my schedules monthly to give my staff time to base their free time around their work schedule. Generally, maybe an hour or so. It would really depend on if someone is going on holiday or has specific requests for some days off"
3. "Staff… If I had more staff the schedule would be easy to make. If I kept all the information on a whiteboard, I could remember how people can work and not have to change the schedule after its finished"
4. "Managing availability, but also my staff like to swap shifts often. Which is frustrating as there are times, I don't want a certain two people working together"
5. "If someone says they can't come in, I will generally try find someone else or come in myself, if I'm not working."
6. "I have a few full-time staff which is handy, they're reliable and good at their job. The part-timers always have some sort of requests, to do with college or its their dogs'

birthday. Some would tell me a month before, but some decide to tell me the day before."

7. "I'm not the brightest when it comes to computers, I know the basics of certain apps"
8. "I wouldn't oppose to it; it would save me some time for sure"
9. "As long as it gives back a good schedule and everyone is happy, I'm happy"

3.2.3.3 Results interviewees 2 (Fred)

Fred is the owner a coffee shop in Dublin and is currently opening another in the city centre.

1. "I use excel spreadsheets"
2. "Maybe one hour, an hour and a half"
3. "Lacking better management and information about my staff"
4. "Dealing with specific days off"
5. "I would generally try call someone if its mid-day shift but if its early morning I would come in myself"
6. "New staff comes in regularly and leaves, which is frustrating. I also have many part-time staff which are mainly available on weekends"
7. "Good"
8. "Maybe, depending on how it fits with my case"
9. "Tracking attendance for payroll, notifications about relating to the future schedules"

3.2.3.4 Overall Analysis

Both interviews were valuable insights to the requirements section. The responses highlighted that neither business use a scheduling software, instead to relying on manual methods. A common pain point shared was the management of employee availability. Paul expressed his frustration over frequent shift swaps and last-minute request, while Fred highlighted changes related to a specific day-off requests and a high staff turnover rate. This suggests that both companies need more management in their current scheduling process.

There was a difference between the two interviewees regarding technical familiarity, Paul admitted to having basic computer skills, while Fred indicated he was comfortable technically. This indicates the importance of a user-friendly interface. Both interviewees showed openness to an automated application, with Fred specifying that features like attendance tracking for payroll and schedule-related notifications would be highly beneficial.

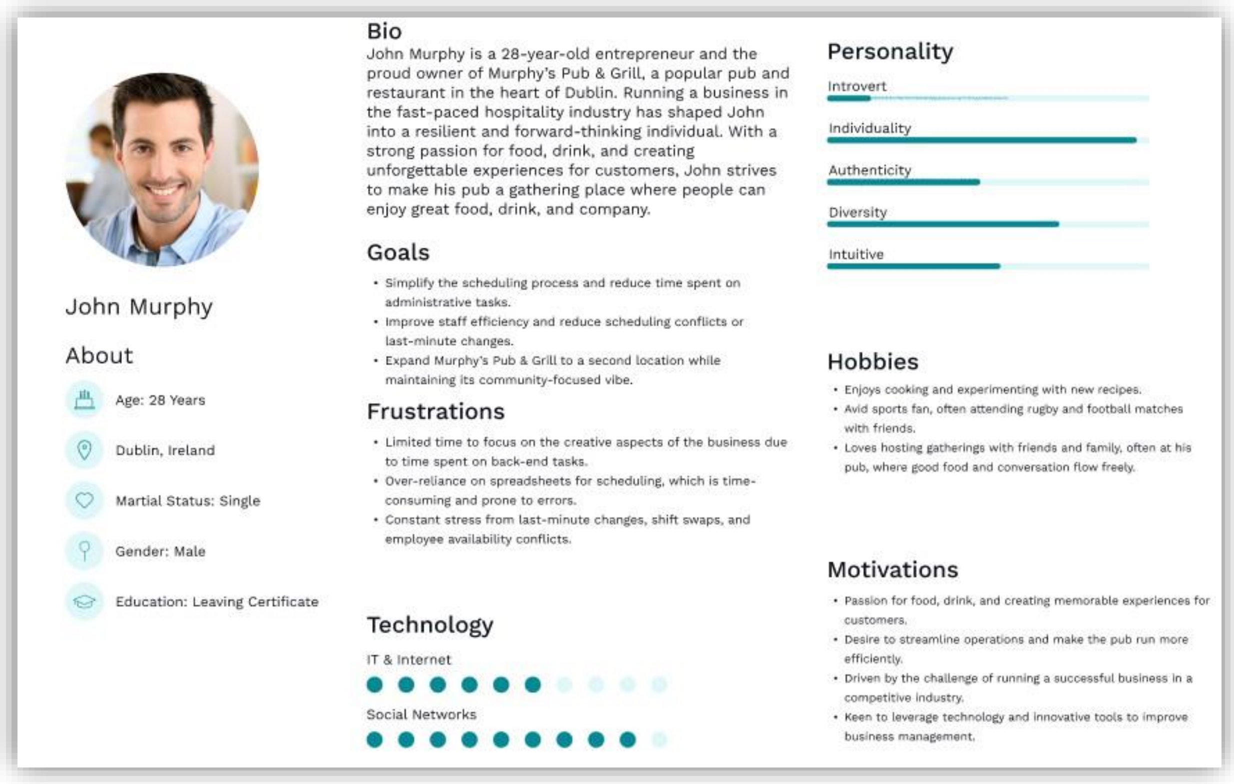
Paul, on the other hand, prioritized a system that generates fair and balanced schedules for staff satisfaction. These insights can help shape requirements model and boost satisfaction from similar customers in the future.

3.3 Requirements modelling

3.3.1 Personas

Personas are fictional characters which are created to help differentiate the user types that might use the application. The personas are based on the feedback from the survey respondents. There are two personas developed, each within a separate industry with different goals and motivations. Both personas both relate to a similar struggle or frustration, which is dealing with scheduling employees.

Below is an illustration of the first persona that was developed, John Murphy (*Illustration 1*). From the biography we can see that John owns Pub and Grill in Dublin who faces challenges in managing employee schedules. As highlighted in the survey response (*3.2.3.2)*, businesses in the hospitality industry often struggle with last-minute shift changes. John's persona reflects these common frustrations. Johns' motivations include improving operational efficiency to streamline scheduling. By understanding Johns challenges and goals it helps to design a scheduling application tailored to his small hospitality business.



John Murphy

About

Age: 28 Years

Dublin, Ireland

Martial Status: Single

Gender: Male

Education: Leaving Certificate

Bio

John Murphy is a 28-year-old entrepreneur and the proud owner of Murphy's Pub & Grill, a popular pub and restaurant in the heart of Dublin. Running a business in the fast-paced hospitality industry has shaped John into a resilient and forward-thinking individual. With a strong passion for food, drink, and creating unforgettable experiences for customers, John strives to make his pub a gathering place where people can enjoy great food, drink, and company.

Goals

- Simplify the scheduling process and reduce time spent on administrative tasks.
- Improve staff efficiency and reduce scheduling conflicts or last-minute changes.
- Expand Murphy's Pub & Grill to a second location while maintaining its community-focused vibe.

Frustrations

- Limited time to focus on the creative aspects of the business due to time spent on back-end tasks.
- Over-reliance on spreadsheets for scheduling, which is time-consuming and prone to errors.
- Constant stress from last-minute changes, shift swaps, and employee availability conflicts.

Technology

IT & Internet

●●●●●●○○○○

Social Networks

●●●●●●●●●○

Personality

Introvert

Individuality

Authenticity

Diversity

Intuitive

Hobbies

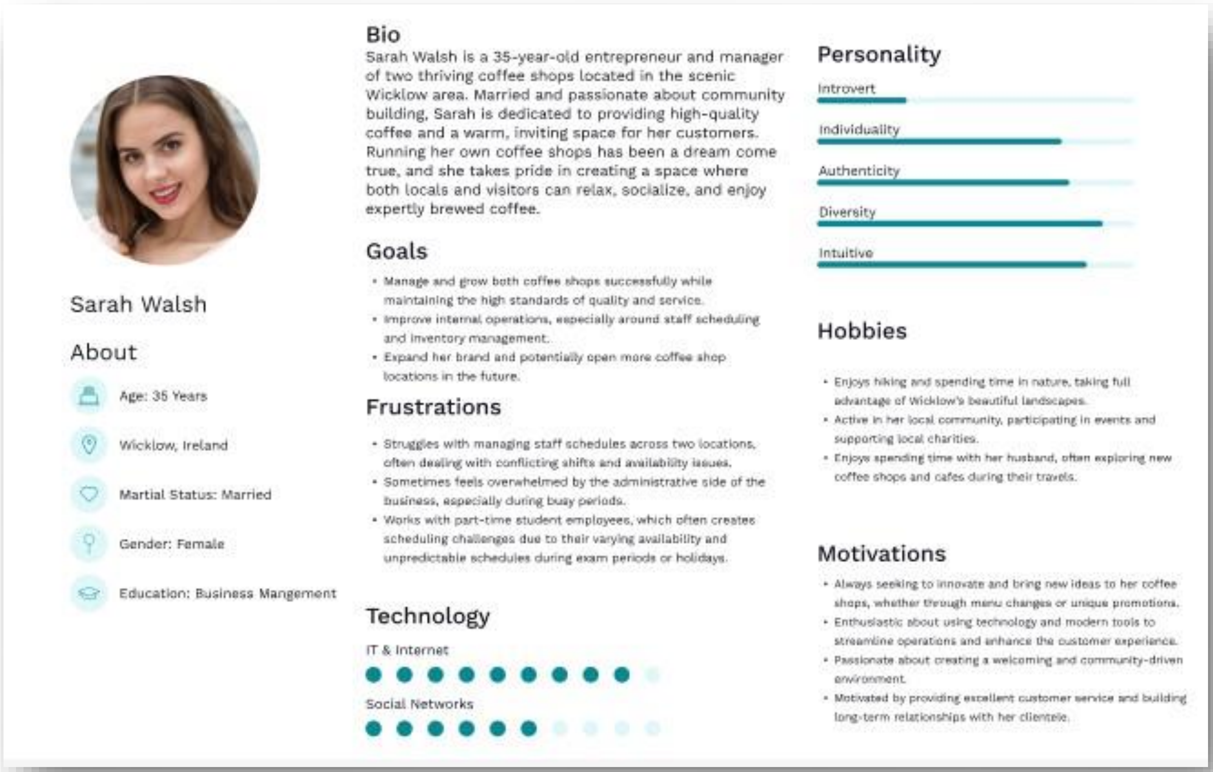
- Enjoys cooking and experimenting with new recipes.
- Avid sports fan, often attending rugby and football matches with friends.
- Loves hosting gatherings with friends and family, often at his pub, where good food and conversation flow freely.

Motivations

- Passion for food, drink, and creating memorable experiences for customers.
- Desire to streamline operations and make the pub run more efficiently.
- Driven by the challenge of running a successful business in a competitive industry.
- Keen to leverage technology and innovative tools to improve business management.

Illustration 1 (Persona #1)

An illustration of a second persona was developed, Sarah Walsh (illustration 2). Sarah is a manger of two coffee shops in Wicklow, she is facing similar challenges which are based on the survey responses (*3.2.3.2*). However, Sarahs team in the workplace includes part-time student employees, which creates availability conflicts. As Sarahs manages two coffees shops, she finds scheduling time-consuming due to the difficult availability problems, her persona emphasizes the need for a scheduling solution. Understanding Sarahs motivations and frustrations also helps shape the development of the application focusing on the pain points faced by small café managers.



Sarah Walsh

About

Age: 35 Years

Wicklow, Ireland

Martial Status: Married

Gender: Female

Education: Business Mangement

Bio

Sarah Walsh is a 35-year-old entrepreneur and manager of two thriving coffee shops located in the scenic Wicklow area. Married and passionate about community building, Sarah is dedicated to providing high-quality coffee and a warm, inviting space for her customers. Running her own coffee shops has been a dream come true, and she takes pride in creating a space where both locals and visitors can relax, socialize, and enjoy expertly brewed coffee.

Goals

- Manage and grow both coffee shops successfully while maintaining the high standards of quality and service.
- Improve internal operations, especially around staff scheduling and inventory management.
- Expand her brand and potentially open more coffee shop locations in the future.

Frustrations

- Struggles with managing staff schedules across two locations, often dealing with conflicting shifts and availability issues.
- Sometimes feels overwhelmed by the administrative side of the business, especially during busy periods.
- Works with part-time student employees, which often creates scheduling challenges due to their varying availability and unpredictable schedules during exam periods or holidays.

Technology

IT & Internet ●●●●●●●●●○

Social Networks ●●●●●●○○○○

Personality

Introvert

Individuality

Authenticity

Diversity

Intuitive

Hobbies

- Enjoys hiking and spending time in nature, taking full advantage of Wicklow's beautiful landscapes.
- Active in her local community, participating in events and supporting local charities.
- Enjoys spending time with her husband, often exploring new coffee shops and cafes during their travels.

Motivations

- Always seeking to innovate and bring new ideas to her coffee shops, whether through menu changes or unique promotions.
- Enthusiastic about using technology and modern tools to streamline operations and enhance the customer experience.
- Passionate about creating a welcoming and community-driven environment.
- Motivated by providing excellent customer service and building long-term relationships with her clientele.

Illustration 2, (Persona #2)

3.3.2 User Requirements

The system will be role-based, this would mean that user requirements would differ depending on the role of the user. The two roles in the system will be employer and employee, each would have their own requirements. However, some requirements would apply to both roles.

3.3.2.1 All Users

Within this section the requirements are for any role. This would mean that all the users which use the application should be able to access both requirements, login/registration and dashboard/notifications.

<u>Login and Registration</u>

Each user should be able to register for an account using a unique email address which is secured with an encrypted password. Once a user is registered, they should be able to log in securely using their credentials. Each user should be able to reset or recover their password, in the case that a user has forgotten their password. They should be able to update their credentials once they are logged in, this could include name, contact information, preferences, and employment option.

Dashboard and Notifications

Upon login, users should be directed to a dashboard which will hold relevant information depending on their role. Users should receive real-time notifications about updates such as schedule creations or changes and requests and response from one user to another relating to shifts and future schedules.

3.3.2.2 Employer

This section focuses on the user requirements of users which hold the employer role. The employer role can only be attached once the user has created a team. These user requirements would look like admin privileges but only within their team.

CRUD functionality of Team, Shifts, Expertise and Schedules

The employer should have create, read, update, and delete rights (CRUD) within the team they create, this would include shifts, expertise, and schedules. The most important requirement would be creating new schedules from the Constraint Satisfaction Problem Solver (CSPs) and editing the automated schedule to meet their needs. The employer will then be able to publish the finished schedule to let employees view the new schedule.

Managing Employees

Employers should only be able to add users to their team through requests, once the user accepts the invitation, they are a part of the team. Employers should have full rights to the users within their team apart from their personal details such as name and contact details. The employer should be able to assign expertise to a specific user. Employers should be able to view requests sent by employees about availability within their team and could send back responses of acceptance and decline.

Viewing Sensitive information

Employers should be able to view sensitive information such as other employees' information within the team which is relevant to holidays, preferences, and availability. Other information which should only be seen by the employer are the statistics of employees, such as total hours worked.

Employee

This section focuses on the user requirements of users which hold the employee role. The employee role is attached once the user accepts an invitation to a team. This role would now change the view of the user, giving them some rights to the team and other user requirements.

Viewing Schedules

Employees should see their work schedule clearly and in an organised format. The schedule should be easily accessible through the dashboard. Employees should also receive notifications whenever a new schedule is created or updated. His will allows users to be informed of their shifts and reducing conflicts and improving overall communication between the management and staff.

Submitting requests and receiving responses

Employees should have the ability to submit requests related to availability, holidays, and other work-related matters through the application. These requests should be logged and easily accessible for both the employee and employer. Employers should be able to review and respond to requests, with employees receiving real-time updates on the status of their request.

3.3.3 Technical Requirements

This section outlines technical requirements required to develop and maintain the application. This section will include the system architecture, which explains how components communicate between one another, and the technology stack, which outlines the frameworks, libraries and services which need to be used to build the system.

3.3.3.1 System Architecture

The system architecture defines the structural design and how the system interacts and communicates together. This will include the figure of the system architecture for visual explanation, how WebSocket's will encounter the front and backend, how the constraint satisfaction problem solver (CSPs) requests and sends data, the authentication needed into the application, and the communication between the front and backend.

<u>Illustration of the system architecture</u>

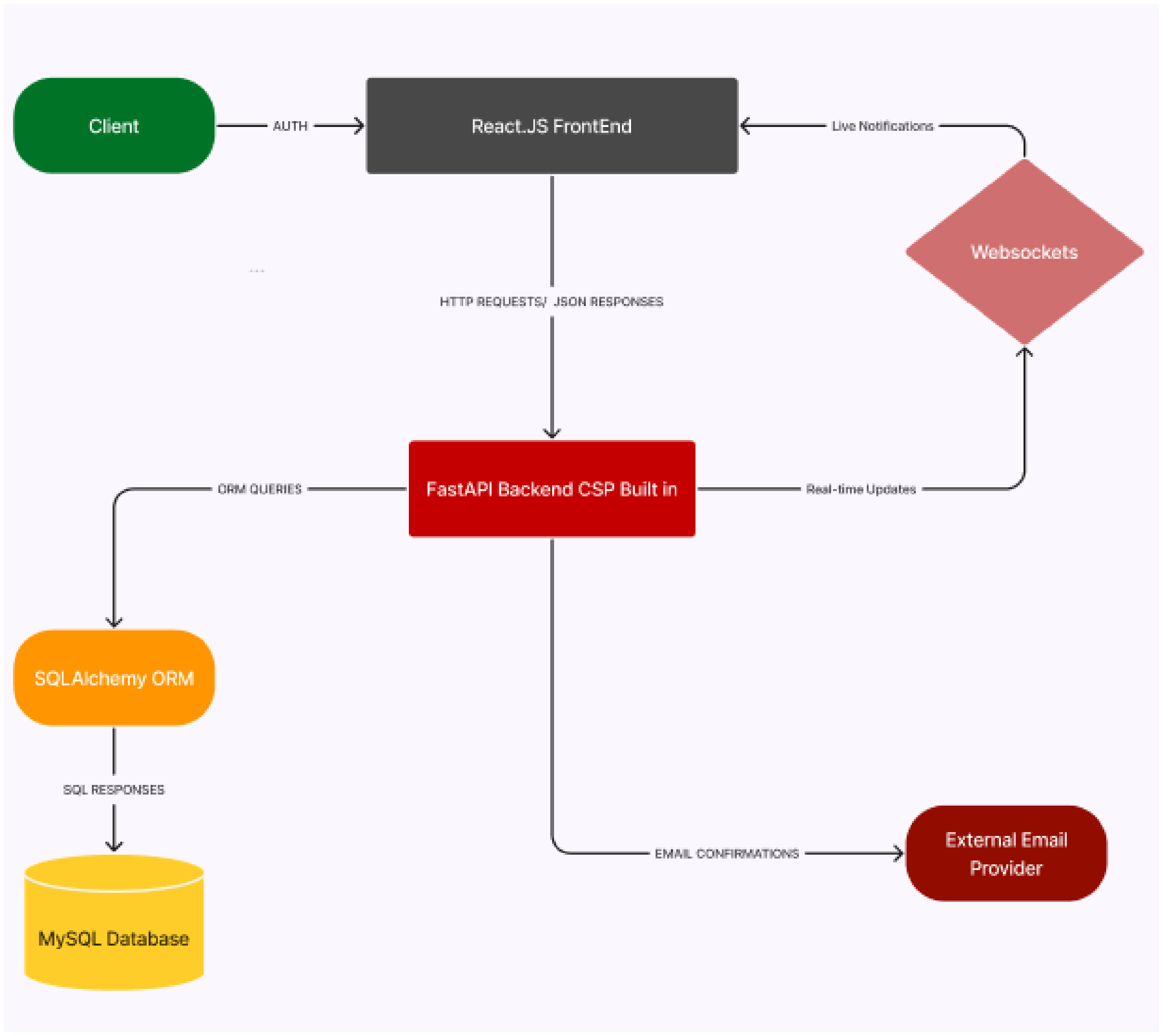


Illustration 3, System Architecture

WebSocket's

WebSocket's is a protocol which allows for real-time interaction between the client and server by keeping the connection open and allowing for continuous updates without making multiple requests. The client will initialise a connection to the backend using WebSocket, the backend must accept the WebSocket connection. Requests and responses will run asynchronously once the connection is established between both the frontend and backend.

This makes it possible to have real time interaction between the employer and employee, which can be through requests and responses about availability or receiving important announcements from employer to all team users. Using WebSocket's allows for other features to be brought in such as real-time chat between the employer and employee, this would keep the team connected.

Constraint Satisfaction Problem (CSP) Solver

The CSP must be established with the FastAPI backend, this will allow for quick a request and response from the database. The CSP needs all the variables, domains and constraints which will be established within the backend and stored within the SQL database. The response from the CSP will be sent directly to the database to store all solutions. This would allow for the solutions to be accessed by the frontend through backend routes.

Authentication

Authentication is crucial as it ensures only authorised users can access protected routes with unique functionality and protects users' sensitive information. Authentication must be done using token-based authentication, this is when a user receives a unique token upon login and can use this token to make protected requests. JSON Web Tokens (JWT) will be used to implement this stateless authentication. The JWT token would be created within the server, which requires specific libraries to be used such as 'python-Jose'. The token would be sent within the response after login, then the token must be collected and stored in the client side within HTTP Only cookies. Adding a token expiry would refresh the token after a certain amount of time to prevent from the token to be tampered with.

Communication between front and back end

The communication between the frontend (React) and backend (FastAPI) must be done using RESTful API endpoints over the HTTP protocol. They would interact through a series of HTTP GET, POST, PUT, DELETE requests, while ensuring the data is authenticated and validated within the middleware. The routes would be set up within the backend with a series of middleware, the frontend would send a request and if the request is successful, the backend would respond with a successful status code and additional data. If the request is invalid, the backend responds with an unsuccessful status code in the ranges 400 and provides a response to the frontend.

3.3.3.2 Technology Stack

This section will outline the key technologies selected for the development of the application, including the backend framework, database, frontend technologies, version control methods, real-time update strategy, and hosting solutions.

Backend

The backend of the application will be developed using FastAPI, which is a modern web framework for building APIs within Python. FastAPI provides high performance due to its asynchronous operations, offering Starlette for the web part and Pydantic for data validation. In addition, it uses Swagger and ReDoc which generates documentation in real-time and allows for viewing and testing of routes within the browser.

The backend will handle the business logic such as authentication and serving the API endpoints. The core functionality within the backend will be the Constraint Satisfaction Problem (CSP) Solver. The solver will process requests by taking in a set of variables, domains and constraints and then return solutions effectively in relation to the specific business scheduling problem.

Database

The application will need to use a relational SQL database, specifically MySQL, to store and manage data. MySQL would need to be implemented as the applications database will need to handle structured data, which is crucial for relational use cases such as storing team data.

The backend will interact with the database using SQL Alchemy, an Object-Relational Mapping (ORM) tool that takes SQL queries and converts them into Python objects. This would make it beneficial as it simplifies interactions between the database and backend. During the development process, MySQL Workbench must be used to visualise and manage the database schema and run any database management tasks.

Frontend

The frontend of the application must be developed using React.js, which is a JavaScript library used for designing reliable and fast web applications. The React frontend will be designed using Shadcn UI which is a customizable component library which offers a range of components, themes and typography. Other packages will be installed to allow for other requirements to be accomplished, these include, React-big-Callendar, React-DnD and, React-Router. These libraries will help streamline the complexity of the user requirements and enhance usability of the application.

Code Management and Version Control

Version control is essential for maintaining code and showing the progression of work. The project will use Git for version control, with GitHub serving as the central repository for code hosting. GitHub would be essential especially when tracking any errors or issues and marking them down after deploys.

Real-time Updates

As discussed in the WebSocket section, real-time updates will be implemented using WebSocket's, allowing for continuous communication between the frontend and backend without needing to repeat a singular HTTP request. To enable features like notifications and chat to work in real-time, built in libraries are available within FastAPI, where Socket.IO will be used within React. Importing and handling WebSocket's will be crucial for these real-time updates to work. The state of the connection can be handled using context within the frontend to determine if the current connection of the WebSocket is closed or open.

Hosting

Deploying the application for other users to use will be crucial after the development is complete. Hosting the application within Amazon Web Services (AWS) or Microsoft Azure, allows for the whole stack to be hosted within a single service. The application can also be hosted individually by having different services to host the stack. Hosting will be dependent on the progress of the application and the funding received to host.

Hosting with AWS, the frontend can be deployed with S3 and CloudFront for Global Content Delivery (CDN), the backend can be deployed using EC2 instances and the WebSocket's can be managed directly on EC2, then using Amazon RDS to fully manage the MySQL instance. This path could be more rewarding for the applications growth due to its scalability and reliability however, it could become very expensive to hold.

Hosting the application using multiple services can be beneficial for short term but can become a problem as the application grows. The frontend can be deployed using Vercel, which deploys on change with GitHub, the backend can be deployed with Render as it quick to set up with FastAPI and supports WebSocket's, the database can be hosted on PlanetScale which offer free offers dependant on the scale of the database.

3.3.4 Functional Requirements

This section covers the functional requirements of what features and functions the user can do within the application. The functional requirements were developed based on responses of the survey (3.2.3), responses from the interviews (3.2.2), and the research on competitors with similar applications (3.2.1). The functional requirements are not displayed in order of importance but are all crucial for the deployment of the application.

3.3.4.1 User Management

User management focuses on what the user can do with their own account. A user should be able to register an account within the application once the requirements within the register form are successful. A user can log in with the correct credentials and if the credentials are incorrect, they should be notified with an error. If the user has forgotten their credentials, for example a password, user should be able to reset their password. Once the user is authenticated and logged in, they should be able to change their details. These details include, name, password, email address, mobile number and availability. In relation to the user requirements (3.3.2), the system should enforce role-based access control (RBAC) to provide different functionality and features to the two different roles (Employer and Employee).

3.3.4.2 Team and Shift Management

Upon login, user's functionality and view depends on their role. Users with the employer role have full access and rights to all the data in relation to the team. This would mean that employers can Create, Read, Update and Delete, Users and their availability from the team, except for their personal information, the shifts within the team, the expertise within the team, the schedules within the team. The employees only have read rights within the team to tables such as shifts, approved schedules, and have full rights to their own availability.

3.3.4.3 Requests and Approvals

Employees have specific requests within the team, which then needs to be approved by the employer. These requests from the employee can be regarding to holidays, availability and shift swaps. These requests must be approved by the employer for the solver to consider them as constraints. These request and approvals must be passed through WebSocket's to allow real-time updates and notifications between the employer and employee.

3.3.4.4 Scheduling System

Once the RBAC is implemented only the employer should be able Create, Read, Update and Publish Schedules. The system will use a Constraint Satisfaction Problem (CSP) Solver to generate schedules based on the variables given from the team and can be generated only by the employer. Employers when updating a generated schedule must be provided with all the statics on users and be able to use features to change the view of the schedule and use the drag and drop to remove and add users to generated shift assignments. The Employer should be able to lock specific assignments on the generated schedule such as a specific employee on a shift and regenerate the rest of the schedule around that locked assignment. The employer should be able to publish the finished schedule, this allows for both employees and employers to view the published schedule. The employees should be notified through real-time updates about any changes or creations of schedules.

3.3.4.5 Notifications and Alerts

The system should send real-time notifications to all users within the team about new or updated schedules and approved or rejected requests regarding availability. The employer can also send out alerts to the employees of the team about any important announcements. Any notifications should be sent in-app and via email service library, this will decrease the chances of any confusion.

3.3.4.6 Reports

Employers should be able to download previous schedule week reports. The report would contain employee work hours, shift statistics, employee request history and other relevant information. This report can be then downloaded as PDF or Excel file.

3.3.5 Non-Functional Requirements

This section defines how the system should perform, focusing on aspects such as performance, usability, availability and security. Meeting these requirements would lead to positive user experience and cost-effective application.

3.3.5.1 Performance and Scalability

The system should handle several operations, the key operation would be the Constraint Satisfaction Problem (CSP) Solver. The CSP should be capable to handle up to one-hundred employees and provide several solutions if available. The CSP should generate a set number of solutions and be stored in the database. A solution which is not chosen should be deleted automatically to keep the database cleaned from any unnecessary data.

3.3.5.2 Usability and Accessibility

The application should prioritise the user experience by providing a responsive application on all breakpoints, specifically on mobile and computer. Creating a user-friendly dashboard which is easy to navigate must be responsive to the change of breakpoint to increase user experience. Creating an application which is accessible on all browsers would increase the accessibility of the application. By implementing WCAG 2.1 accessibility standards can ensure the platform is accessible for users with disabilities.

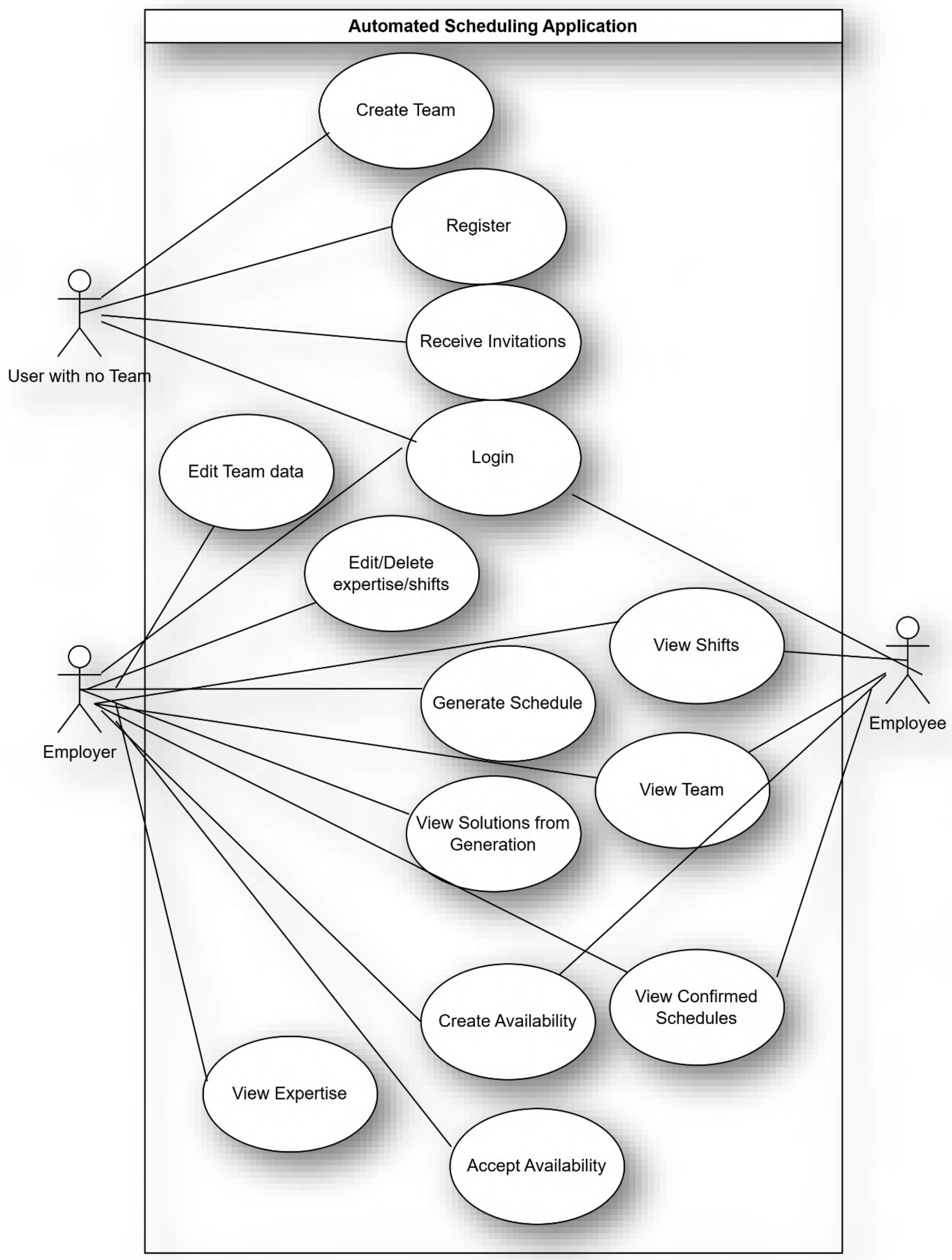
3.3.5.3 Security and Access Control

Security and access control is a fundamental requirement to protect the user’s data and ensure trustworthiness of the application. The application must use JSON Web Tokens (JWT) for user authentication. Users’ passwords, mobile numbers and email addresses must be encrypted using a strong hashing algorithm such as bcrypt. Users will receive specific roles depending on the team they’re in. This ensures that only users with a specific role can see certain features and pages.

3.3.6 Use Cases

This section will include a use case diagram which is based on personas which were developed. This use case diagram represents the different types of users’ interaction with the system and the different functions they can access. Two use cases were developed to capture different scenarios the user might interact with the software. They originate from the use case diagram, placing personas in common scenarios where they would interact with the application.

3.3.6.1 Use Case Diagram



3.3.6.2 Use Case 1

Description: New User – Creating the skeleton of the schedule

Actor: Sarah Walsh

Scenario: Sarah is finally going home after an eleven-hour shift. Sarah was meant to work a seven-hour shift but made a mistake in this week's work schedule. She had scheduled an employee which has informed her in the past that they cannot work that specific day due to college. The employee did not show up and therefore left Sarah working a double shift alone. Sarah being tired and frustrated, she searches on Google for a schedule helper and come across RosterReady. She reads the description of the application on the home page, she anticipates she must create an account before using any features to help her with her schedule.

Flow of events:

1. Navigate to Register Page, fill the form correctly and submit.
2. Create shifts dependant on current schedule.
3. Attach the shifts to the necessary days to be repeated.
4. Navigate to Callender Page, double check the view of the schedule is correct.
5. Invite users to the team.
6. Create expertise's.
7. Attach Expertise to shifts (if required).
8. Wait for users to join the team then apply created expertise to users (if required).

3.3.6.3 Use Case 2

Description: Registered Employer – Generating next week's schedule

Actor: John Murphy

Scenario: It is just after midnight on a Friday night. John has just finished work and has been getting texts from his employees about next week's roster. John opens his RosterReady application on his laptop and needs to generate a schedule for his employees to be satisfied.

Flow of events:

1. Login.
2. View requests regarding availability within the inbox.
3. Try accepting all requests and decline the ones that provide you an error.
4. Navigate to Callender and view the week the roster needs to be generated for.
5. Press Generate.
6. Lock certain assignments which are suited.
7. Regenerate a roster around the locked assignments.
8. Review and Accept Solution.
9. Navigate to Callender and view the results for the chosen week.

3.4 Feasibility

The feasibility analysis section will evaluate the practicality of implementing the proposed system through multiple perspectives, these include technical, operational, management, economic and legal considerations. Addressing potential challenges and strategies will ensure the project will be cost effective, efficient and successful.

3.4.1 Technical Feasibility

In relation to the technical requirements section (3.3.3), the system will be developed using a FastAPI backend, a React frontend and a SQL database. WebSocket's will be implemented to create open connections for real-time updates and authentication will be managed through JWT security. The technologies are well-suited but certain challenges could arise such as computational performance of the CSP solver, Server overload due to WebSocket's, and risks with the interaction between the technology stack.

Computational performance of the CSP solver:

In relation to the research section, specifically CSPs (2.3). CSPs can be computationally expensive, due to factors such as, input size and problem dependant. To mitigate this, a caching strategy will need to be implemented to keep input local. However, this may lead to data exposure if the data is not encrypted.

Server overload due to WebSocket's:

Maintaining real-time requests for multiple users at once could increase the server load, leading to a server overload. Using an additional tool such as Redis Pub/Sub can help scale the WebSocket's and reduce the chances of a server overload.

Risks with the interaction between the technology stack:

To keep a seamless connection between the stack, additional tests and debugging must take place to ensure that every vital interaction is tested and debugged before hosting.

3.4.2 Operational Feasibility

The system is designed to simplify scheduling for both employers and employees while eliminating mistakes and keeping both up to date with information. Issues may arise if users are new or have limited technical skills. To mitigate these issues, user adoption and usability strategies must be implemented.

New User Adoption:

Employers and Employees may require onboarding to understand how to navigate and use the system. This can be addressed by implementing tutorials or adding user guides such as tooltips and popups to help users create and generate a schedule.

Usability Concerns:

The dashboard and general interface need to be clear to help users with limited technical skills to use the application functionality and navigate freely.

3.4.3 Project Management Feasibility

The project must follow a structured approach such as the agile development approach. This will keep clear goals for each sprint and keeps time management successful. Complex issues may arise during development such as the CSPs, WebSocket's, and unexpected bugs and delays.

CSPs Development:

As the CSP is the main functional feature within the application and has many different complexities, issues may arise during development with inputting correct data and sending back optimal solutions. By starting with a basic CSP and implementing features one by one, rather than building a fully optimised CSP. This would reduce time in debugging and missing any important features that needed.

WebSocket's Integration:

WebSocket's will be implemented in specific routes. By testing each route before starting a new route can reduce debugging and build a stronger understanding of each component.

Unexpected Bugs and Delays:

Unexpected bugs and delays may be present during the development of the application. By leaving buffer time in each sprint will allow for any fixes or delays to be made up for, rather than cutting into the next sprints time.

3.4.4 Economic Feasibility

The cost analysis of the application would include the development, hosting and maintenance expenses. As the application uses open-source technologies, it helps reduce the initial expenses. However, Hosting the application will depend on the scalability. With the initial host, choosing free hosting providers would be beneficial and as the application grows choosing a flexible and scalable hosting providers could be implemented. The layout of the application currently keeps the economic feasibility positive as long as the project remains scaled.

3.5 Conclusion

This chapter has detailed the process of requirement analysis and specification for the development of the automated employee scheduling application. By evaluating industry-standard applications such as ConnectTeam and HomeBase, gathering primary data through surveys and interviews with local business owners, and developing detailed personas and use cases, the research has identified essential features and highlighted gaps in the market.

The resulting user, technical, functional, and non-functional requirements establish a structured foundation for system design, while the feasibility analysis assesses the project's viability and outlines potential challenges and mitigations. With this framework in place, the next phase will focus on system design and user experience enhancements, ensuring that the application is both robust and user centric.

4 Design

4.1 Introduction

This section is divided into two key areas: Program Design and User Interface Design. Each area addresses critical aspects of the development of this automated scheduling application. The main purpose of this section is to transform the requirements, which were outlined in the previous section (3.3). This process allows for the application to not only be technically efficient but also user-friendly.

4.2 Program Design

The program design section focuses on the technology stack chosen for the project and which technologies are considered. The backend and frontend file structures are chosen based on design pattern, which outlines benefits for the current design pattern. A more detailed diagram provides the application architecture, then dives deeper into the relationships within the database design. The section then concludes with the process design of how the application will communicate for certain tasks.

4.2.1 Technologies

This section includes a list of technologies used within the application. More information about each technology and how they interact with each other can be found within the technical requirements section above (3.3.3).

- **FastAPI** – Python framework for building APIs
- **React.js** – JavaScript library for building user interfaces
- **Vite** – Frontend build tool for fast development and optimized production builds
- **React Router** – Library for handling client-side routing in React applications
- **Shadcn** – Prebuilt UI components for React, based on Radix UI and Tailwind
- **React Big Calendar** – A React component for displaying and managing calendar events
- **MySQL** – Relational database management system
- **MySQL Workbench** – Database design and administration tool for MySQL

- **Insomnia** – API testing and development tool

Other technologies were considered but were not suited for various reasons. These technologies are listed below with reasoning why they were not chosen.

Express.js – While it is a popular backend framework, it was not chosen as I had recently developed a backend project using this technology and wanted to learn building an API using Python

OR-Tools (Google's Optimization Tools) – Considered for constraint satisfaction problems (CSP) as it has powerful and easy to use features with built in components, but it was found to be incompatible with FastAPI.

React Native – While useful for cross-platform mobile development, it was not chosen because the primary users of the application rely on laptops rather than mobile devices.

4.2.2 The Structures of the Backend and Frontend

4.2.2.1 Backend Structure

Figure 33 holds the visual representation of the folder structure for the FastAPI back-end, which uses MySQL as the database and SQL Alchemy as the ORM (Object-Relational Mapping) for database interactions. The project follows an MVC-inspired architecture, where the Models, CRUD operations (acting as Controllers), and Routes are clearly separated. Unlike traditional MVC, the Schemas define the data validation and serialization layer, and the "Views" are handled entirely by the React frontend.

The main components of the backend are:

- Schemas – Defines Pydantic models for request validation and response serialization.
- CRUD (Controllers) – Handles business logic and interacts with the database through SQL Alchemy models.
- Routes – Defines the FastAPI endpoints that interact with the CRUD layer.

- Models – Contains SQL Alchemy ORM models representing the database tables.
- Dependencies – Holds essential modules such as:
    - auth.py – Manages authentication and authorization.
    - db_config.py – Handles the database connection setup using SQL Alchemy.
- CSPs – A dedicated folder containing implementations and configurations for Constraint Satisfaction Problem solvers.
- Association.py – Defines the linking tables.
- Main.py – The entry point of the application, which initialises the app and includes all router imports.

Each folder includes an __init__.py file to ensure proper modularization, allowing smooth imports across the project.

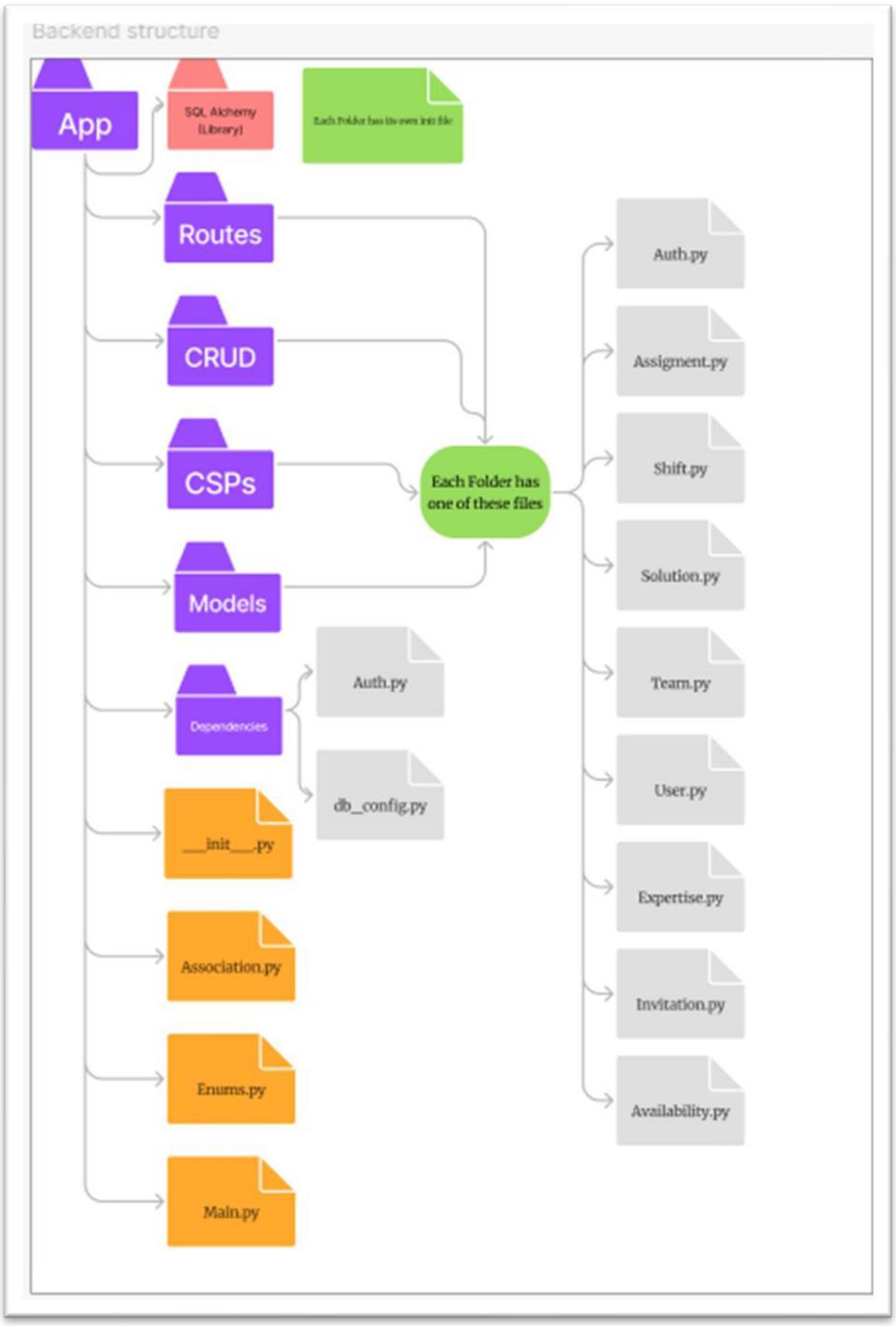


Figure 33, Backend File Structure

4.2.2.2 Frontend Structure

Figure 34 holds the visual representation of the React front-end, which was built with TypeScript and using modern tools such as Vite, React Router and Shadcn. The architecture of the frontend demonstrates the modularity and scalability.

The main components of the frontend are:

- Components – handles the global reusable UI elements across multiple pages, with subfolders within.
- Config – Holds the configuration files such as the API connection setup.
- Contexts – Different contexts which implement the global state management using Context API
- Hooks – Which contains the custom React hooks for sharing logic
- Pages – Which contains all the application view with sub folders for each model created and containing unique components.
- Types – Contains Typescript type definitions which are used across the app.
- Other Core files are placed in SRC folder
    - App.tsx – holds the client-side routing using React Router and wraps the routes in a layout.
    - App.css – Contains the global style sheet
    - Main.tsx – This is the entry point file of the application and mounts the react app to the DOM and wraps it with essential providers.

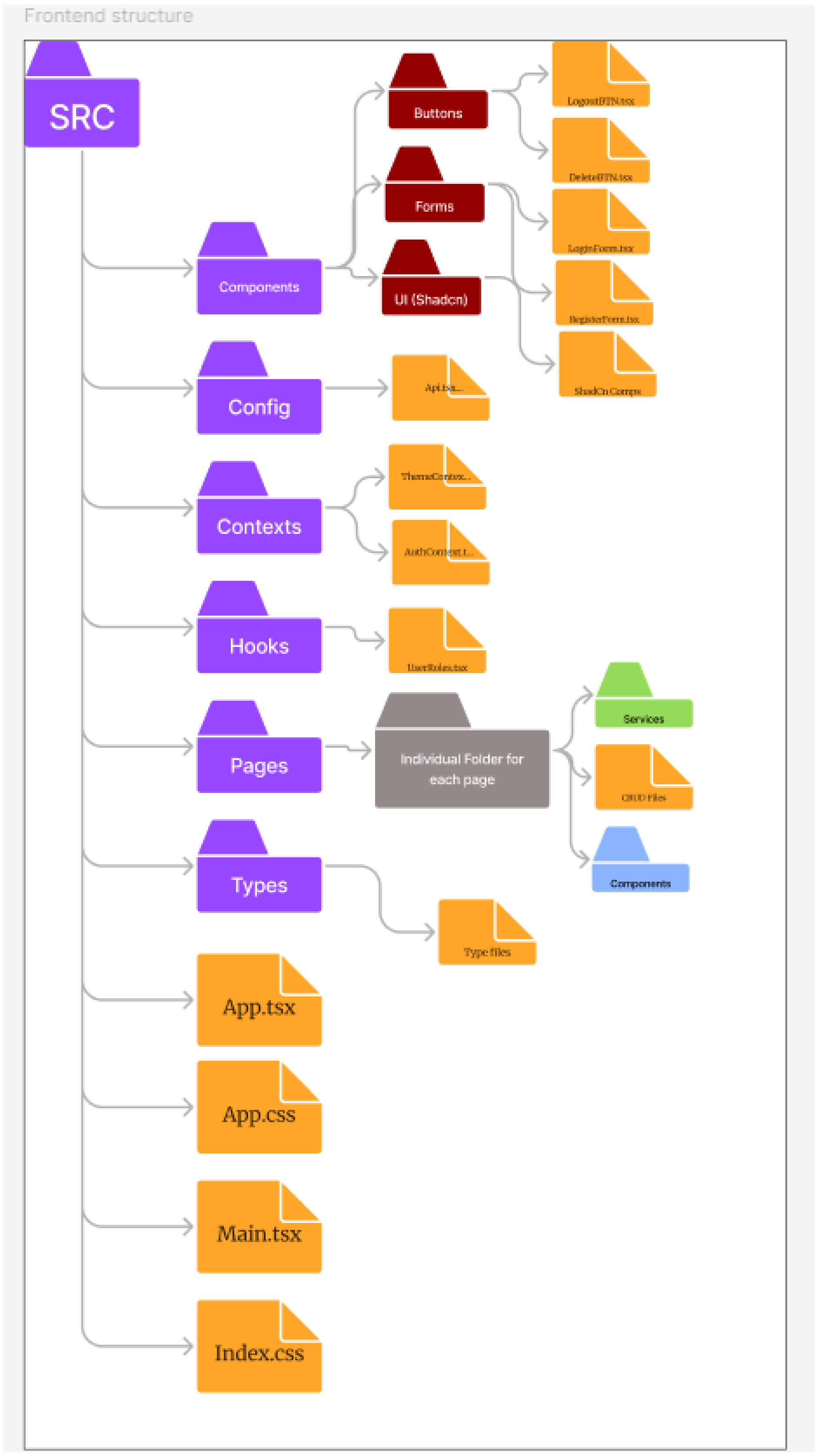


Figure 34, Frontend File Structure

4.2.3 Design Patterns

This project follows a Model-View-Controller (MVC) inspired architecture, which is very popular in modern web application development. While this project doesn’t follow the traditional MVC structure exactly, the core principles are still applied between the front end and back end.

Within the FastAPI backend, the Model and Controller are defined. The Model represents the database schema which uses SQL Alchemy ORM models. The Controller layer is identified as the CRUD layer in my application. The Controller would handle any business logic and the access to the database. The View is not directly handed in my backend, like generic MVC architecture. It gives the Frontend this responsibility.

Within the React frontend, the View structure is broken into separate concerns making it easier to read and manage, these concerns are components, pages, and state management. The controller layer which is passed is manage through state management such as context and hooks, which interact with the API endpoints. The components manage the structure of the how the data is being shown. The pages use the state management to request the data and pass the refined data to the components.

4.2.4 Application architecture Diagram

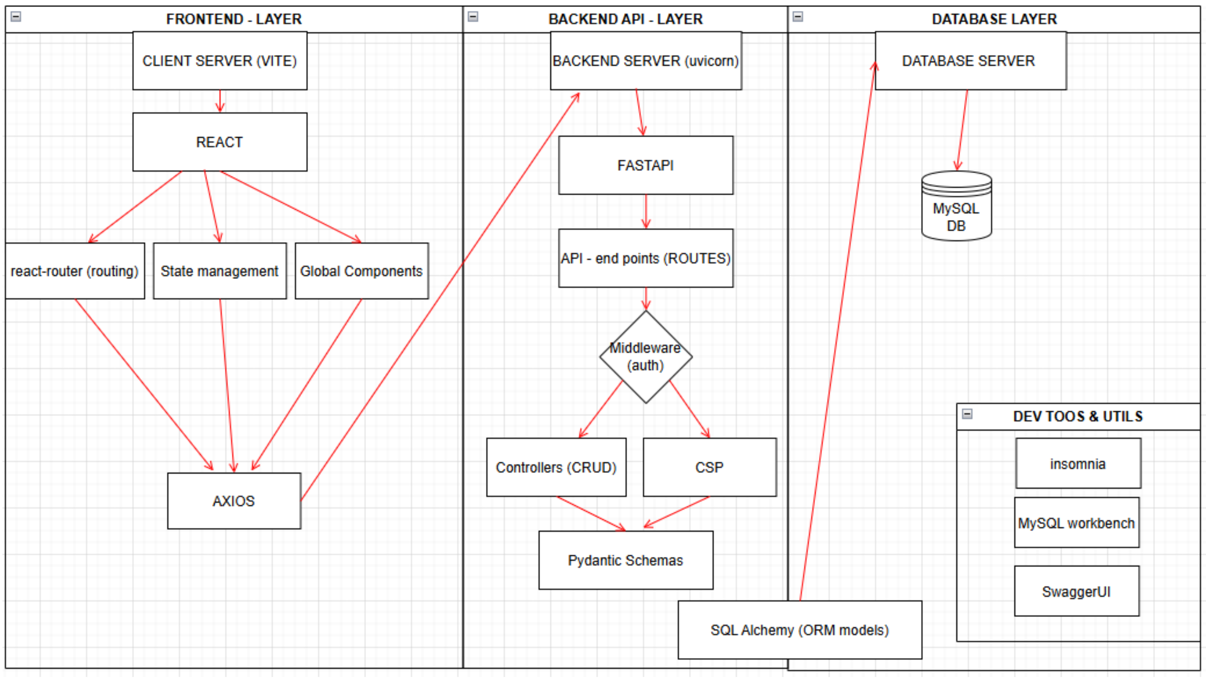


Figure 35, Application Architecture Diagram

4.2.5 Database design

This section includes a detailed overview of the database design. Figure 36 includes the Entity Relationship Diagram (ERD), which provides a general description of the relationships and tables with the application. Building upon this ERD, the Database Schema (DS) dives deeper into the breakdown of each entities columns, data types and the connectors between each entity which can be seen in Figure 37. Given the complexity of the applications relationships, it provides reasoning behind choosing a relational database like MySQL.

One of the central entities in this design is Team, which is intentionally connected to many other entities such as Users, Shifts, Schedules, and Availability. This reflects the real-world structure of the application, where most actions and associations occur within the context of a team. By anchoring many relationships to the Team entity, the application ensures that data remains scoped and organized logically, enabling features like role-based access, team-specific scheduling, and filtered views based on team affiliation.

4.2.5.1 ERD

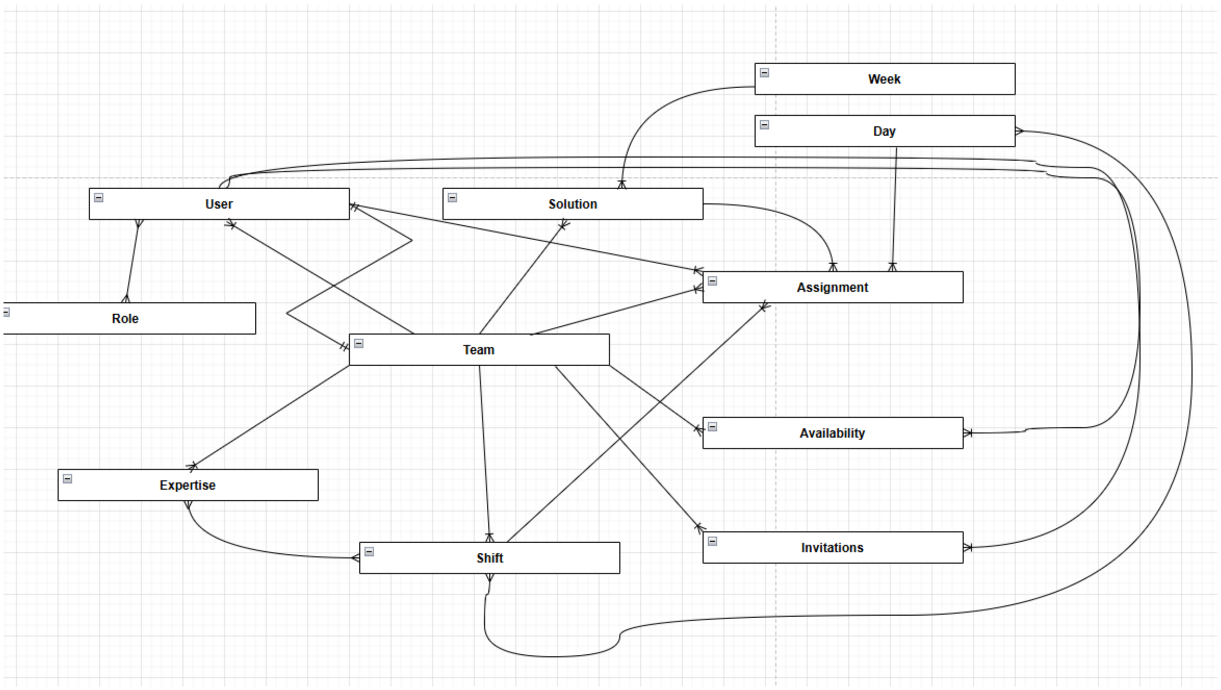


Figure 36, Entity Relationship Diagram

Relationships

Team:

- Team has **Many** Assignments
- Team has **Many** Solutions
- Team has **Many** Shifts
- Team has **Many** Expertise's
- Team has **Many** Users
- Team has **One** owner (User)
- Team has **Many** Availabilities
- Team has **Many** Invitations

User:

- User has **Many** Roles
- User has **One** Team
- A Creator (User) has **One** Team

- User has **Many** Invitations
- User has **Many** Availabilities
- User has **Many** Assignments

Shift:

- Shift has **Many** Days
- Shift has **Many** Assignments
- Shift has **Many** Expertise’s
- A Shift has **One** Team

Role:

- Role has **Many** Users

Expertise:

- Expertise has **One** Team
- Expertise has **Many** Shifts

Invitation:

- Invitation has **One** User
- Invitation has **One** Team

Availability:

- Availability has **One** Team
- Availability has **One** User

Assignment:

- Assignment has **One** Shift
- Assignment has **One** Team
- Assignment has **One** User
- Assignment has **One** Day
- Assignment has **One** Solution

Solution:

- Solution has **One** Week
- Solution has **One** Team
- Solution has **Many** Assignments

Day:

- Day has **Many** Assignments
- Day has **Many** Shifts

Week:

- Week has **Many** Solutions

4.2.5.2 Database Schema

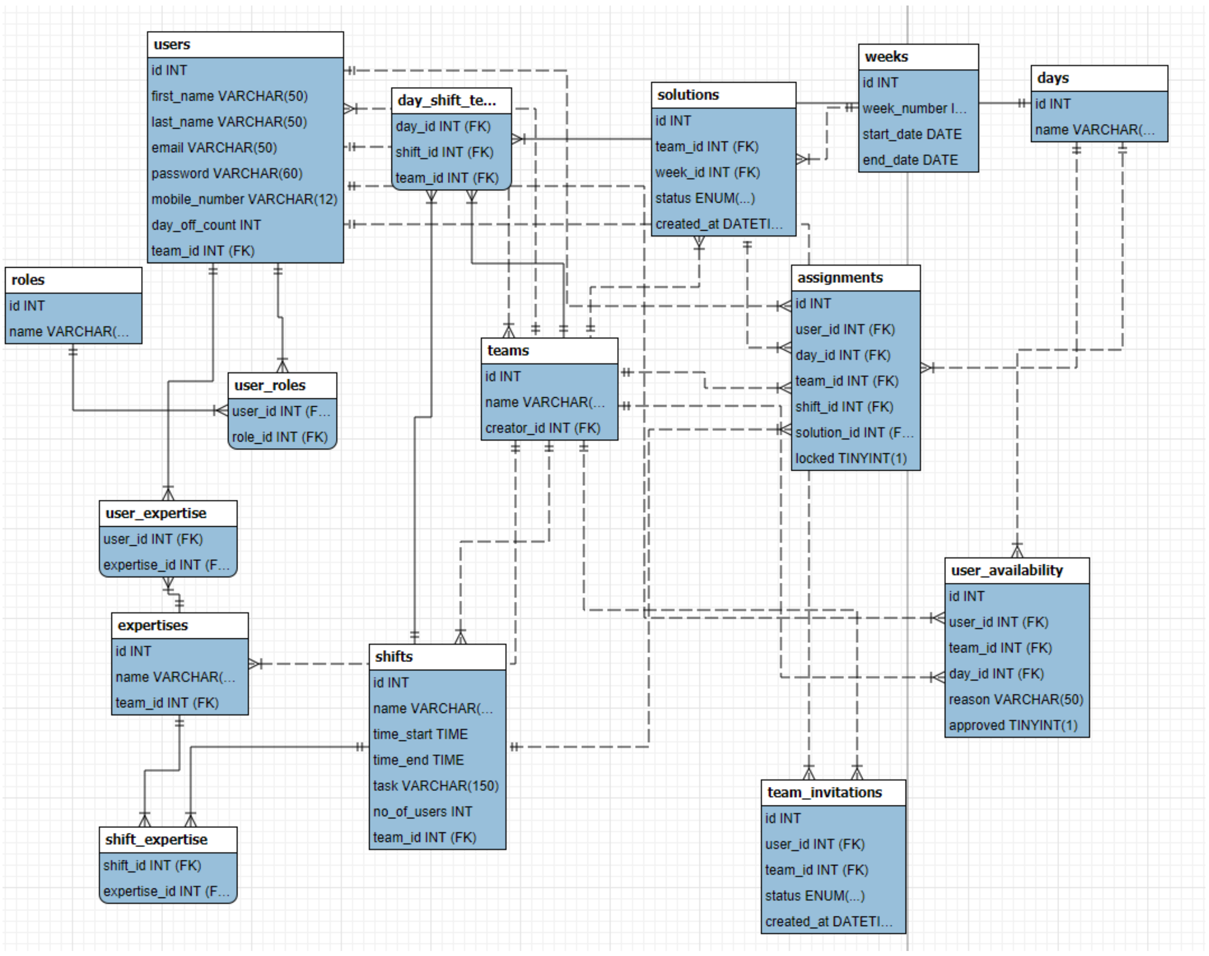


Figure 37, Database Schema

4.2.6 Process design

This section provides diagramming techniques such as sequence diagrams, to show the interaction between components in specific tasks, and flowcharts to outline different decision trees and workflows within the application.

Sequence Diagrams 1: Login/ Authentication

Figure 38 provides the sequence of events which occur once a previous user attempts to login.

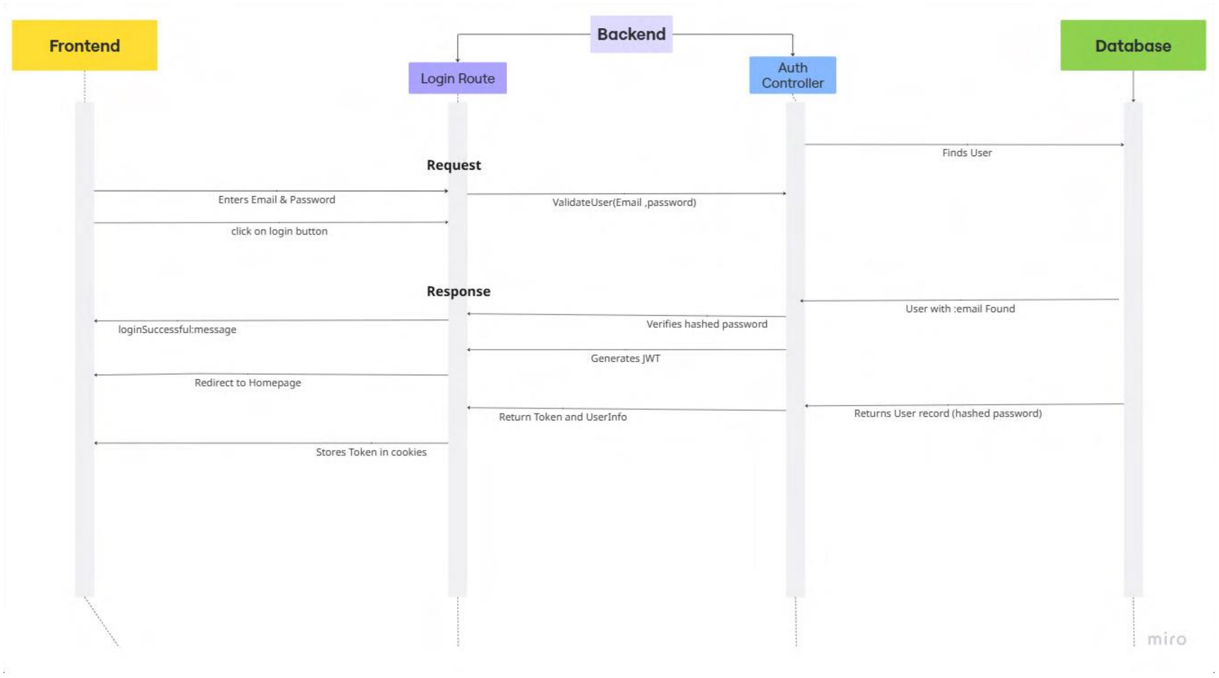


Figure 38, Login Sequence Diagram

Sequence Diagrams 2: CSP Solver being Triggered

Figure 39 provides the sequence of events which occur once the generate button is pressed.

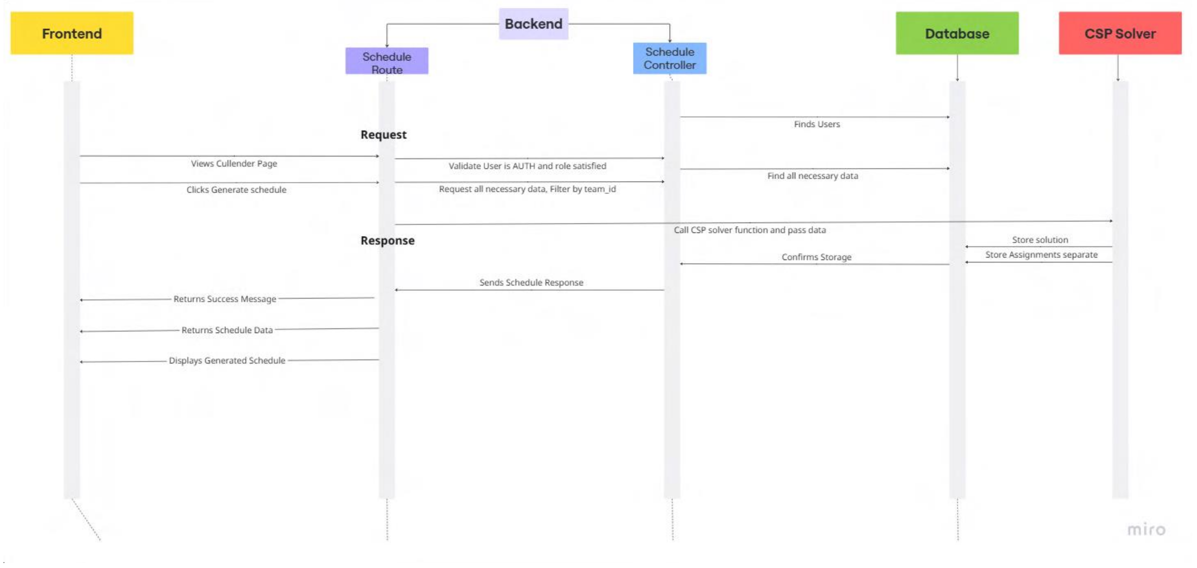
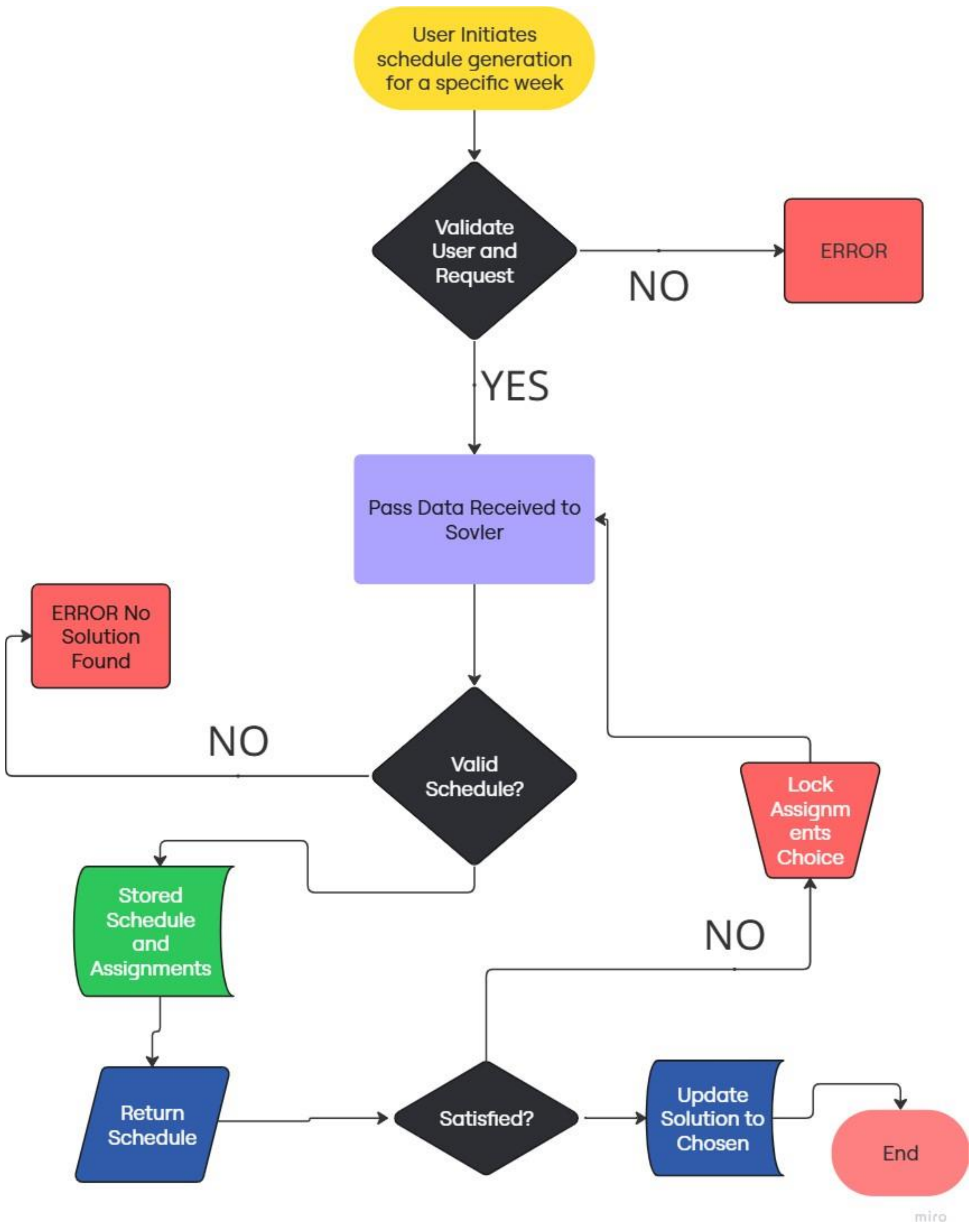


Figure 39, CSP solver being triggered Sequence Diagram

Flowcharts 1: CSP Flow

Figure 40 illustrates the process an employer follows when creating a schedule for an upcoming week.



User Initiates schedule generation for a specific week
Validate User and Request
NO
ERROR
YES
Pass Data Received to Sovler
ERROR No Solution Found
NO
Valid Schedule?
Lock Assignm ents Choice
Stored Schedule and Assignments
NO
Return Schedule
Satisfied?
Update Solution to Chosen
End
miro

Figure 40, CSP Flow

Flowcharts 2: Role-Based Access Control

Figure 41 illustrates the process a user with a specific role follows to navigate the system.

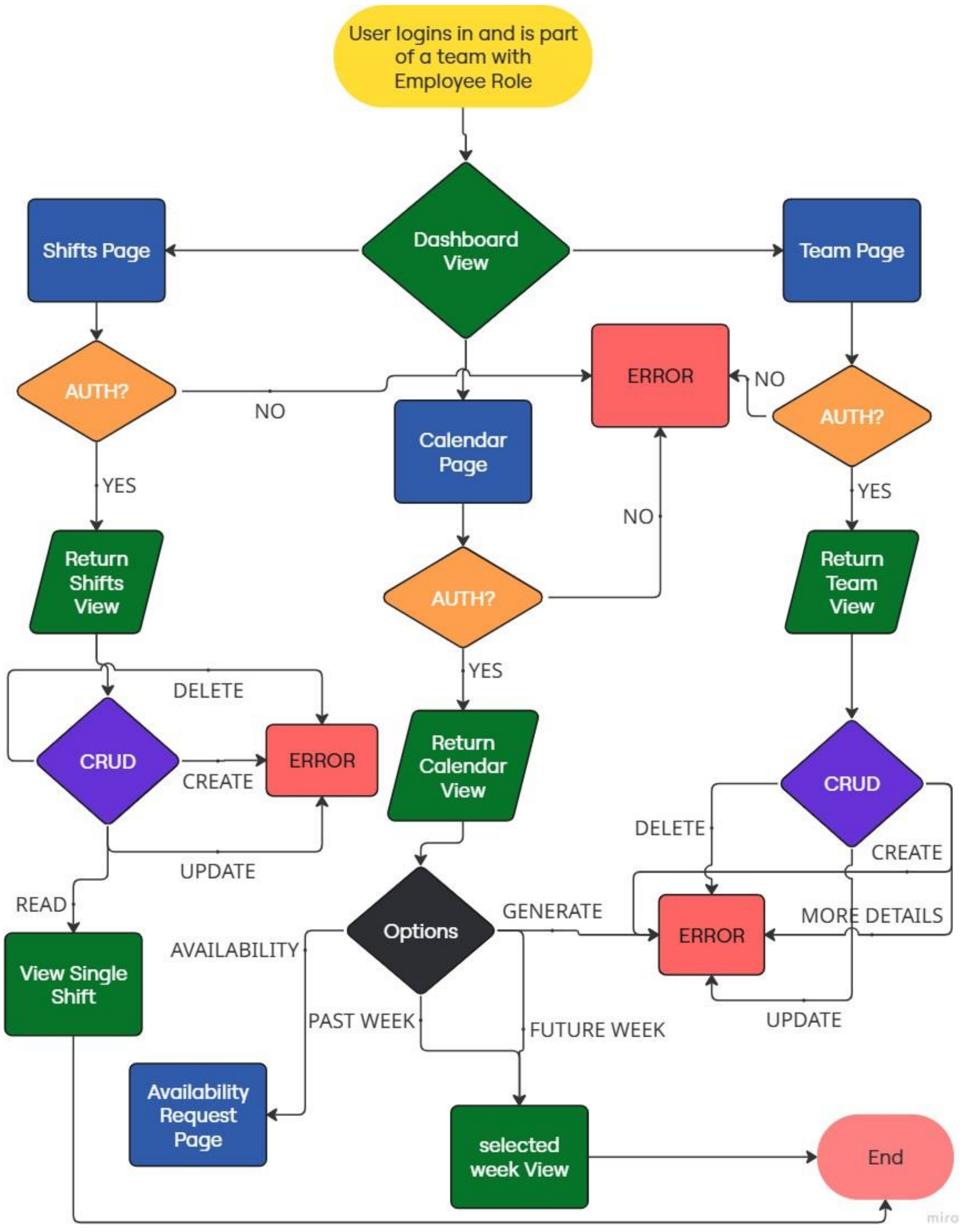


Figure 41, Role-Based Access Control Flow

4.3 User interface design

This section describes the progression of the applications user interface design. The UI design had iterative process, beginning with building low-fidelity wireframes within Figma. These wireframes mapped out the core functionality and layout of the key pages. The low-fidelity wireframes had soon evolved into high-fidelity wireframes. The structure of the wireframes allowed for visualising the user flow within the application. Once the user flow was fluid and met the non-functional requirements (3.3.5), the style guide was chosen in relation to Shadcn typography and colours, which was shortly implemented to the finished wireframes.

4.3.1 Wireframe

The low-fidelity wireframes acted as blueprint and evolved over time. Once the blueprints were implemented, the process of building high-fidelity wireframes using prebuilt components from Shadcn Figma had begun. This allowed to pass consistent patterns across the application and add a cleaner and more modern look.

The primary focus of the UI was the calendar page, which would use the interactive prebuilt calendar, React-Big-Callender. Building this was tricky as there was no template available within Figma. I created each component separately and created variants for each component to change in different flows or iterations.

The Figures below ranging from Figure 42 to Figure 45, shows the development of the wireframes beginning with the low fidelity to high-fidelity. This provides the iterative process of the user-interface design.

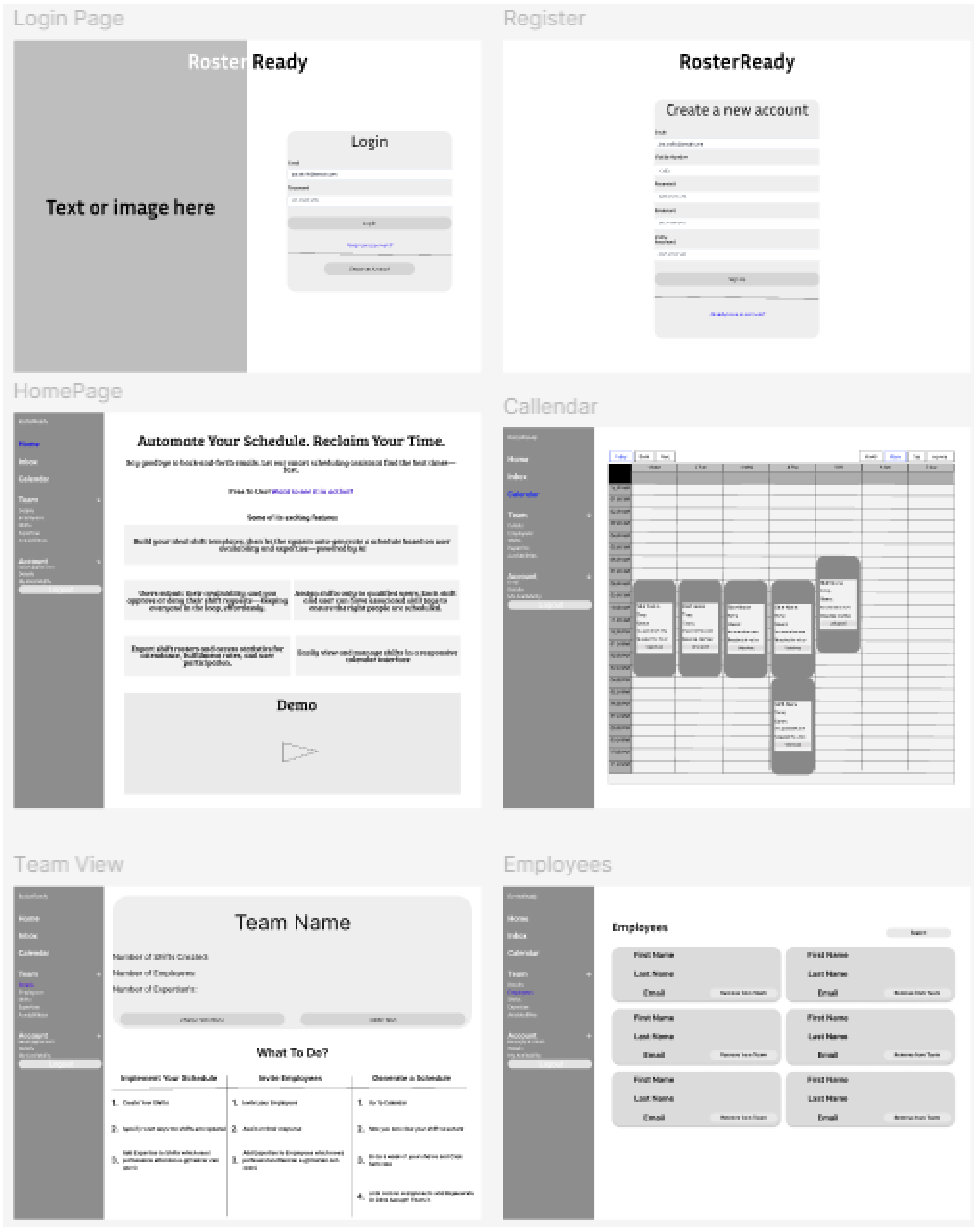


Figure 42, Chosen Wireframes 1

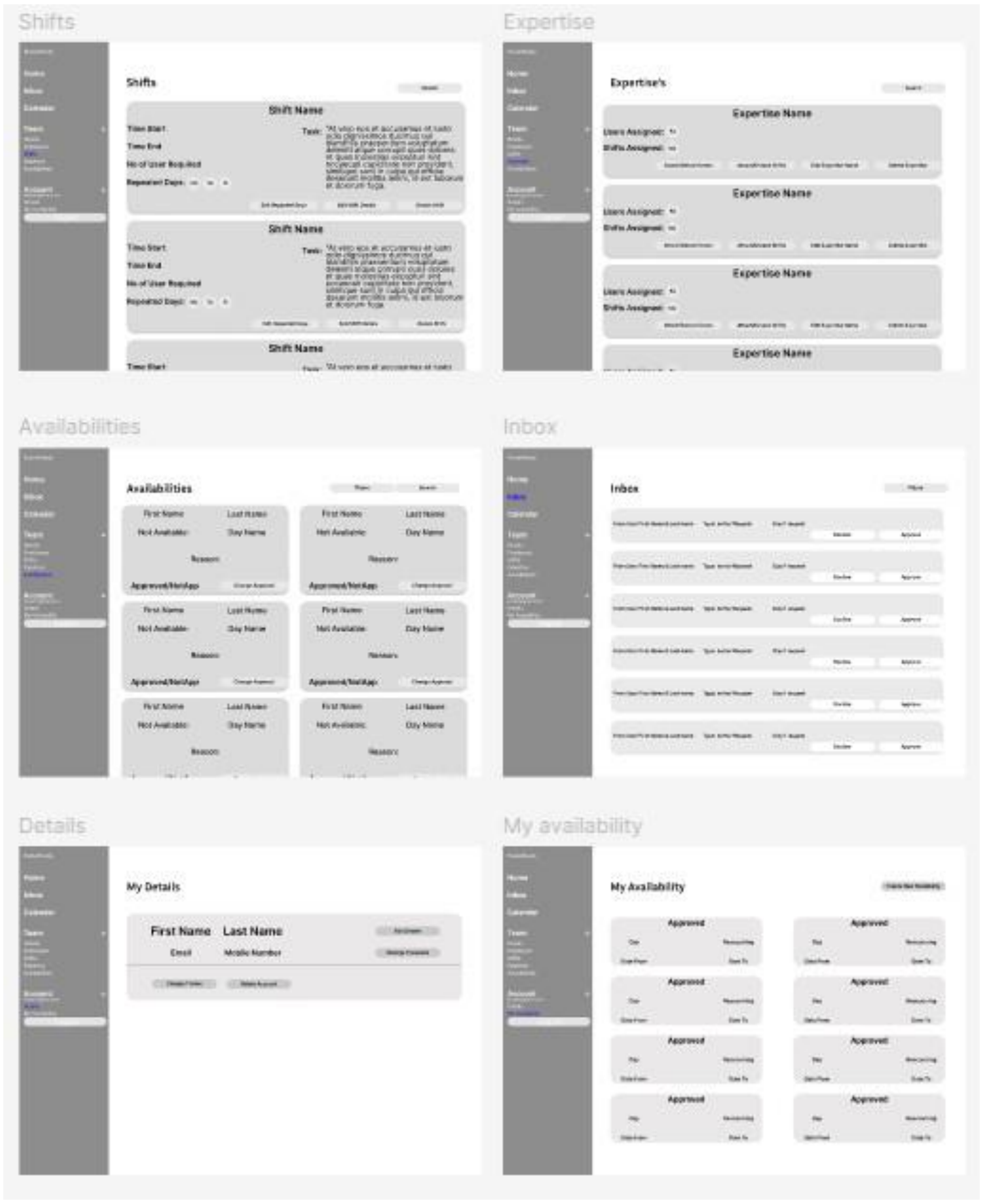


Figure 43, Chosen Wireframes 2

Figure 44, Iterations of Month Callendar

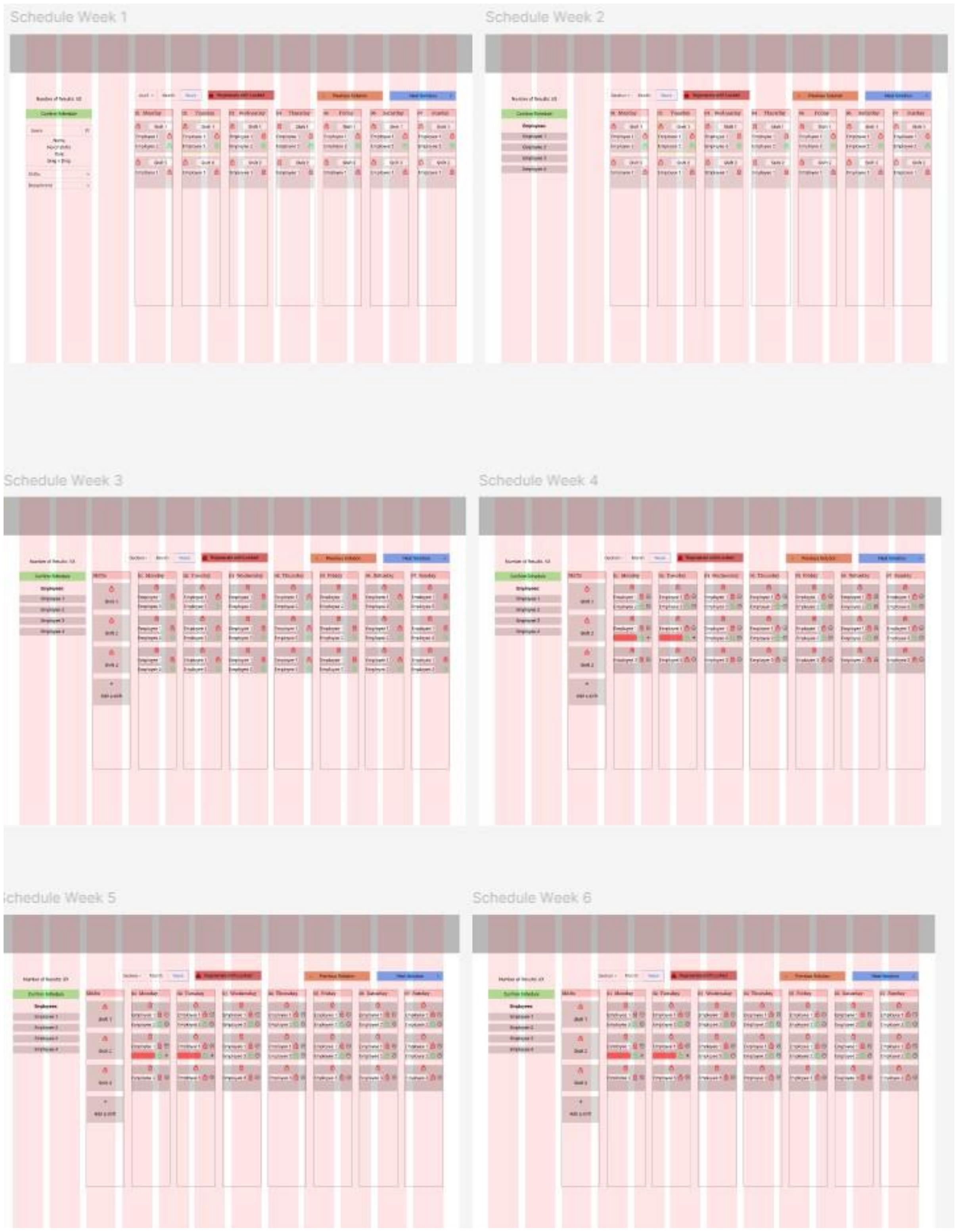


Figure 45, Iterations of Week Callendar

4.3.2 User Flow Diagram

This section shows the flow of navigation for a popular task within the page. This task would be viewing the schedule for the following week. The flow is visualised for both roles, Employee and Admin. This allows to get a deeper understanding of the difference of flow between each role type. In relation to the Figures below, each user has the same navigation process, but they become restricted to certain features and aspects due to their role.

Employer:

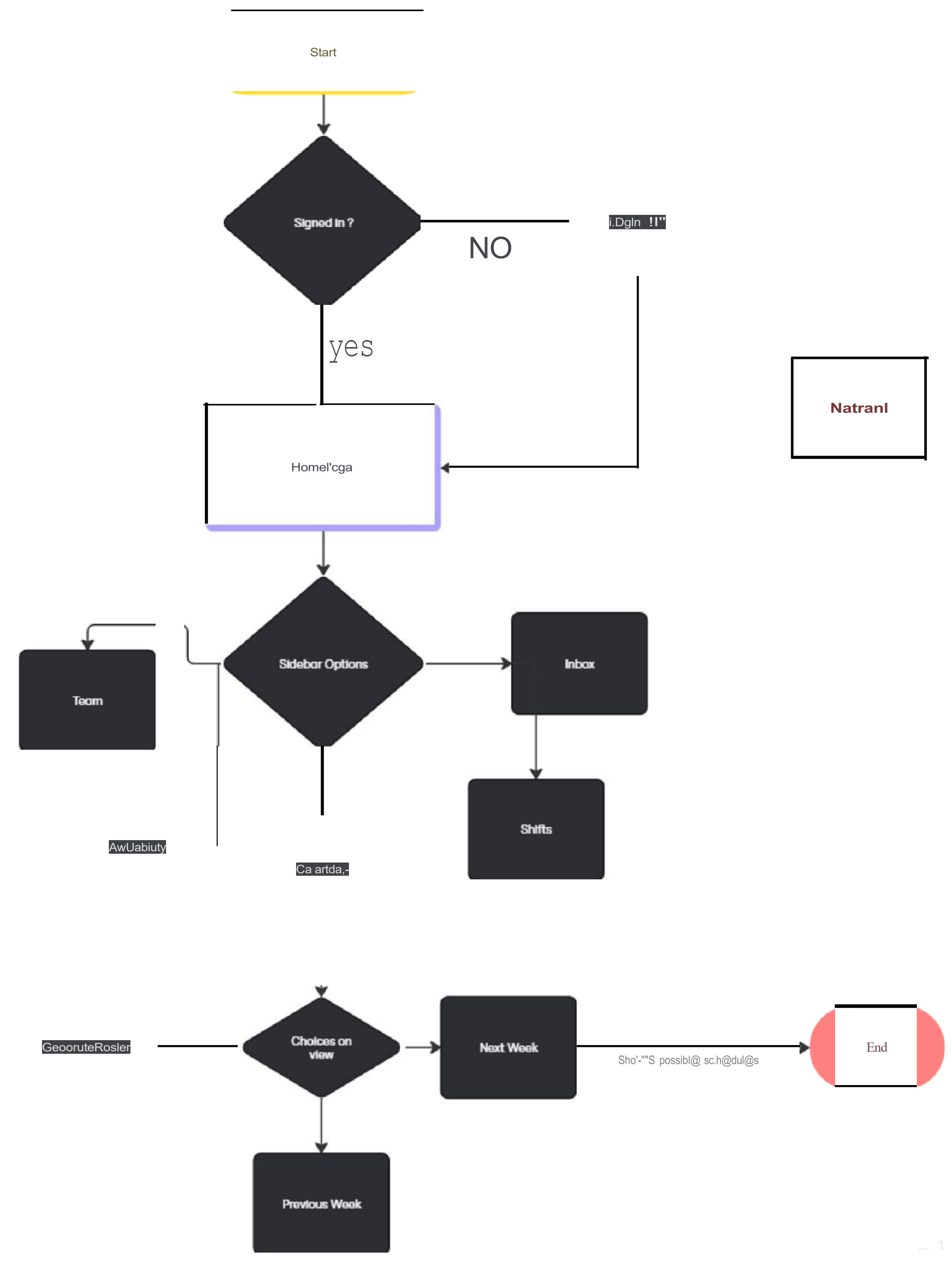


Figure 46, User Flow 1

Employee:

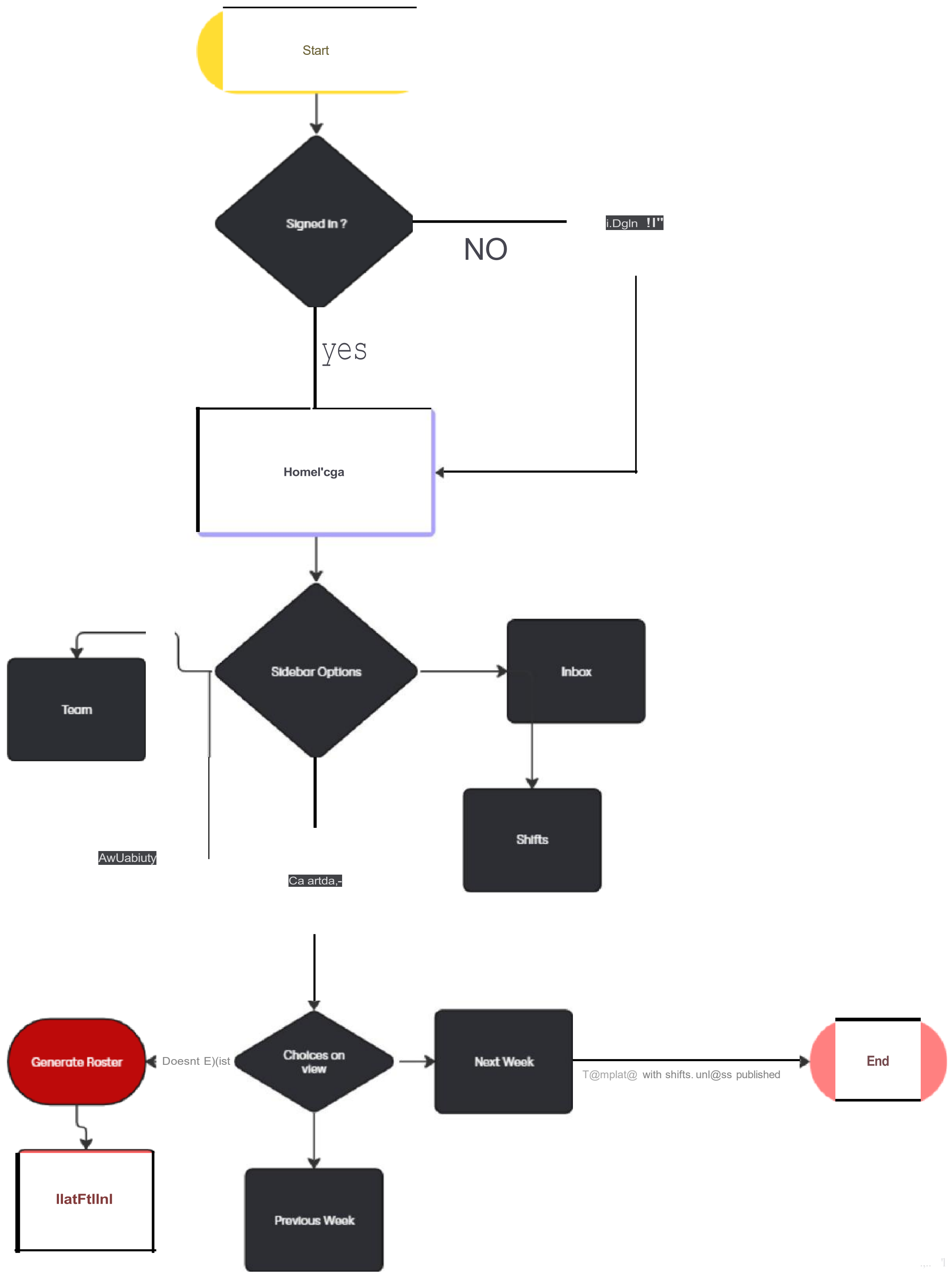


Figure 47, User Flow 2

4.3.3 Style guide

The style guide focuses on the selection of a colour palette and typography for the application. The colour choices were influenced by the palettes of similar applications, ConnectTeam and HomeBase (3.2.1), which used vibrant colour types of blues or purples. Alternatively, a green palette was selected to symbolize success and growth, contributing to a positive user experience. The typography was chosen to provide readability and a modern look. The typography and colour palette selection can be seen in the Figure 48 below.

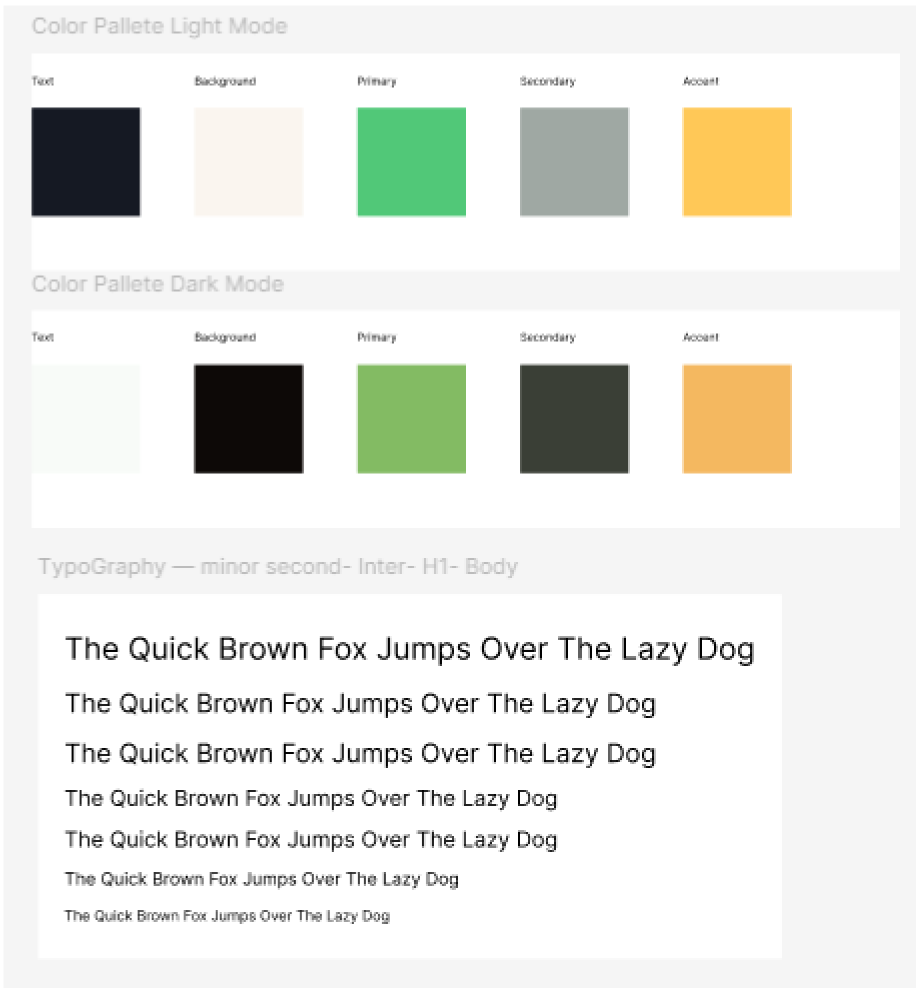

Color Pallete Light Mode
Text
Background
Primary
Secondary
Accent
Color Pallete Dark Mode
Text
Background
Primary
Secondary
Accent
TypoGraphy — minor second- Inter- H1- Body
The Quick Brown Fox Jumps Over The Lazy Dog
The Quick Brown Fox Jumps Over The Lazy Dog
The Quick Brown Fox Jumps Over The Lazy Dog
The Quick Brown Fox Jumps Over The Lazy Dog
The Quick Brown Fox Jumps Over The Lazy Dog
The Quick Brown Fox Jumps Over The Lazy Dog
The Quick Brown Fox Jumps Over The Lazy Dog

Figure 48, Style Guide

4.4 Conclusion

The design phase of the project's development is crucial, this section outlined how the requirements of the application will transform to build a successful, reliable and user-friendly automated scheduling application. By carefully selecting the combination of modern technologies such as FastAPI, React and MySQL, the application shown potential in creating a successful application. The adoption of the MVC-inspired architecture allowed for a vision of structured and organised folder structure. The database design had provided proof for the choice of database, due to the number of relationships, especially with the core entity.

The program design was crucial especially due to the complexity of the backend and main focuses around the CSP. Understanding the complexity of the program design, it provided a guided approach to the user interface (UI) design. By providing different levels of fidelity, it showed the iterations of the UI development. The completion of the wireframes allowed for a clear view of how a specific user would navigate throughout this automated scheduling application. By adding in vibrant colours into the colour palette and choosing a clear and modern typography, the application now had some structure to match the requirements needed. This visual of both, program design and UI design, gave a clear vision of how this automated scheduling application will work and how it should look during implementation phase.

5 Implementation

5.1 Introduction

The Implementation section details the development environment, highlighting the Integrated Development Environment (IDE) used, which was essential for managing and maintaining the implementation process. The section dives deeper into the implementation of the system architecture, which consisted of four core components, database, back-end, Constraint Satisfaction Problem (CSP) solver, and front-end. Each subsection provides a detailed description of the steps taken to match the requirements.

The projects implementation phase was developed using Scrum methodology, which provided a flexible and structured ideology to manage the development across nine unique sprints. Each sprint included goals and the deliverables to ensure continuous progress for both the front end and back end.

5.2 Development environment

The structure of this application was carried out in a structured and efficient environment to ensure smooth and reliable progress. The primary Integrated Development Environment (IDE) was Visual Studio Code (VS Code) for the development of this automated scheduling application. VS Code was lightweight and allowed for support of both Python and JavaScript. This IDE also provided various extensions for formatting, management and error checking, which ensure smooth and clear development. The terminal within VS Code was used for frequently running commands for FastAPI, React and Git.

Git was used to handle Version control, with the project hosted on GitHub for backups and version history. Git played a huge role in the implementation of this application as it allowed for commits with messaging, document changes or issues at the end of each day. Git also has issue tracking which keeps track of bugs, which is useful for future development. ". gitignore" file removed any files or folders which including any sensitive data to being passed to GitHub, such as the .env file which stored all the environment variables.

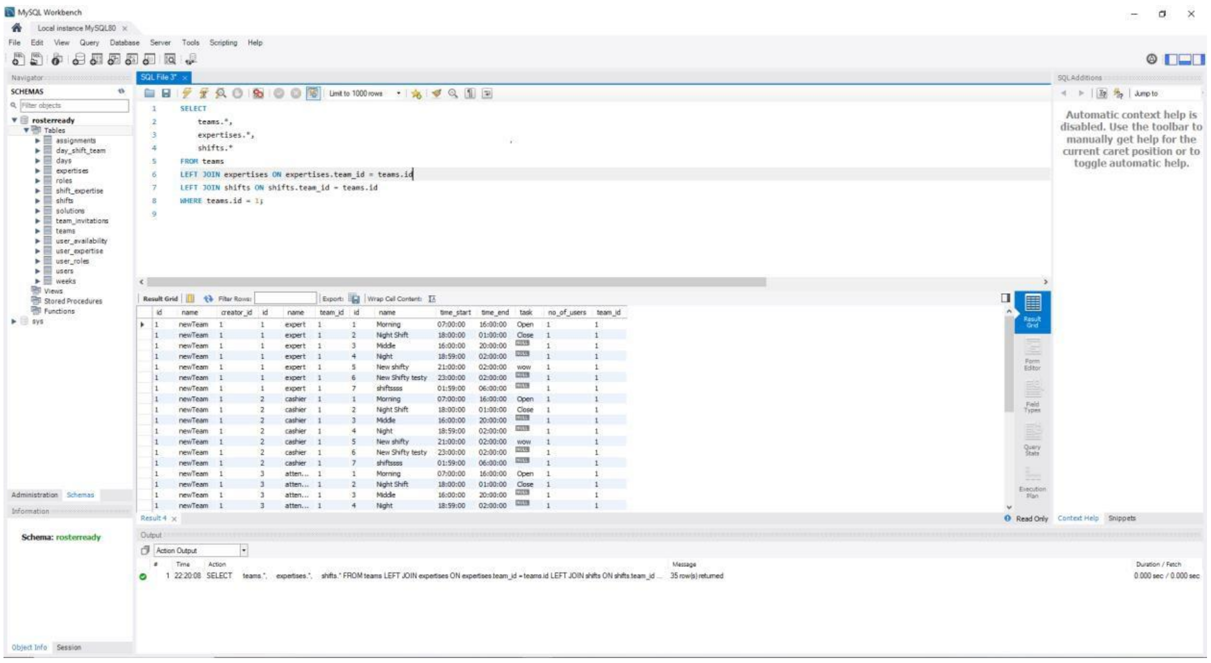
To host the development of the backend, FastAPI was used in combination with Uvicorn, an ASGI server that allowed the application to run locally with high performance and asynchronously. FastAPI also provided an interactive API documentation interface through Swagger UI, which was extremely useful for testing and exploring the defined routes during development. The frontend was hosted using Vite, a fast build tool and development server. Vite enabled hot reloading, allowing for instant updates during the development of the frontend.

In addition, Insomnia was used for managing and testing API requests. Insomnia helped streamline communication between frontend and backend by allowing repeated testing of endpoints with different payloads and headers. Routes were organised into folders based on functionality, making it easier to maintain a clean and logical structure throughout the development process.

5.3 Database

MySQL, an open-source relational database management system (RDBMS) was used as my database in my automated scheduling application. The database stored and managed all the data, which provided a structured data storage using tables and utilised in SQL queries for data manipulation and retrieval. MySQL came with a powerful graphical interface, MySQL Workbench. Workbench provided tools for creating and visualising the database schema, running SQL queries and managing tables and relationships.

In a broader view, a relation database model was chosen for this application due to its structure and ability to handle complex relationships between entities. This model was suited for the needs of this automated scheduling application, where multiple entities such as users, schedules, availability, shifts, and expertise needed to be interconnected. In relation to the database schema at Figure 37, the team served as a core entity, linking to most other tables through foreign key relationships. This allowed for a simplified data retrieval of related information. Specifically, when executing the Constraint Satisfaction Problem Solver (CSPs) to retrieve all relevant data associated with a specific team. Passing the correct data quickly was essential as this was the main focus around the automated scheduling application.



Automatic context help is disabled. Use the toolbar to manually get help for the current caret position or to toggle automatic help.

```sql
SELECT
    teams.*,
    expertises.*,
    shifts.*
FROM teams
LEFT JOIN expertises ON expertises.team_id = teams.id
LEFT JOIN shifts ON shifts.team_id = teams.id
WHERE teams.id = 1;
```

id	name	creator_id	id	name	team_id	id	name	time_start	time_end	task	no_of_users	team_id
1	newTeam	1	1	expert	1	1	Morning	07:00:00	16:00:00	Open	1	1
1	newTeam	1	1	expert	1	2	Night Shift	18:00:00	01:00:00	Close	1	1
1	newTeam	1	1	expert	1	3	Middle	16:00:00	20:00:00	NULL	1	1
1	newTeam	1	1	expert	1	4	Night	18:59:00	02:00:00	NULL	1	1
1	newTeam	1	1	expert	1	5	New shifty	21:00:00	02:00:00	wow	1	1
1	newTeam	1	1	expert	1	6	New Shifty testy	23:00:00	02:00:00	NULL	1	1
1	newTeam	1	1	expert	1	7	shiftssss	01:59:00	06:00:00	NULL	1	1
1	newTeam	1	2	cashier	1	1	Morning	07:00:00	16:00:00	Open	1	1
1	newTeam	1	2	cashier	1	2	Night Shift	18:00:00	01:00:00	Close	1	1
1	newTeam	1	2	cashier	1	3	Middle	16:00:00	20:00:00	NULL	1	1
1	newTeam	1	2	cashier	1	4	Night	18:59:00	02:00:00	NULL	1	1
1	newTeam	1	2	cashier	1	5	New shifty	21:00:00	02:00:00	wow	1	1
1	newTeam	1	2	cashier	1	6	New Shifty testy	23:00:00	02:00:00	NULL	1	1
1	newTeam	1	2	cashier	1	7	shiftssss	01:59:00	06:00:00	NULL	1	1
1	newTeam	1	3	atten...	1	1	Morning	07:00:00	16:00:00	Open	1	1
1	newTeam	1	3	atten...	1	2	Night Shift	18:00:00	01:00:00	Close	1	1
1	newTeam	1	3	atten...	1	3	Middle	16:00:00	20:00:00	NULL	1	1
1	newTeam	1	3	atten...	1	4	Night	18:59:00	02:00:00	NULL	1	1

```
#  Time      Action                                                                                                                  Message              Duration / Fetch
1  22:20:08  SELECT  teams.*,  expertises.*,  shifts.* FROM teams LEFT JOIN expertises ON expertises.team_id = teams.id LEFT JOIN shifts ON shifts.team_id ...  35 row(s) returned  0.000 sec / 0.000 sec
```

Figure 49, MySQL workbench

5.4 Backend

5.4.1 Overview

In relation to the program design section (4.2), the backend was developed based on the chosen architecture, which incorporates an MVC-inspired pattern. The backend was built using the FastAPI Python framework, selected for its asynchronous performance and high-speed capabilities, allowing the system to efficiently handle multiple requests concurrently.

The backend leverages SQL Alchemy for easy interaction with the MySQL database. The combination of FastAPI and SQL Alchemy enabled the creation of RESTful APIs that the frontend can interact with. The backend structure was designed and developed to be modular, consisting of models, CRUD operations (controller), routes, and schemas.

5.4.2 Virtual Environment Setup and Dependency Management

After the initial setup of FastAPI, the projects virtual environment needed to be setup to keep the dependencies isolated to avoid conflicts with global environments from other projects. A virtual environment was created using Venv. Using Venv it isolated the dependencies in a clean and controlled environment.

To create a virtual environment, this command was used within Bash terminal.

```
Python -m venv env
```

After Running this command, it would create a folder which stores the dependencies. Before installing any dependencies, the activation of the virtual environment is essential.

```
./env/Scripts/activate
```

Once the activation was made the necessary dependencies can now be installed, such as FastAPI, SQL alchemy and unicorn, which was done through pip installations. The successful activation of the virtual environment can be seen when the terminal prompt is slightly changed depending on the terminal. In terms of Bash, it presents two stars at the start of the prompt.

The application required dependencies could now be managed by freezing the dependencies installed into a requirements.txt. Allowing for future installation on a new device based on the requirements.txt file.

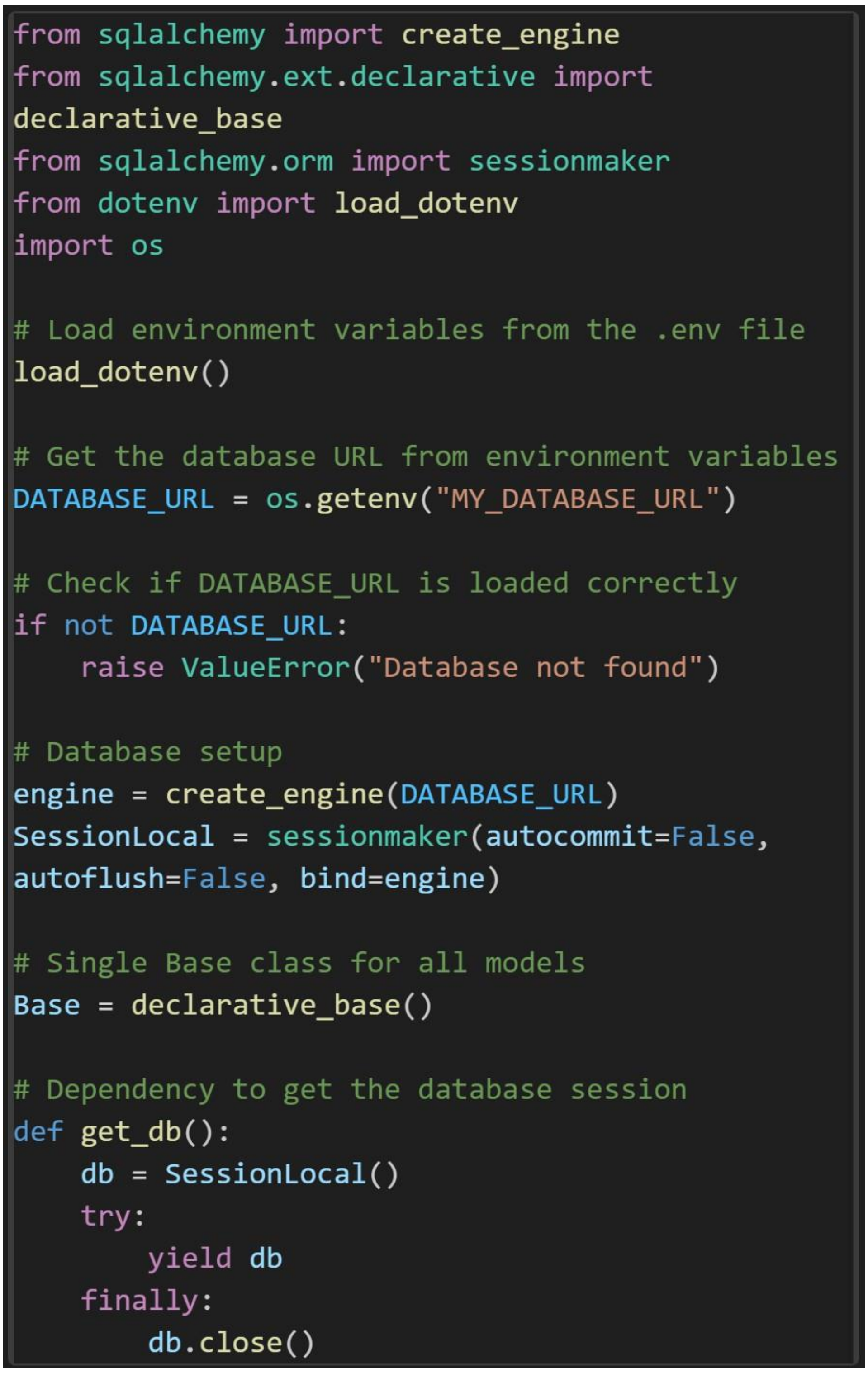
```
pip freeze > requirements.txt
```

5.4.3 Database Configuration

To configure the database for this application, SQL Alchemy was used as the ORM tool to manage the interactions with the database. To establish the connection to the database, db_config.py file was created. Concepts from SQL Alchemy were imported to create this functioning database file.

- Create engine function concept was to define the connection to the database. The engine serves as the interface essentially for interacting with the database, managing connections and executing SQL queries.
- Declarative base function is the base class that all the other models will inherit. These models will represent the tables in my database.

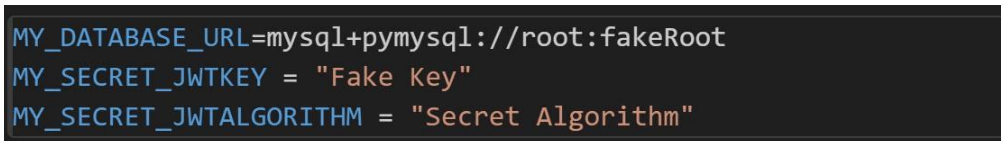
- The session maker function is used to create a session factory, which then can create an argument between the engine and the session, which associates the engine with the session.



```python
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import
declarative_base
from sqlalchemy.orm import sessionmaker
from dotenv import load_dotenv
import os

# Load environment variables from the .env file
load_dotenv()

# Get the database URL from environment variables
DATABASE_URL = os.getenv("MY_DATABASE_URL")

# Check if DATABASE_URL is loaded correctly
if not DATABASE_URL:
    raise ValueError("Database not found")

# Database setup
engine = create_engine(DATABASE_URL)
SessionLocal = sessionmaker(autocommit=False,
autoflush=False, bind=engine)

# Single Base class for all models
Base = declarative_base()

# Dependency to get the database session
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

Figure 50, Database Config

From the Figure above we can see that database URL is being passed through dotenv, using a `python-dotenv` library. This was to hide any sensitive information from the public, especially once the application is hosted. Creating a `.env` file and applying this file in the gitignore, provided that this file would not be push on git and displayed on GitHub. The figure Below shows a copy of how this `.env` file is viewed.

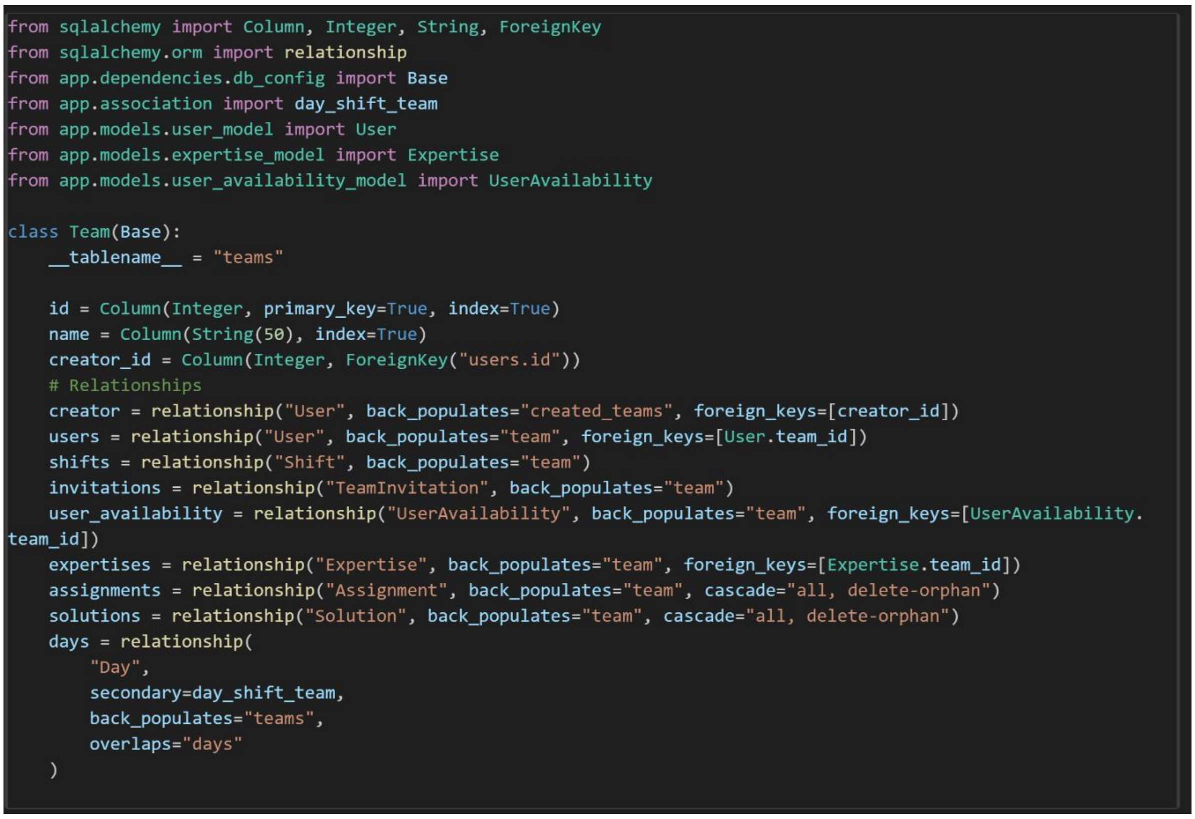


```
MY_DATABASE_URL=mysql+pymysql://root:fakeRoot
MY_SECRET_JWTKEY = "Fake Key"
MY_SECRET_JWTALGORITHM = "Secret Algorithm"
```

Figure 51, ENV example

5.4.4 Database Models and Relationships

Database models were created using SQL Alchemy to match the structure and relationships of the database schema and pass it through to the MySQL database. Each model corresponds to a table within the database, relationships within the model is how the model can interact with each other.



```
from sqlalchemy import Column, Integer, String, ForeignKey
from sqlalchemy.orm import relationship
from app.dependencies.db_config import Base
from app.association import day_shift_team
from app.models.user_model import User
from app.models.expertise_model import Expertise
from app.models.user_availability_model import UserAvailability

class Team(Base):
    __tablename__ = "teams"

    id = Column(Integer, primary_key=True, index=True)
    name = Column(String(50), index=True)
    creator_id = Column(Integer, ForeignKey("users.id"))
    # Relationships
    creator = relationship("User", back_populates="created_teams", foreign_keys=[creator_id])
    users = relationship("User", back_populates="team", foreign_keys=[User.team_id])
    shifts = relationship("Shift", back_populates="team")
    invitations = relationship("TeamInvitation", back_populates="team")
    user_availability = relationship("UserAvailability", back_populates="team", foreign_keys=[UserAvailability.
team_id])
    expertises = relationship("Expertise", back_populates="team", foreign_keys=[Expertise.team_id])
    assignments = relationship("Assignment", back_populates="team", cascade="all, delete-orphan")
    solutions = relationship("Solution", back_populates="team", cascade="all, delete-orphan")
    days = relationship(
        "Day",
        secondary=day_shift_team,
        back_populates="teams",
        overlaps="days"
    )
```

Figure 52, Team Model

The figure above shows the Team model, which represents the team’s table. This model holds all necessary columns that must be passed to the table and all the relationships that need to be declared based on the database schema. Each model which is created, inherits from Base, which maps the model class to its corresponding database table. Models that define relationships must import both Foreign Key and relationship from SQL Alchemy.

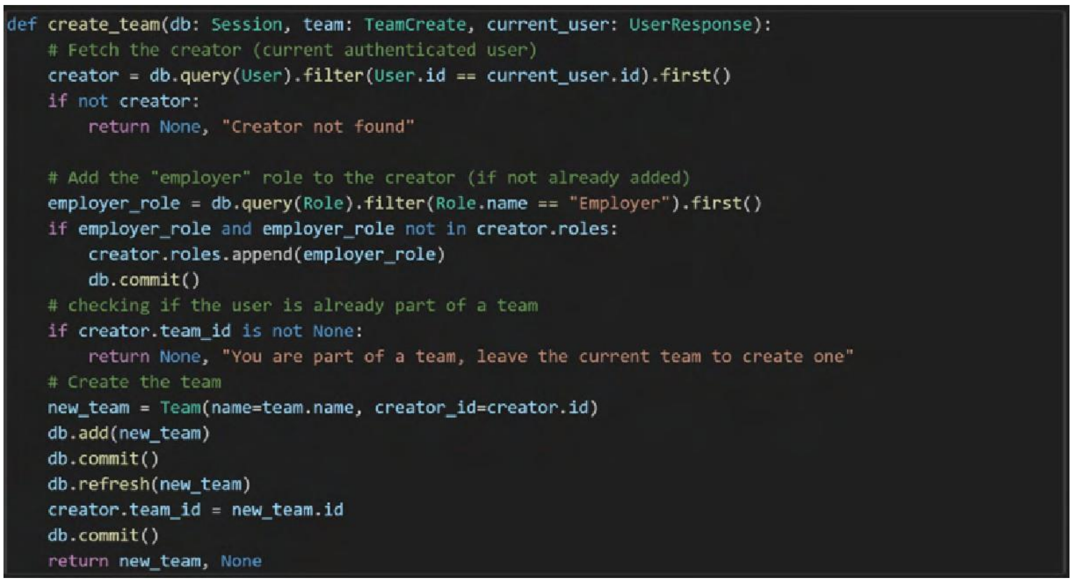
The relationship function typically takes in two or more parameters. The first is the model’s name as a string that it's linking to, and the second is `back_populates`, which sets up a two-way connection with the specified model. It can also take optional arguments like cascade, which defines how changes on a parent object affect related objects, and secondary, which specifies the linking table used for many-to-many relationships. Tables involved in one-to-one (1:1) or one-to-many (1:M) relationships, act as the parent, typically include a Foreign Key column to associate themselves with another table. In the Team model's case, the `creator_id` represents a one-to-one relationship with a User.

5.4.5 CRUD Operations and Business Logic

This section explains how the CRUD (Create, Read, Update, Delete) requests and business logic was implemented. Each model which was created had corresponding set of CRUD requests. Each of these requests was defined as a function, which took multiple parameters. A reoccurring necessary parameter, db with a type annotation of Session which was provided by SQL alchemy. Db allows for access to the database session to interact with the database. The other parameters would be data we are passing through, e.g. id, or data we are receiving, e.g. Specific Team. Each of these parameters had types to explicitly say what data is expected.

The business logic took place within these functions, which would vary depending on the table and request type. This logic helps maintain data consistency and enforcing the intended permissions. The Figure below shows the create team function.

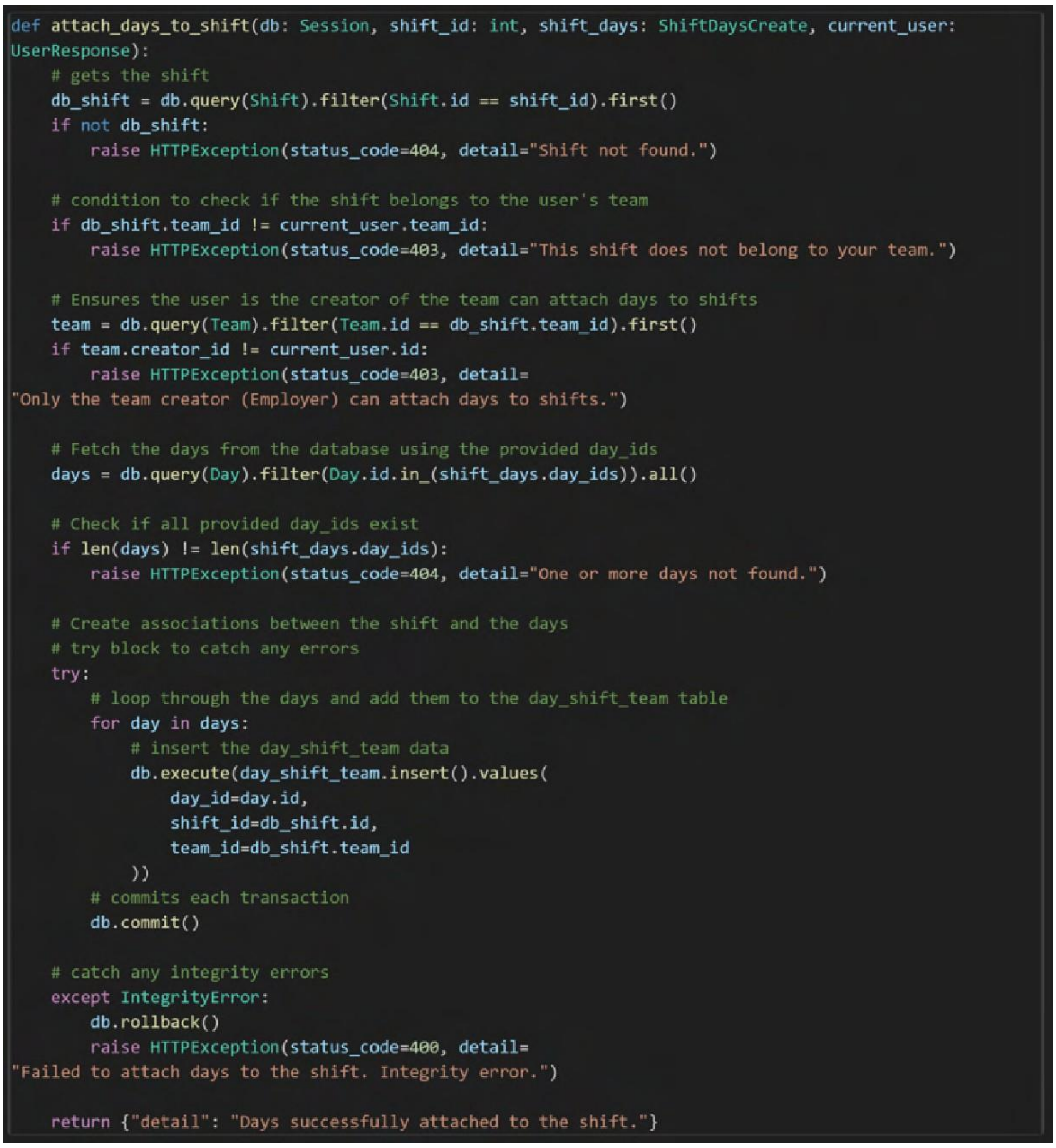
This function handled the creation of new teams into the database. The logic began at line 3, initialising a new variable by querying the database to filter through the user table to match the id to the current users id. If the id was not found in the database from the current user, it would provide an error message “Creator not found”. The second part of the logic was to find the role name under employer, which then had a condition to check if the name exists and if the user didn’t already have this role. The third condition was to double check that the user was not already a creator of another team. If all the conditions passed it would create a new team instance passing the team's name provided and the id of the current user. This was then added and committed to the database, which then got the created teams id and passed it to the user, which was then stored in the foreign key under `team_id`. This new team was then returned with errors specified as none.



```python
def create_team(db: Session, team: TeamCreate, current_user: UserResponse):
    # Fetch the creator (current authenticated user)
    creator = db.query(User).filter(User.id == current_user.id).first()
    if not creator:
        return None, "Creator not found"

    # Add the "employer" role to the creator (if not already added)
    employer_role = db.query(Role).filter(Role.name == "Employer").first()
    if employer_role and employer_role not in creator.roles:
        creator.roles.append(employer_role)
        db.commit()
    # checking if the user is already part of a team
    if creator.team_id is not None:
        return None, "You are part of a team, leave the current team to create one"
    # Create the team
    new_team = Team(name=team.name, creator_id=creator.id)
    db.add(new_team)
    db.commit()
    db.refresh(new_team)
    creator.team_id = new_team.id
    db.commit()
    return new_team, None
```

Figure 53, Create Team Function

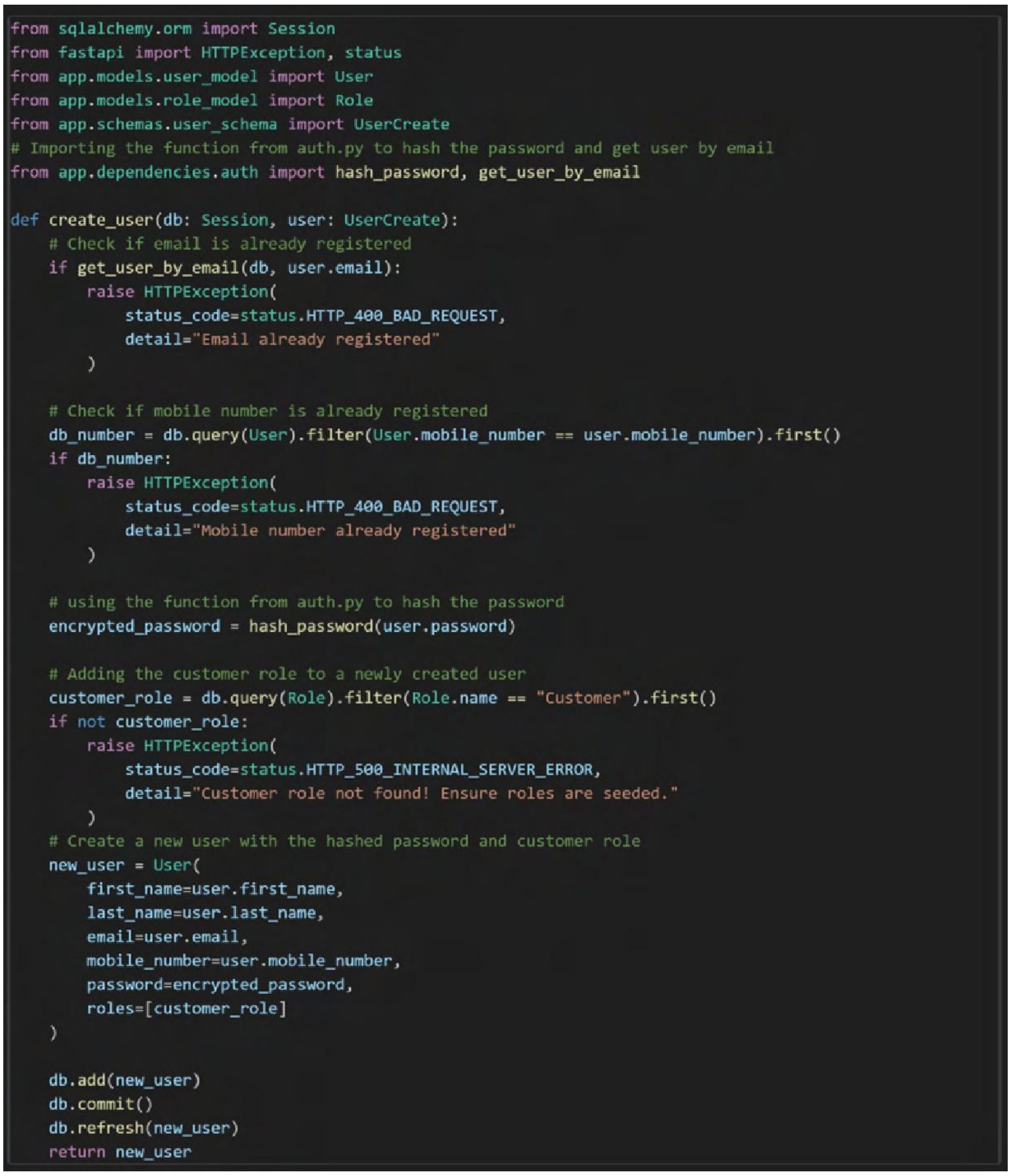
These functions became more complex when dealing with linking tables. The `attach_days_to_shift` function within `shift_crud.py` file, required multiple conditions and loops. The figure below displays the various conditions to check if the shift exists, if it belongs to the user's team, and if the creator is the current user trying to attach. Once all these user dependent conditions pass, the days variable will be initialised by querying the Day table and getting all the ids which match to the request. If at least one day id doesn't exist it will throw an error. Using a try-except statement to handle any errors that may occur during the execution. It loops through each day and inserting the value of the `team_id`, `shift_id` and the day_id foreach day present. Creating X number of tables dependent of X length of days.



```python
def attach_days_to_shift(db: Session, shift_id: int, shift_days: ShiftDaysCreate, current_user:
UserResponse):
    # gets the shift
    db_shift = db.query(Shift).filter(Shift.id == shift_id).first()
    if not db_shift:
        raise HTTPException(status_code=404, detail="Shift not found.")

    # condition to check if the shift belongs to the user's team
    if db_shift.team_id != current_user.team_id:
        raise HTTPException(status_code=403, detail="This shift does not belong to your team.")

    # Ensures the user is the creator of the team can attach days to shifts
    team = db.query(Team).filter(Team.id == db_shift.team_id).first()
    if team.creator_id != current_user.id:
        raise HTTPException(status_code=403, detail=
"Only the team creator (Employer) can attach days to shifts.")

    # Fetch the days from the database using the provided day_ids
    days = db.query(Day).filter(Day.id.in_(shift_days.day_ids)).all()

    # Check if all provided day_ids exist
    if len(days) != len(shift_days.day_ids):
        raise HTTPException(status_code=404, detail="One or more days not found.")

    # Create associations between the shift and the days
    # try block to catch any errors
    try:
        # loop through the days and add them to the day_shift_team table
        for day in days:
            # insert the day_shift_team data
            db.execute(day_shift_team.insert().values(
                day_id=day.id,
                shift_id=db_shift.id,
                team_id=db_shift.team_id
            ))
        # commits each transaction
        db.commit()

    # catch any integrity errors
    except IntegrityError:
        db.rollback()
        raise HTTPException(status_code=400, detail=
"Failed to attach days to the shift. Integrity error.")

    return {"detail": "Days successfully attached to the shift."}
```

Figure 54, Attach Days Function

Another unique function, `create_user` found within `user_crud.py` file is displayed in the figure below. This function has several conditions and unique logic which attaches a specific role to a new user, Customer. The function uses an imported function which was created within `auth.py`, the purpose of this function is to encrypt the password given by the new user to ensure password is not exposed in plain text in the database. This encryption function will be seen in Authentication and Access Control section (5.5.10).



```python
from sqlalchemy.orm import Session
from fastapi import HTTPException, status
from app.models.user_model import User
from app.models.role_model import Role
from app.schemas.user_schema import UserCreate
# Importing the function from auth.py to hash the password and get user by email
from app.dependencies.auth import hash_password, get_user_by_email

def create_user(db: Session, user: UserCreate):
    # Check if email is already registered
    if get_user_by_email(db, user.email):
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail="Email already registered"
        )

    # Check if mobile number is already registered
    db_number = db.query(User).filter(User.mobile_number == user.mobile_number).first()
    if db_number:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail="Mobile number already registered"
        )

    # using the function from auth.py to hash the password
    encrypted_password = hash_password(user.password)

    # Adding the customer role to a newly created user
    customer_role = db.query(Role).filter(Role.name == "Customer").first()
    if not customer_role:
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="Customer role not found! Ensure roles are seeded."
        )
    # Create a new user with the hashed password and customer role
    new_user = User(
        first_name=user.first_name,
        last_name=user.last_name,
        email=user.email,
        mobile_number=user.mobile_number,
        password=encrypted_password,
        roles=[customer_role]
    )

    db.add(new_user)
    db.commit()
    db.refresh(new_user)
    return new_user
```

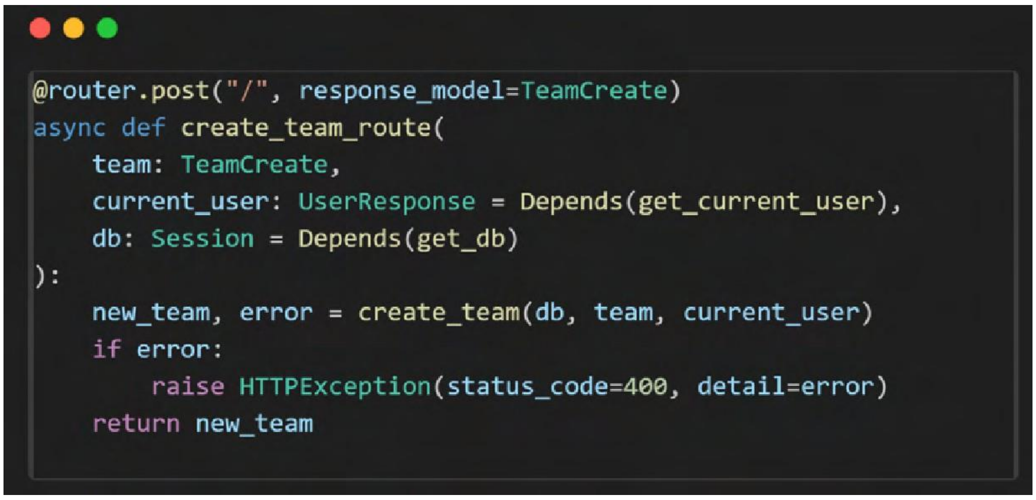
Figure 55, Create User Function

5.4.6 Routes and API Endpoints

This section explains how the routes were created, and the corresponding API endpoints were defined in the main.py file. In each route the API Router function would be called to group a set of routes, allowing the application to be modular. The route would handle HTTP requests using FastAPI `@router.post` decorator to register the function under the current router.

The decorator would be followed with two sets of parameters, the relative route path, and the response model, which tells FastAPI to return and validate the response in a structured format. Each route is defined using async to allow asynchronous execution, with included parameters which need to be passed to the CRUD function afterwards. These parameters specify the schema for the request or response body and use dependency injections to call specific functions in the background using `Depends` and stores them in a variable. Dependency injections wait for the current function to be called before executing the route. These variables are passed to the CRUD function with all the variables needed, which then the return from the CRUD function is stored in variable to return.

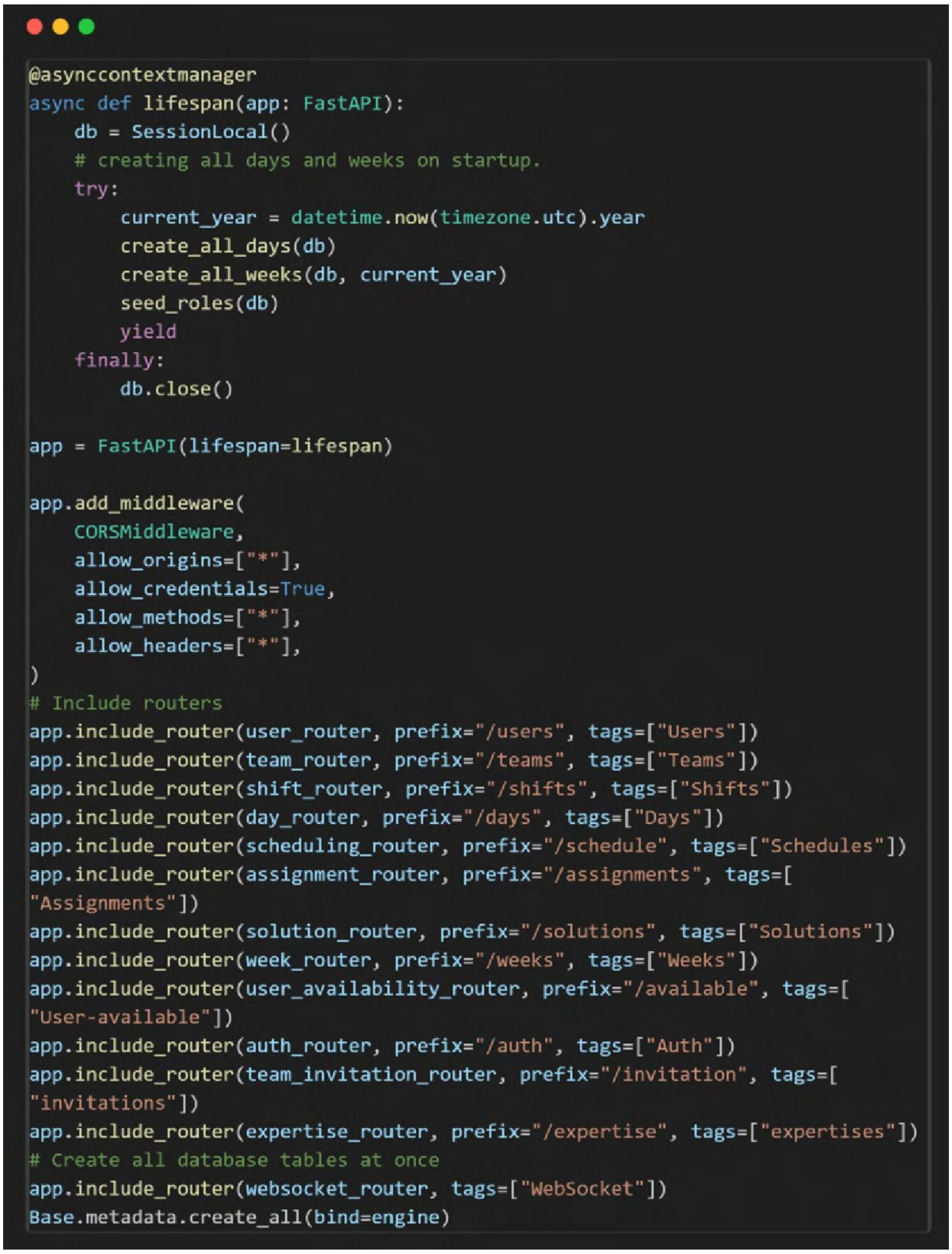
In the Figure below, an example of the team creation route is shown. This route handles a POST request, which is set to the URL of "/" and uses the Team Response schema as the response model. The function `create_team_route` accepts three parameters, the team, which is of type Team Create, the current user, which is of type User Response and uses FastAPI dependency injection to get the current user through a function. Db, the most important parameter, which is of type Session and injects the `get_db` function. These variables are passed to the `create_team` function, where the team creation logic is handled. The response from the function is then stored in `new_team` variable which is then returned in the route.



```
@router.post("/", response_model=TeamCreate)
async def create_team_route(
    team: TeamCreate,
    current_user: UserResponse = Depends(get_current_user),
    db: Session = Depends(get_db)
):
    new_team, error = create_team(db, team, current_user)
    if error:
        raise HTTPException(status_code=400, detail=error)
    return new_team
```

Figure 56, Create Team Route

The `main.py` file acts as the central entry point for the application, which is responsible for managing application lifecycle events, configuring middleware, initialising the FastAPI instance and including all the route modules under a specific prefix.



```
@asynccontextmanager
async def lifespan(app: FastAPI):
    db = SessionLocal()
    # creating all days and weeks on startup.
    try:
        current_year = datetime.now(timezone.utc).year
        create_all_days(db)
        create_all_weeks(db, current_year)
        seed_roles(db)
        yield
    finally:
        db.close()

app = FastAPI(lifespan=lifespan)

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
# Include routers
app.include_router(user_router, prefix="/users", tags=["Users"])
app.include_router(team_router, prefix="/teams", tags=["Teams"])
app.include_router(shift_router, prefix="/shifts", tags=["Shifts"])
app.include_router(day_router, prefix="/days", tags=["Days"])
app.include_router(scheduling_router, prefix="/schedule", tags=["Schedules"])
app.include_router(assignment_router, prefix="/assignments", tags=[
"Assignments"])
app.include_router(solution_router, prefix="/solutions", tags=["Solutions"])
app.include_router(week_router, prefix="/weeks", tags=["Weeks"])
app.include_router(user_availability_router, prefix="/available", tags=[
"User-available"])
app.include_router(auth_router, prefix="/auth", tags=["Auth"])
app.include_router(team_invitation_router, prefix="/invitation", tags=[
"invitations"])
app.include_router(expertise_router, prefix="/expertise", tags=["expertises"])
# Create all database tables at once
app.include_router(websocket_router, tags=["WebSocket"])
Base.metadata.create_all(bind=engine)
```

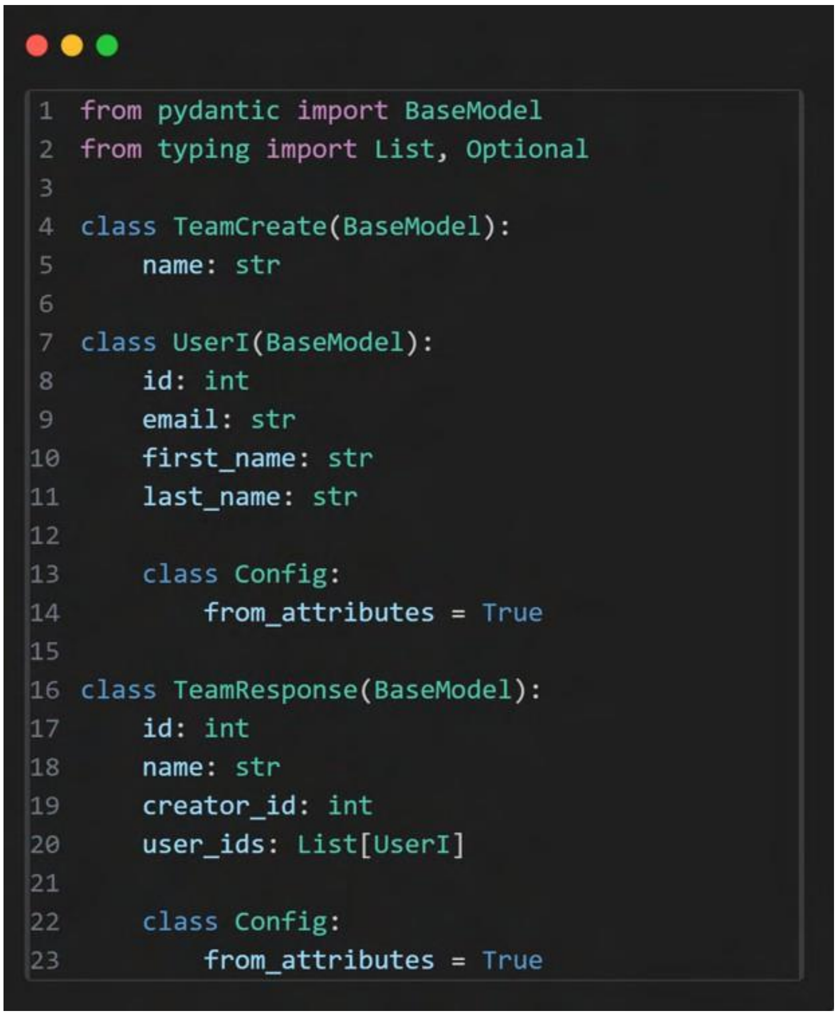
Figure 57, Central Entry Point

In relation to the figure below, all the route files are imported and included using app. `include_router` by passing the specific router and the prefix I want to use. Each route also includes a tag which was to categories the routes in the swagger UI

5.4.7 Schema Design with Pydantic

This section focuses on how Pydantic was used for data validation and serialisation. Each model created had an associated Pydantic schema file to define the structure of the data that is expected in every case. Using Pydantic made life easier as there didn't need to be conditions to check specific response or requests, if certain data was passed wrong such as data type expecting a string was passed a number, Pydantic would raise an error with a detailed response.

In the figure below, each new class inherits from the Pydantic base model which activates the validation and serialisation of data input and output. Certain fields can be made optional meaning that data can be assigned to specific type if they exist. Certain fields might inherit from another schema such as `user_id` expects a list which is of type `UserI`. The config class with the attribute `from_attributes = true`, indicates to Pydantic that it can extract from objects rather than plain dictionaries.



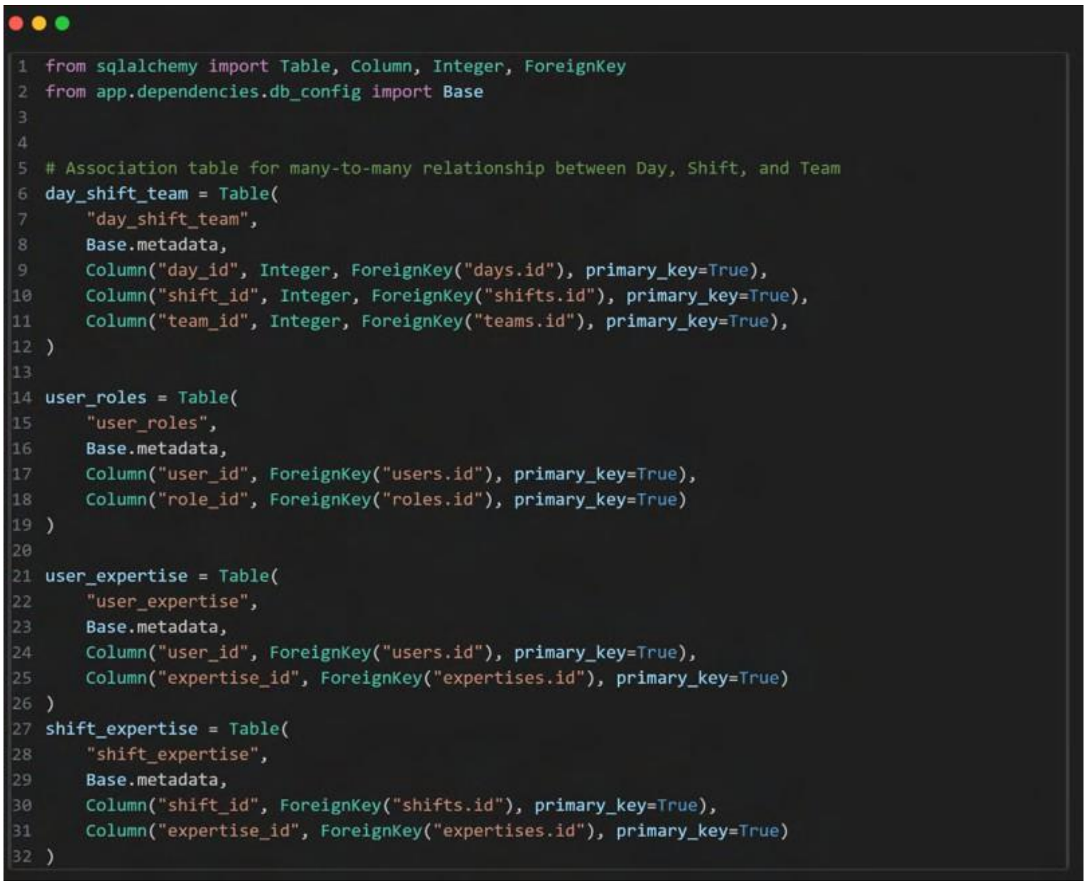
```python
from pydantic import BaseModel
from typing import List, Optional

class TeamCreate(BaseModel):
    name: str

class UserI(BaseModel):
    id: int
    email: str
    first_name: str
    last_name: str

    class Config:
        from_attributes = True

class TeamResponse(BaseModel):
    id: int
    name: str
    creator_id: int
    user_ids: List[UserI]

    class Config:
        from_attributes = True
```

Figure 58, Schema for Team

5.4.8 Association tables and Many-to-Many Relationships

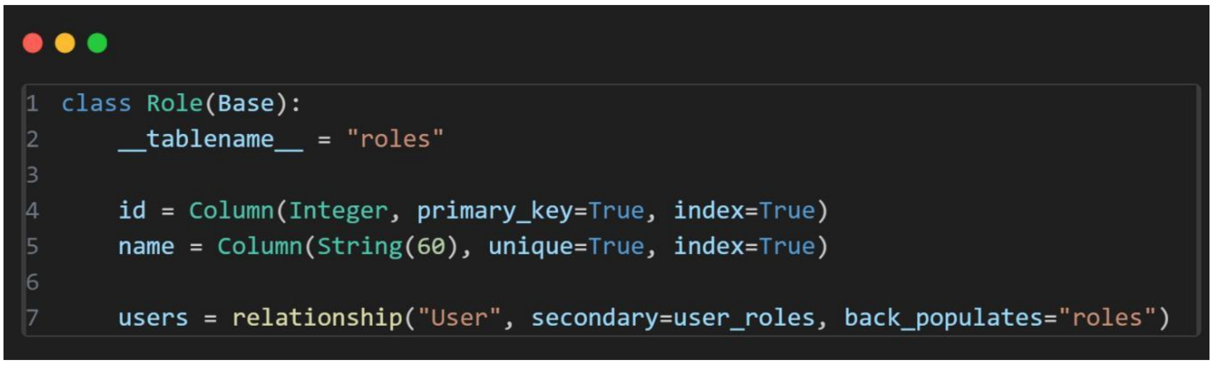
As the application included many to many relationships, association tables had to be created specifically within `association.py` file. The relationship within the model had to specify the secondary table it was linking, to allow a connection between one table and the many to many tables.

The figures below show how the association tables were created and what relationships were needed to associate to the linking table. The relationship between user and role are shown how they both are linking to a secondary table called `user_roles`.



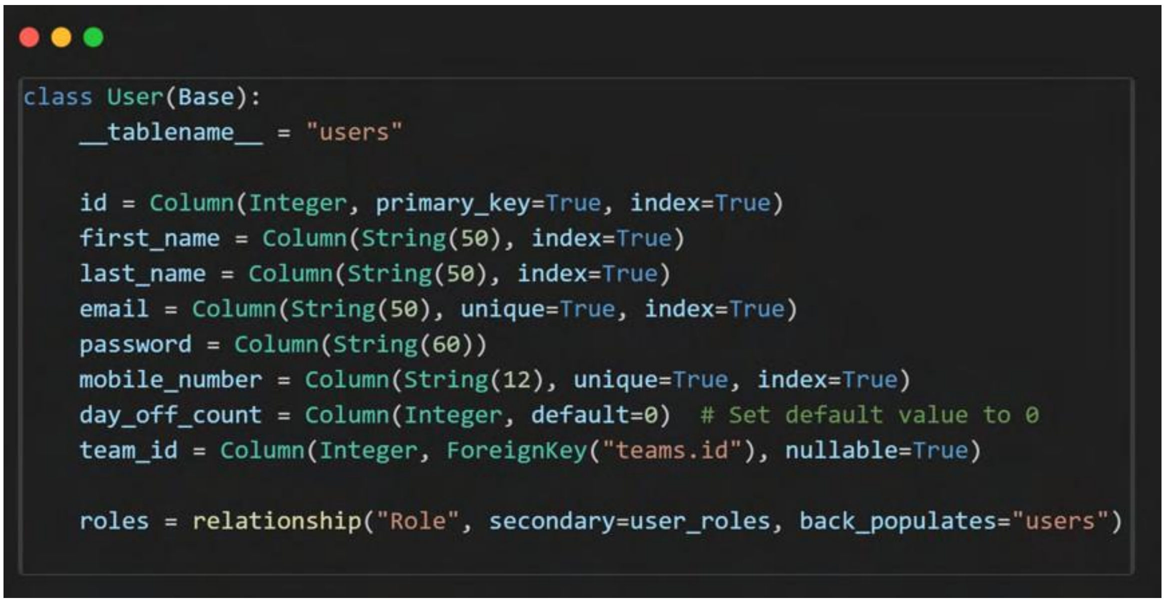
```python
from sqlalchemy import Table, Column, Integer, ForeignKey
from app.dependencies.db_config import Base


# Association table for many-to-many relationship between Day, Shift, and Team
day_shift_team = Table(
    "day_shift_team",
    Base.metadata,
    Column("day_id", Integer, ForeignKey("days.id"), primary_key=True),
    Column("shift_id", Integer, ForeignKey("shifts.id"), primary_key=True),
    Column("team_id", Integer, ForeignKey("teams.id"), primary_key=True),
)

user_roles = Table(
    "user_roles",
    Base.metadata,
    Column("user_id", ForeignKey("users.id"), primary_key=True),
    Column("role_id", ForeignKey("roles.id"), primary_key=True)
)

user_expertise = Table(
    "user_expertise",
    Base.metadata,
    Column("user_id", ForeignKey("users.id"), primary_key=True),
    Column("expertise_id", ForeignKey("expertises.id"), primary_key=True)
)
shift_expertise = Table(
    "shift_expertise",
    Base.metadata,
    Column("shift_id", ForeignKey("shifts.id"), primary_key=True),
    Column("expertise_id", ForeignKey("expertises.id"), primary_key=True)
)
```

Figure 59, Association Tables



```python
class Role(Base):
    __tablename__ = "roles"

    id = Column(Integer, primary_key=True, index=True)
    name = Column(String(60), unique=True, index=True)

    users = relationship("User", secondary=user_roles, back_populates="roles")
```

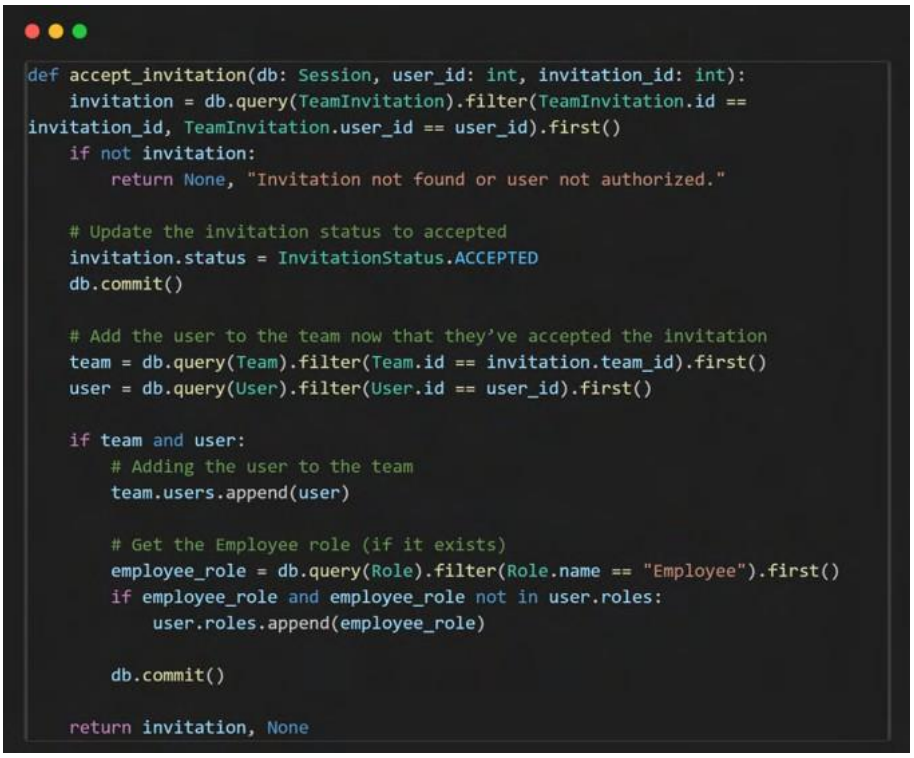
Figure 60, Relationship for Association



```python
class User(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True, index=True)
    first_name = Column(String(50), index=True)
    last_name = Column(String(50), index=True)
    email = Column(String(50), unique=True, index=True)
    password = Column(String(60))
    mobile_number = Column(String(12), unique=True, index=True)
    day_off_count = Column(Integer, default=0)  # Set default value to 0
    team_id = Column(Integer, ForeignKey("teams.id"), nullable=True)

    roles = relationship("Role", secondary=user_roles, back_populates="users")
```

Figure 61, Relationship for Association 2

These many to many tables' columns would be imported or removed during specific functions such as, when user accepts a team invite, the role and user is appended to the many to many. The figure below represents the `accept_inviation` function within `team_inviation_crud.py` to show how this append is implemented.



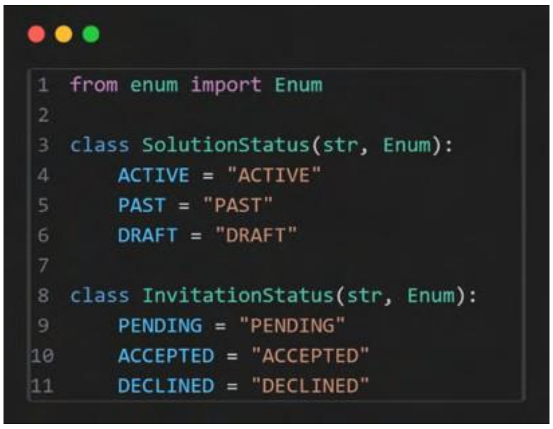
```python
def accept_invitation(db: Session, user_id: int, invitation_id: int):
    invitation = db.query(TeamInvitation).filter(TeamInvitation.id ==
invitation_id, TeamInvitation.user_id == user_id).first()
    if not invitation:
        return None, "Invitation not found or user not authorized."

    # Update the invitation status to accepted
    invitation.status = InvitationStatus.ACCEPTED
    db.commit()

    # Add the user to the team now that they've accepted the invitation
    team = db.query(Team).filter(Team.id == invitation.team_id).first()
    user = db.query(User).filter(User.id == user_id).first()

    if team and user:
        # Adding the user to the team
        team.users.append(user)

        # Get the Employee role (if it exists)
        employee_role = db.query(Role).filter(Role.name == "Employee").first()
        if employee_role and employee_role not in user.roles:
            user.roles.append(employee_role)

        db.commit()

    return invitation, None
```

Figure 62, Editing Association tables

5.4.9 Enums Handling for Status

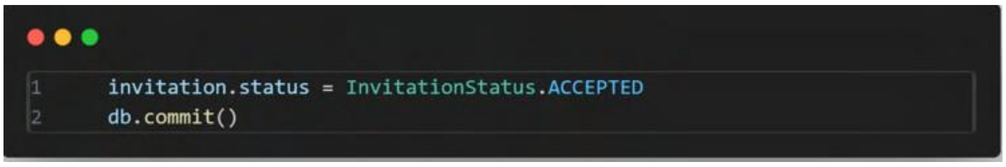
Enumerations (Enums) are used to define a set of predefined values for a specific variable. This would improve data integrity and maintainability. These Enums were used specifically for managing status of various fields such as, solution status and invitation status. They were created within `enums.py` file and imported in schema and the model for the related field.

In the figure below we can see how these Enums were created within the `enum.py` file and how they were used to specify default values within the model, schema and CRUD.



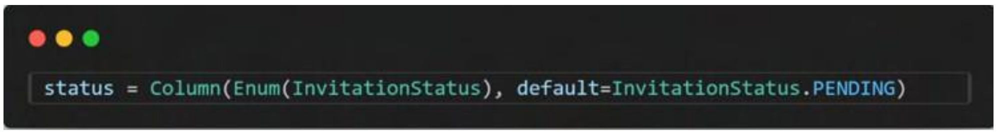
```python
from enum import Enum

class SolutionStatus(str, Enum):
    ACTIVE = "ACTIVE"
    PAST = "PAST"
    DRAFT = "DRAFT"

class InvitationStatus(str, Enum):
    PENDING = "PENDING"
    ACCEPTED = "ACCEPTED"
    DECLINED = "DECLINED"
```

Figure 63, Enum Schema



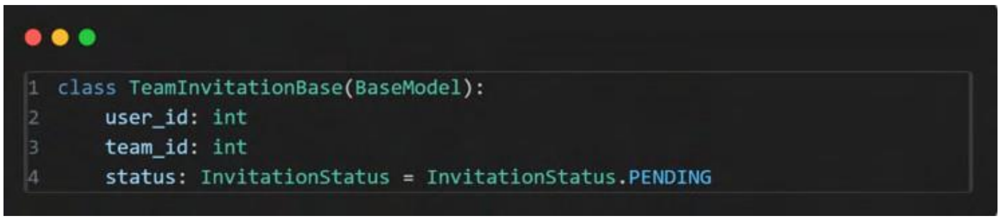
```python
    invitation.status = InvitationStatus.ACCEPTED
    db.commit()
```

Figure 64, Enum Put Request



```python
status = Column(Enum(InvitationStatus), default=InvitationStatus.PENDING)
```

Figure 65, Enum Within Model



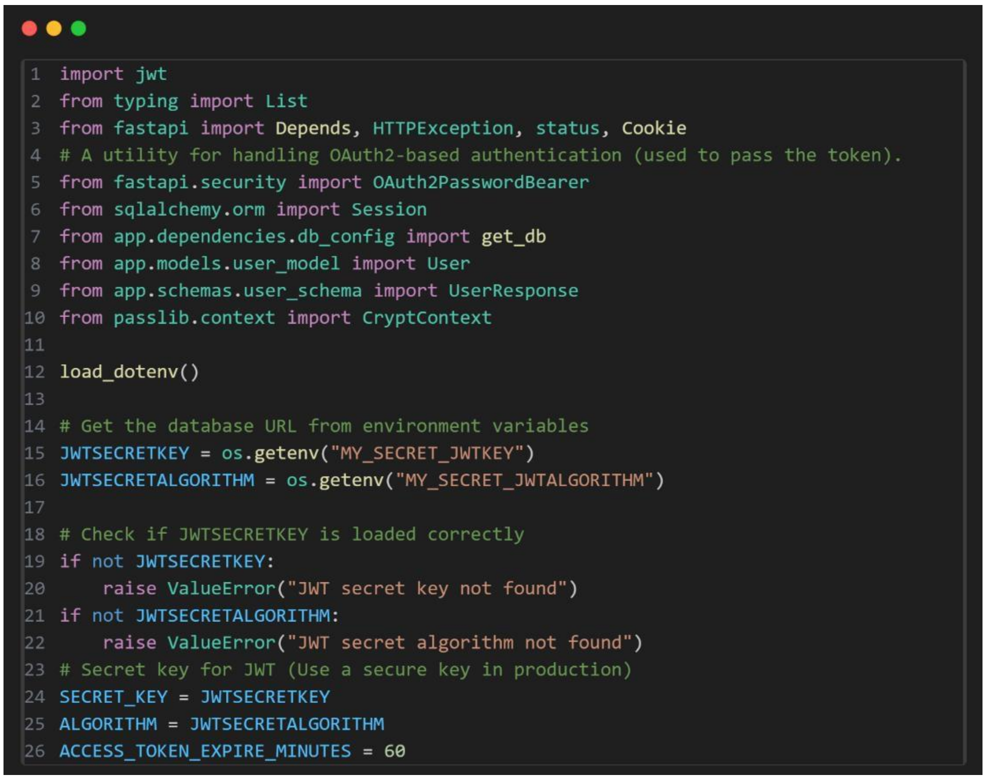
```python
class TeamInvitationBase(BaseModel):
    user_id: int
    team_id: int
    status: InvitationStatus = InvitationStatus.PENDING
```

Figure 66, Enum Schema Child Called

5.4.10 Authentication and Access Control

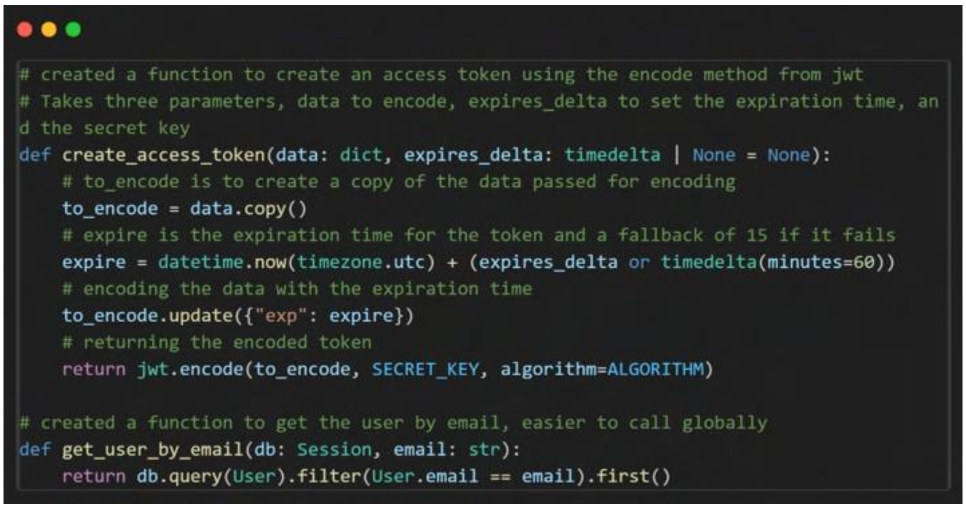
Authentication and Access Control were vital to ensure security and implement RBAC into the application. All related information with authentication and access control was stored within auth.py in the dependencies folder. Inside `auth.py`, numerous functions were implemented such as, access token creation, encrypting the password, verifying the password, getting the current user through the access token, and checking the current users' roles. The following explains how each function was created and how the function was called within the route.

The figure below shows how the `create_access_token` function was implemented to create a token using JWT and store it as a cookie. This function took two parameters, `data`; which is a dictionary about the user's information which will be encoded into JWT and `expires_delta`; which allows to set an expiration time for the token. The function begins with copying the data dictionary to add expiry and ensure the original data isn't tampered with. The JWT token is then encoded with using a JWT function called encode which takes the data, secret key and specified algorithm. This function is enabled during login function within auth_route.py. Once the user passes conditions, it sets the cookie to the access token created, ensures the cookie is inaccessible to JavaScript, ensures the cookie is sent only through a HTTPS connection, and how the cookie is sent with cross-site requests, which will be changed after development.



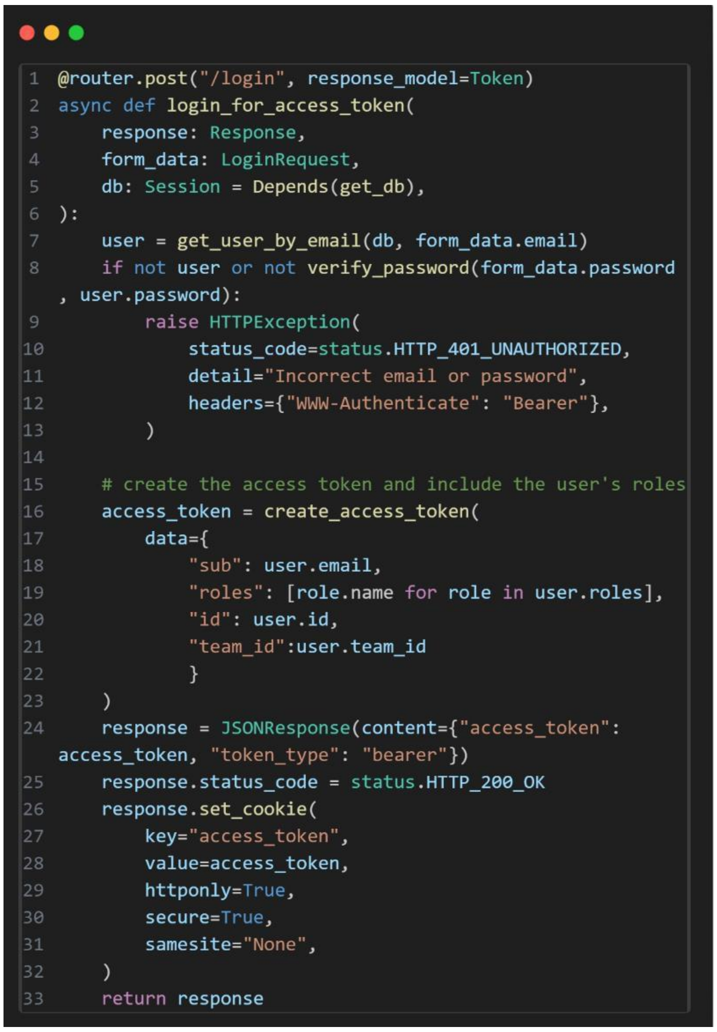
```python
import jwt
from typing import List
from fastapi import Depends, HTTPException, status, Cookie
# A utility for handling OAuth2-based authentication (used to pass the token).
from fastapi.security import OAuth2PasswordBearer
from sqlalchemy.orm import Session
from app.dependencies.db_config import get_db
from app.models.user_model import User
from app.schemas.user_schema import UserResponse
from passlib.context import CryptContext

load_dotenv()

# Get the database URL from environment variables
JWTSECRETKEY = os.getenv("MY_SECRET_JWTKEY")
JWTSECRETALGORITHM = os.getenv("MY_SECRET_JWTALGORITHM")

# Check if JWTSECRETKEY is loaded correctly
if not JWTSECRETKEY:
    raise ValueError("JWT secret key not found")
if not JWTSECRETALGORITHM:
    raise ValueError("JWT secret algorithm not found")
# Secret key for JWT (Use a secure key in production)
SECRET_KEY = JWTSECRETKEY
ALGORITHM = JWTSECRETALGORITHM
ACCESS_TOKEN_EXPIRE_MINUTES = 60
```

Figure 67, Access token Set up



```python
# created a function to create an access token using the encode method from jwt
# Takes three parameters, data to encode, expires_delta to set the expiration time, an
d the secret key
def create_access_token(data: dict, expires_delta: timedelta | None = None):
    # to_encode is to create a copy of the data passed for encoding
    to_encode = data.copy()
    # expire is the expiration time for the token and a fallback of 15 if it fails
    expire = datetime.now(timezone.utc) + (expires_delta or timedelta(minutes=60))
    # encoding the data with the expiration time
    to_encode.update({"exp": expire})
    # returning the encoded token
    return jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)

# created a function to get the user by email, easier to call globally
def get_user_by_email(db: Session, email: str):
    return db.query(User).filter(User.email == email).first()
```

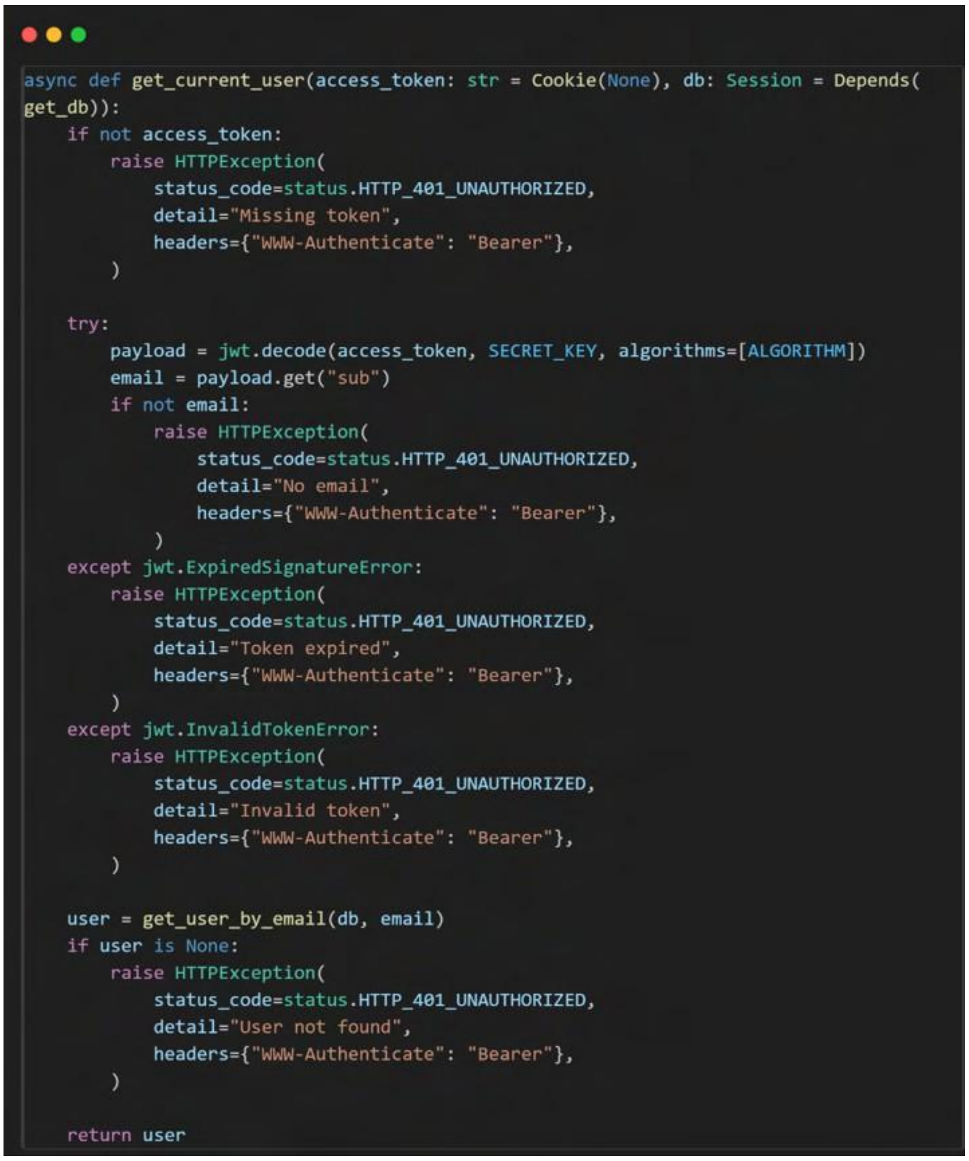
Figure 68, Function To create Access Token



```python
@router.post("/login", response_model=Token)
async def login_for_access_token(
    response: Response,
    form_data: LoginRequest,
    db: Session = Depends(get_db),
):
    user = get_user_by_email(db, form_data.email)
    if not user or not verify_password(form_data.password
, user.password):
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect email or password",
            headers={"WWW-Authenticate": "Bearer"},
        )

    # create the access token and include the user's roles
    access_token = create_access_token(
        data={
            "sub": user.email,
            "roles": [role.name for role in user.roles],
            "id": user.id,
            "team_id":user.team_id
            }
    )
    response = JSONResponse(content={"access_token":
access_token, "token_type": "bearer"})
    response.status_code = status.HTTP_200_OK
    response.set_cookie(
        key="access_token",
        value=access_token,
        httponly=True,
        secure=True,
        samesite="None",
    )
    return response
```

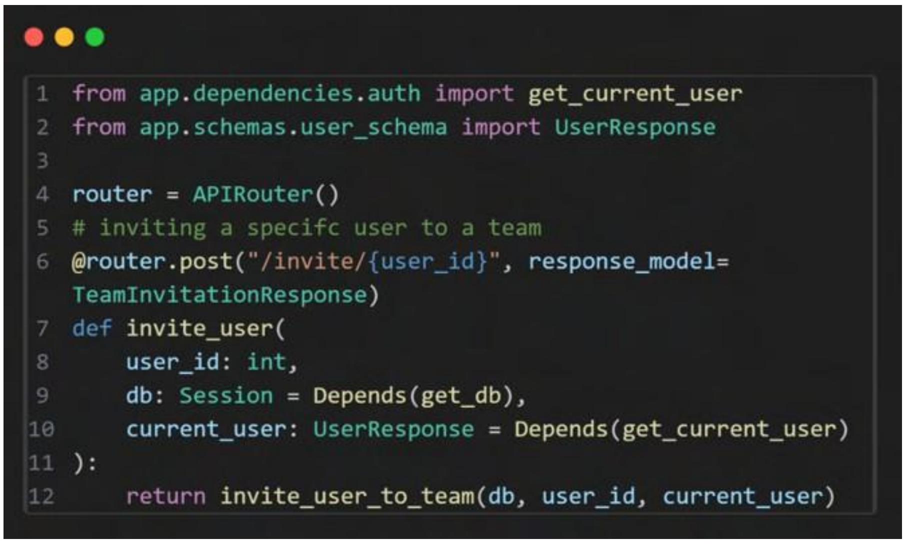
Figure 69, Access Token function called with relevant data passed

The access token is now stored. The `current_user` function can access the token and implement it to specific routes to ensure that routes need to be authenticated before access. This function specifies that it expects a cookie named `access_token`. The function includes, try-except for exception handling of various conditions. These conditions to check if there is no email, if the token is expired, if its invalid or if the token is missing. If all conditions are passed, its tries decode the JWT to verify the users and retrieves information on that user. This function can be used to authenticate routes and stop users who aren’t logged in from accessing specific routes. This function would generally be called during dependency injections to run asynchronously. The following figures provide the `current_user` function and how it is used.



```python
async def get_current_user(access_token: str = Cookie(None), db: Session = Depends(
get_db)):
    if not access_token:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Missing token",
            headers={"WWW-Authenticate": "Bearer"},
        )

    try:
        payload = jwt.decode(access_token, SECRET_KEY, algorithms=[ALGORITHM])
        email = payload.get("sub")
        if not email:
            raise HTTPException(
                status_code=status.HTTP_401_UNAUTHORIZED,
                detail="No email",
                headers={"WWW-Authenticate": "Bearer"},
            )
    except jwt.ExpiredSignatureError:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Token expired",
            headers={"WWW-Authenticate": "Bearer"},
        )
    except jwt.InvalidTokenError:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Invalid token",
            headers={"WWW-Authenticate": "Bearer"},
        )

    user = get_user_by_email(db, email)
    if user is None:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="User not found",
            headers={"WWW-Authenticate": "Bearer"},
        )

    return user
```

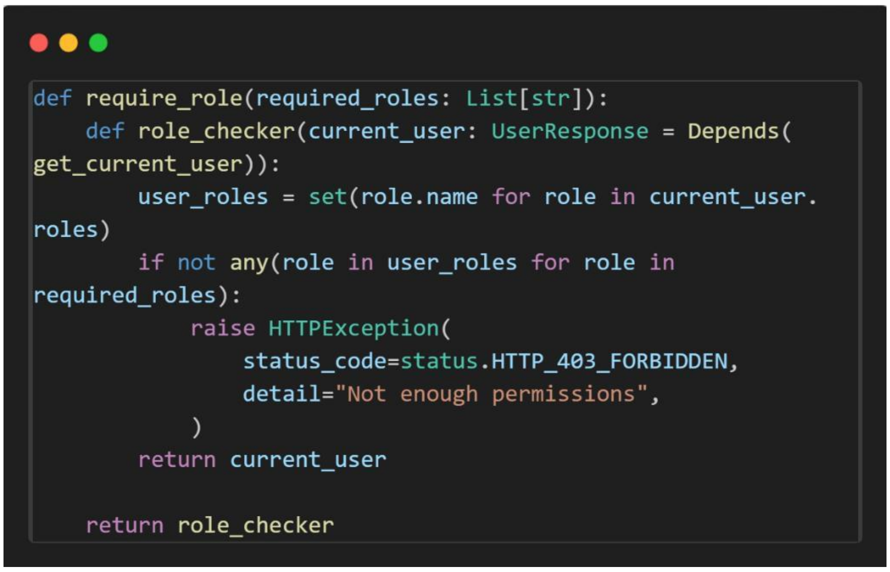
Figure 70, Decrypting Token



```python
from app.dependencies.auth import get_current_user
from app.schemas.user_schema import UserResponse

router = APIRouter()
# inviting a specifc user to a team
@router.post("/invite/{user_id}", response_model=
TeamInvitationResponse)
def invite_user(
    user_id: int,
    db: Session = Depends(get_db),
    current_user: UserResponse = Depends(get_current_user)
):
    return invite_user_to_team(db, user_id, current_user)
```

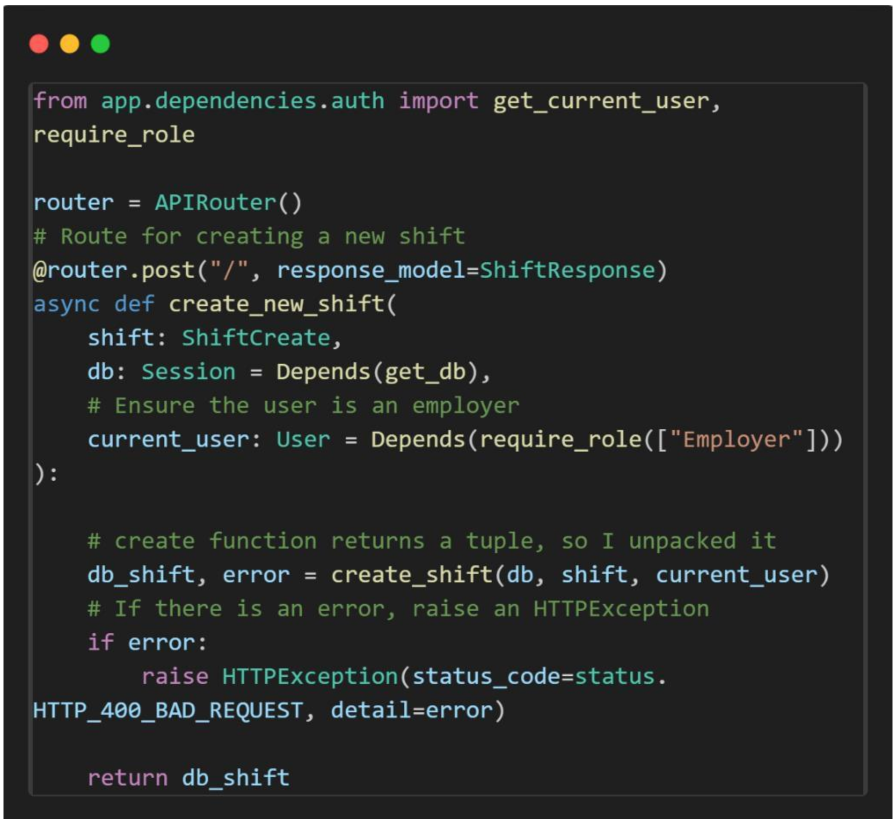
Figure 71, Decrypting Token in Route to Validate

A function was created to implement the RBAC for the application. `Require_role` function would use a dependency injection of `current_user` which would check if the current users' roles match any of the roles which are provided in the required_roles parameter with the function. The required roles would be defined and dependant on the route. If the user didn't match any of the roles, then it would pass an error of status 403, forbidden. The figures below demonstrate how this function was created and implemented in a shift create route where only the Employer could use this route.



```python
def require_role(required_roles: List[str]):
    def role_checker(current_user: UserResponse = Depends(
get_current_user)):
        user_roles = set(role.name for role in current_user.
roles)
        if not any(role in user_roles for role in
required_roles):
            raise HTTPException(
                status_code=status.HTTP_403_FORBIDDEN,
                detail="Not enough permissions",
            )
        return current_user

    return role_checker
```

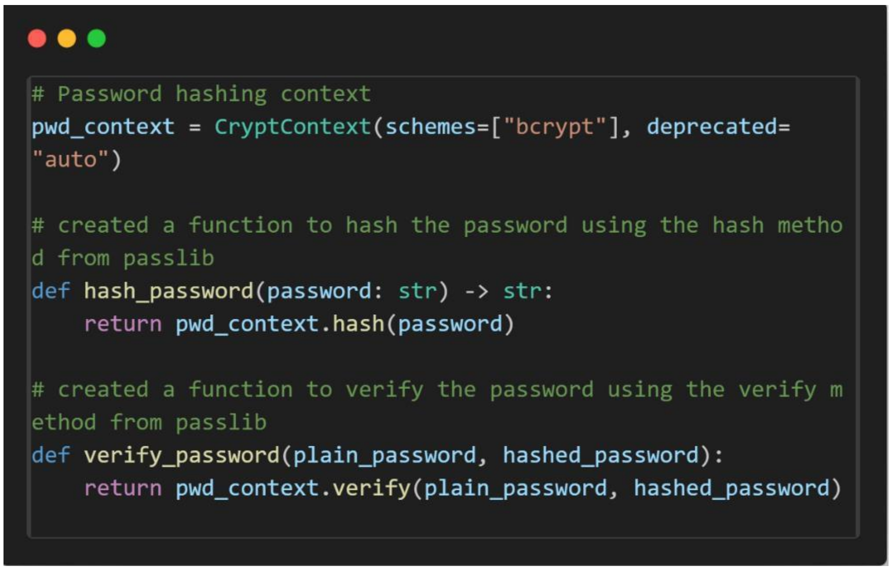
Figure 72, Require Role Function



```python
from app.dependencies.auth import get_current_user,
require_role

router = APIRouter()
# Route for creating a new shift
@router.post("/", response_model=ShiftResponse)
async def create_new_shift(
    shift: ShiftCreate,
    db: Session = Depends(get_db),
    # Ensure the user is an employer
    current_user: User = Depends(require_role(["Employer"]))
):

    # create function returns a tuple, so I unpacked it
    db_shift, error = create_shift(db, shift, current_user)
    # If there is an error, raise an HTTPException
    if error:
        raise HTTPException(status_code=status.
HTTP_400_BAD_REQUEST, detail=error)

    return db_shift
```

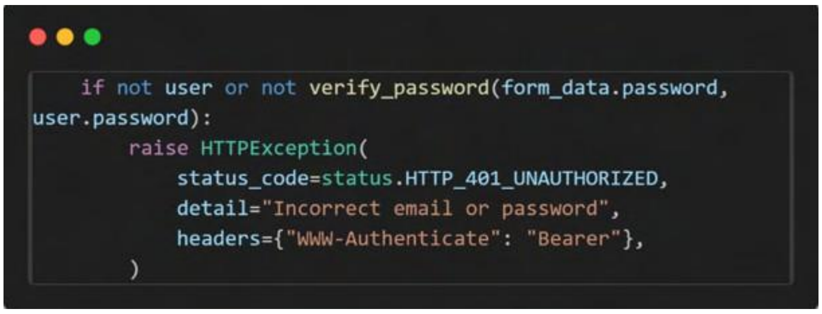
Figure 73, Role function called passing role to be validated

The final functions with auth.py, managed the password encryption and decryption. The `hash_password` function was responsible for encrypting using CryptContext which was a class from passlib library. CryptContext took to parameters, the scheme which was the hashing algorithm (schemes=[“bcrypt”]) and depreciated=” auto”, will automatically mark any future algorithms as deprecated, ensuring that the password hashing uses up-to-date algorithms. This context created can now hash the password using the method provided by the instance of CryptContext. To Verify the password at login the `verify_password` function is used, which uses method provided by CryptContext called verify. This takes the hashed password from the database and the plain password provided in the login form. Figures of both functions are found below, with the use of verify in login and hash in `create_user` routes.



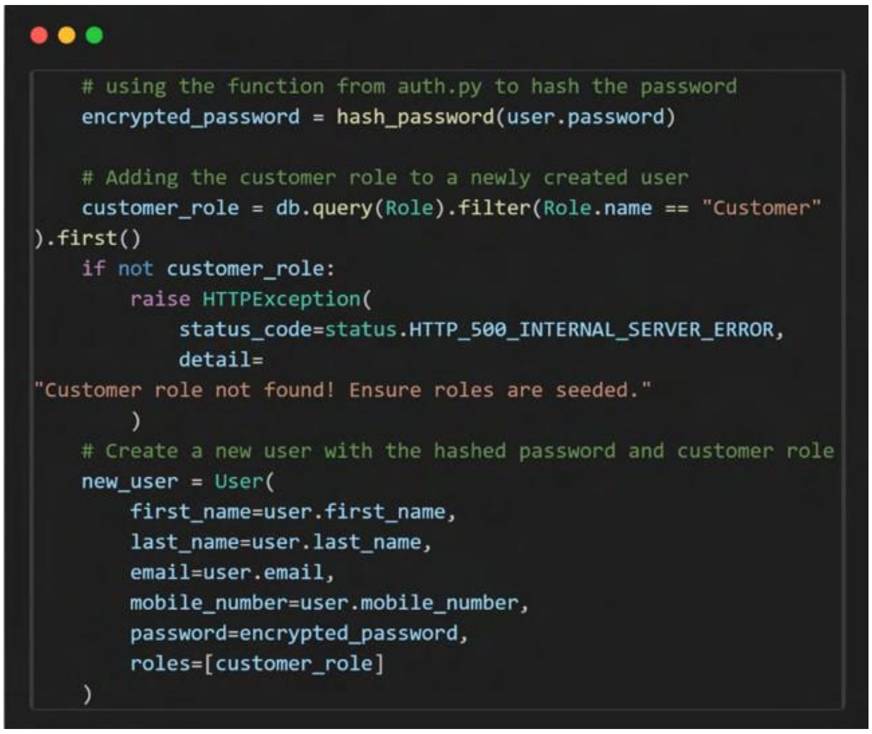
```
# Password hashing context
pwd_context = CryptContext(schemes=["bcrypt"], deprecated=
"auto")

# created a function to hash the password using the hash metho
d from passlib
def hash_password(password: str) -> str:
    return pwd_context.hash(password)

# created a function to verify the password using the verify m
ethod from passlib
def verify_password(plain_password, hashed_password):
    return pwd_context.verify(plain_password, hashed_password)
```

Figure 74, Functions to encrypt and verify the password



```
    if not user or not verify_password(form_data.password,
user.password):
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect email or password",
            headers={"WWW-Authenticate": "Bearer"},
        )
```

Figure 75, Verify password Function in use



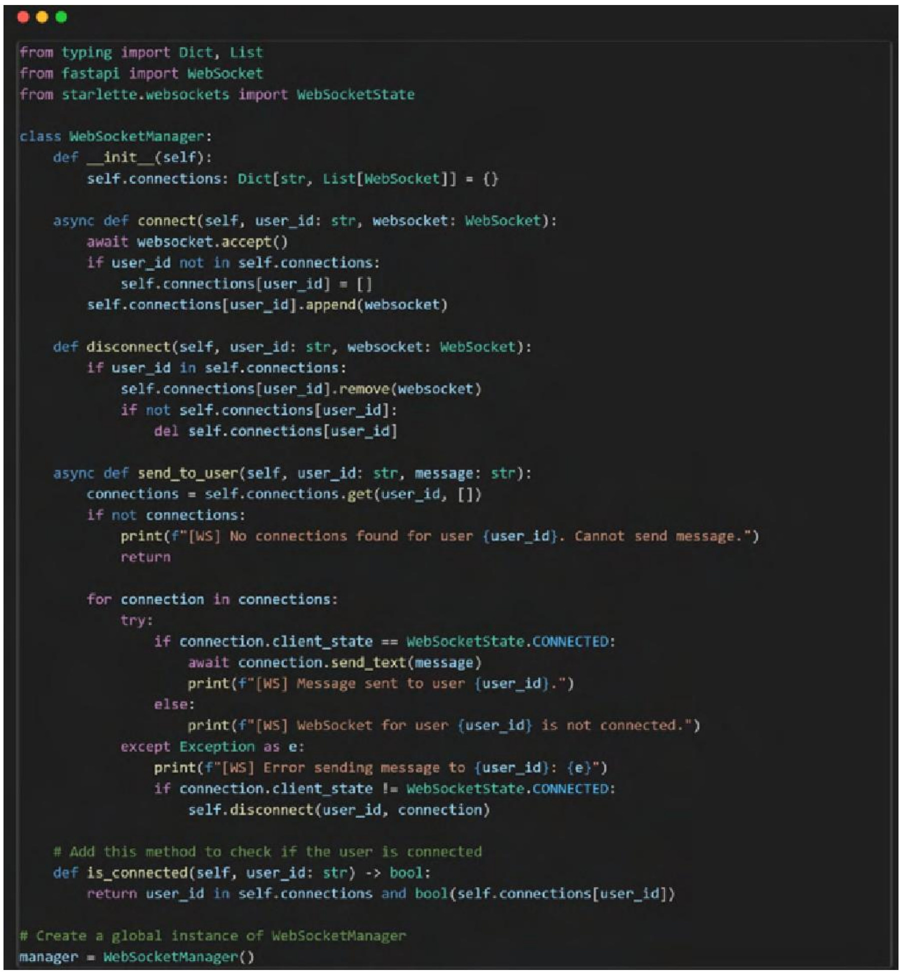
```
    # using the function from auth.py to hash the password
    encrypted_password = hash_password(user.password)

    # Adding the customer role to a newly created user
    customer_role = db.query(Role).filter(Role.name == "Customer"
).first()
    if not customer_role:
        raise HTTPException(
            status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail=
"Customer role not found! Ensure roles are seeded."
        )
    # Create a new user with the hashed password and customer role
    new_user = User(
        first_name=user.first_name,
        last_name=user.last_name,
        email=user.email,
        mobile_number=user.mobile_number,
        password=encrypted_password,
        roles=[customer_role]
    )
```

Figure 76, Hash Password Function used before posting to database

5.4.11 WebSocket's

As mentioned in the system architecture section (3.3.3.1), WebSocket's were implemented within the backend to maintain a connection between the server and the client, allowing real-time notifications with changes. A file called `websocket_manager.py` was created to hold a class with multiple functions handling connection, disconnection and targeted message delivery. The class stored all active connections within a dictionary, using the ids from the user as the key and the list of WebSocket instances as values.

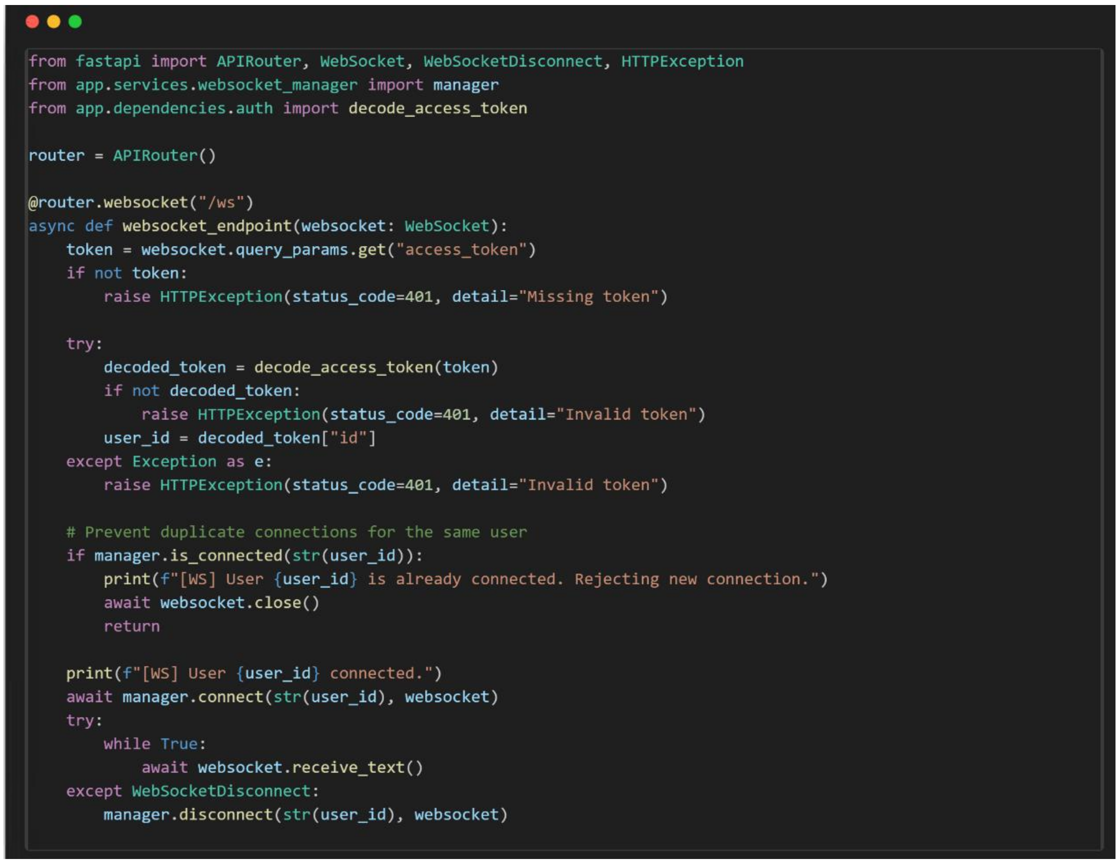
Once a user connected through the `connect` method, their id was stored in the connections dictionary. Once the connection is disconnected using the `Disconnect` method, the user's id is removed from the dictionary. The `send_to_user` method allowed the backend to send the connected user a message directly over the WebSocket channel. A global instance was created which ensured that the WebSocket manager class could be accessed from any file in the back end.



```python
from typing import Dict, List
from fastapi import WebSocket
from starlette.websockets import WebSocketState

class WebSocketManager:
    def __init__(self):
        self.connections: Dict[str, List[WebSocket]] = {}

    async def connect(self, user_id: str, websocket: WebSocket):
        await websocket.accept()
        if user_id not in self.connections:
            self.connections[user_id] = []
        self.connections[user_id].append(websocket)

    def disconnect(self, user_id: str, websocket: WebSocket):
        if user_id in self.connections:
            self.connections[user_id].remove(websocket)
            if not self.connections[user_id]:
                del self.connections[user_id]

    async def send_to_user(self, user_id: str, message: str):
        connections = self.connections.get(user_id, [])
        if not connections:
            print(f"[WS] No connections found for user {user_id}. Cannot send message.")
            return

        for connection in connections:
            try:
                if connection.client_state == WebSocketState.CONNECTED:
                    await connection.send_text(message)
                    print(f"[WS] Message sent to user {user_id}.")
                else:
                    print(f"[WS] WebSocket for user {user_id} is not connected.")
            except Exception as e:
                print(f"[WS] Error sending message to {user_id}: {e}")
                if connection.client_state != WebSocketState.CONNECTED:
                    self.disconnect(user_id, connection)

    # Add this method to check if the user is connected
    def is_connected(self, user_id: str) -> bool:
        return user_id in self.connections and bool(self.connections[user_id])

# Create a global instance of WebSocketManager
manager = WebSocketManager()
```

Figure 77, WebSocket Manager

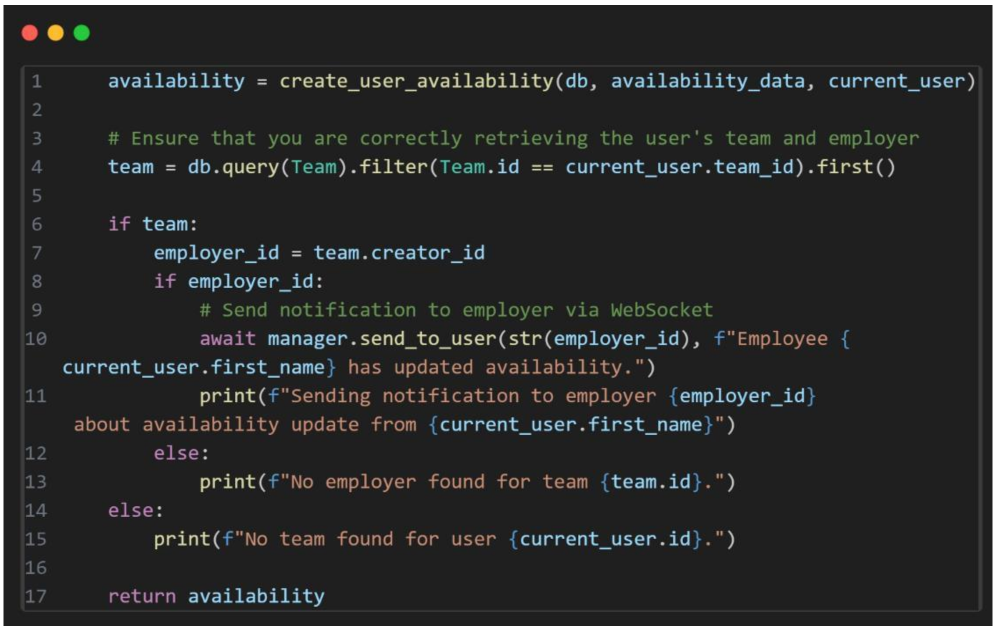
To initiate this communication, a WebSocket route was registered within the main.py file, `app.include_router(websocket_router, tags=["WebSocket"])`. This enabled users to establish a WebSocket connection with the server. The router for WebSocket's was created within the `websocket_route.py` file. This route began with decoding the `access_token` to retrieve the users id. Using the retrieved id, a condition was implemented to check the dictionary of users to double check the user wasn't already connected, this prevented duplicate entries. If the condition was passed it would try connecting the user to the WebSocket server.



```python
from fastapi import APIRouter, WebSocket, WebSocketDisconnect, HTTPException
from app.services.websocket_manager import manager
from app.dependencies.auth import decode_access_token

router = APIRouter()

@router.websocket("/ws")
async def websocket_endpoint(websocket: WebSocket):
    token = websocket.query_params.get("access_token")
    if not token:
        raise HTTPException(status_code=401, detail="Missing token")

    try:
        decoded_token = decode_access_token(token)
        if not decoded_token:
            raise HTTPException(status_code=401, detail="Invalid token")
        user_id = decoded_token["id"]
    except Exception as e:
        raise HTTPException(status_code=401, detail="Invalid token")

    # Prevent duplicate connections for the same user
    if manager.is_connected(str(user_id)):
        print(f"[WS] User {user_id} is already connected. Rejecting new connection.")
        await websocket.close()
        return

    print(f"[WS] User {user_id} connected.")
    await manager.connect(str(user_id), websocket)
    try:
        while True:
            await websocket.receive_text()
    except WebSocketDisconnect:
        manager.disconnect(str(user_id), websocket)
```

Figure 78, WebSocket Route

The Figure below illustrates how the messages were sent upon change to the database. Once a user with the role of Employee creates their availability, `send_to_user` function is initiated by placing the `employer_id` into the parameters, as the target user. The second parameter is the message that is being sent to this target user. In relation to Figure 79, the message is indicating that the user which created the availability has updated their availability. This ensures the team is connected with any urgent requests.



```python
    availability = create_user_availability(db, availability_data, current_user)

    # Ensure that you are correctly retrieving the user's team and employer
    team = db.query(Team).filter(Team.id == current_user.team_id).first()

    if team:
        employer_id = team.creator_id
        if employer_id:
            # Send notification to employer via WebSocket
            await manager.send_to_user(str(employer_id), f"Employee {
current_user.first_name} has updated availability.")
            print(f"Sending notification to employer {employer_id}
 about availability update from {current_user.first_name}")
        else:
            print(f"No employer found for team {team.id}.")
    else:
        print(f"No team found for user {current_user.id}.")

    return availability
```

Figure 79, WebSocket Availability Message

5.4.12 Testing and Debugging

During development the testing and debugging of the application was important. This ensured that if there were any issues upon creation, they can be tackled right away. Debugging the API was done using Swagger UI documentation provided by FastAPI. Testing endpoints and checking validation for each route created. As Swagger UI was updated in real time it decreased time consumption of restarting the server. Insomnia was also used for debugging; the implementation of insomnia allowed to store and export the routes and organise the collections based on model. The following figures show the structure of both, Swagger UI and Insomnia Collection.

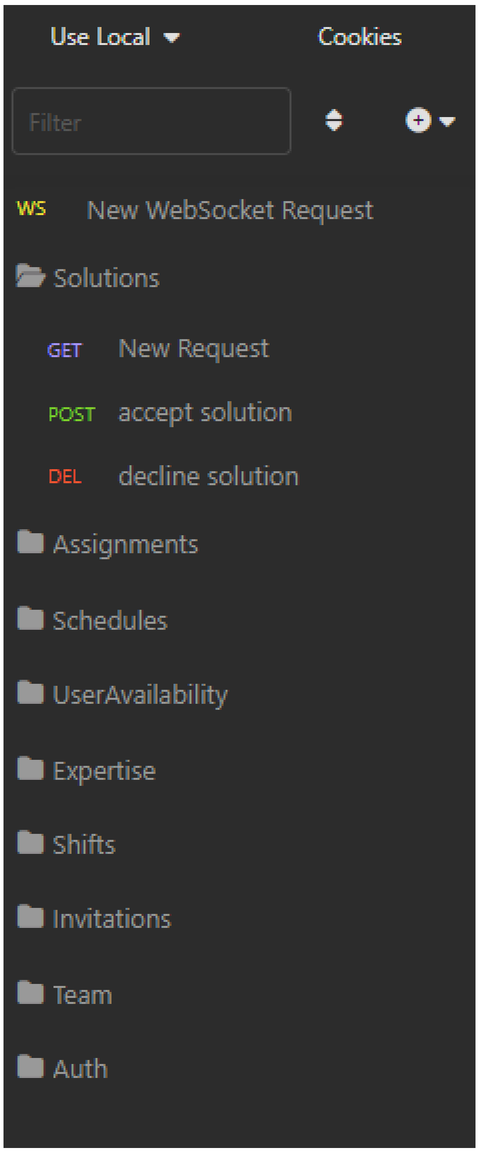
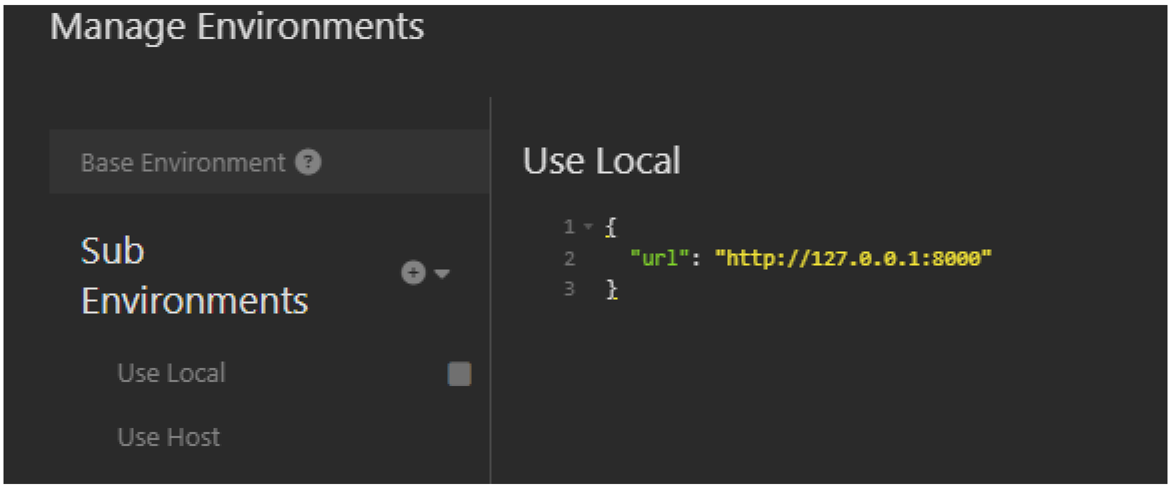
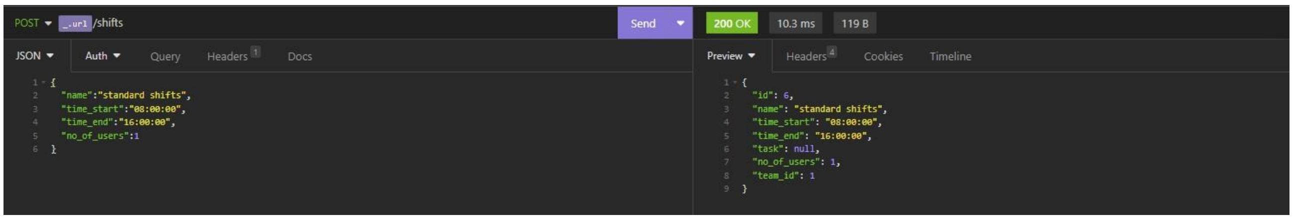


Figure 80, Insomnia Modularisation



```
{
  "url": "http://127.0.0.1:8000"
}
```

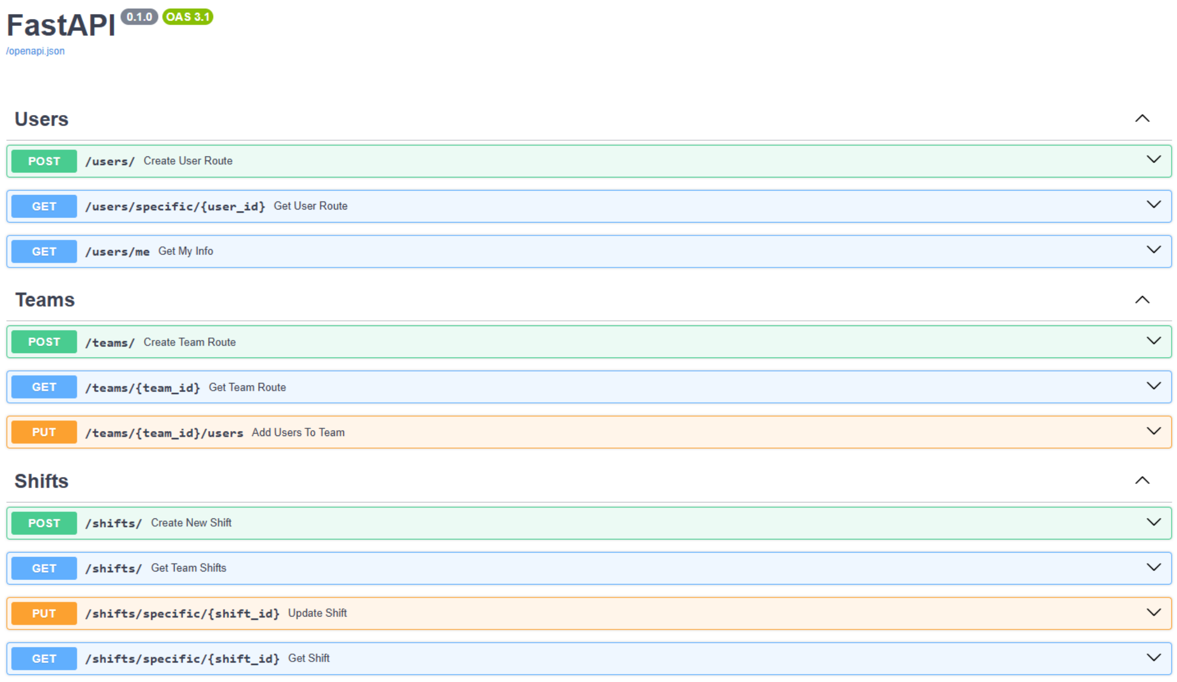
Figure 81, Insomnia Environments



```
{
  "name":"standard shifts",
  "time_start":"08:00:00",
  "time_end":"16:00:00",
  "no_of_users":1
}
```

```
{
  "id": 6,
  "name": "standard shifts",
  "time_start": "08:00:00",
  "time_end": "16:00:00",
  "task": null,
  "no_of_users": 1,
  "team_id": 1
}
```

Figure 82, Insomnia Request and Response



Users

```
POST /users/  Create User Route
GET  /users/specific/{user_id}  Get User Route
GET  /users/me  Get My Info
```

Teams

```
POST /teams/  Create Team Route
GET  /teams/{team_id}  Get Team Route
PUT  /teams/{team_id}/users  Add Users To Team
```

Shifts

```
POST /shifts/  Create New Shift
GET  /shifts/  Get Team Shifts
PUT  /shifts/specific/{shift_id}  Update Shift
GET  /shifts/specific/{shift_id}  Get Shift
```

Figure 83, Swagger UI endpoints

5.5 Constraint Satisfaction Problem Solver

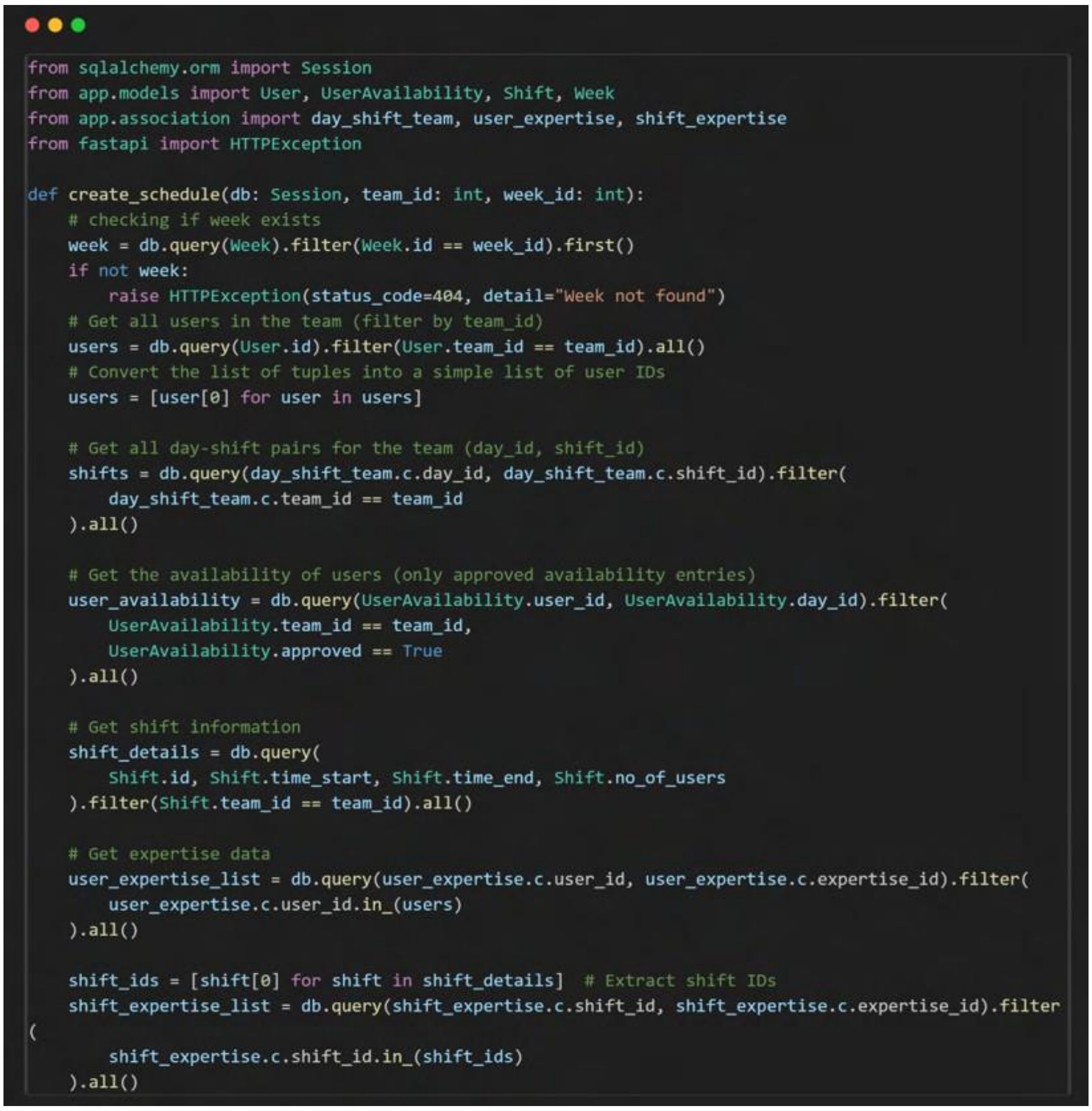
5.5.1 Overview

The Constraint Satisfaction Problem (CSP) Solver was the core functionality of the application which was implemented within the FastAPI backend. This solver needed to be implemented by using all relevant data from the team. This data included users, shift details, availability, shift expertise and user expertise. The CSP solver would be triggered through an API call which would get all data which is filtered by the `team_id` given. The solver would provide a solution or number of solutions based on the data and pass it back to display it within the return.

5.5.2 CRUD

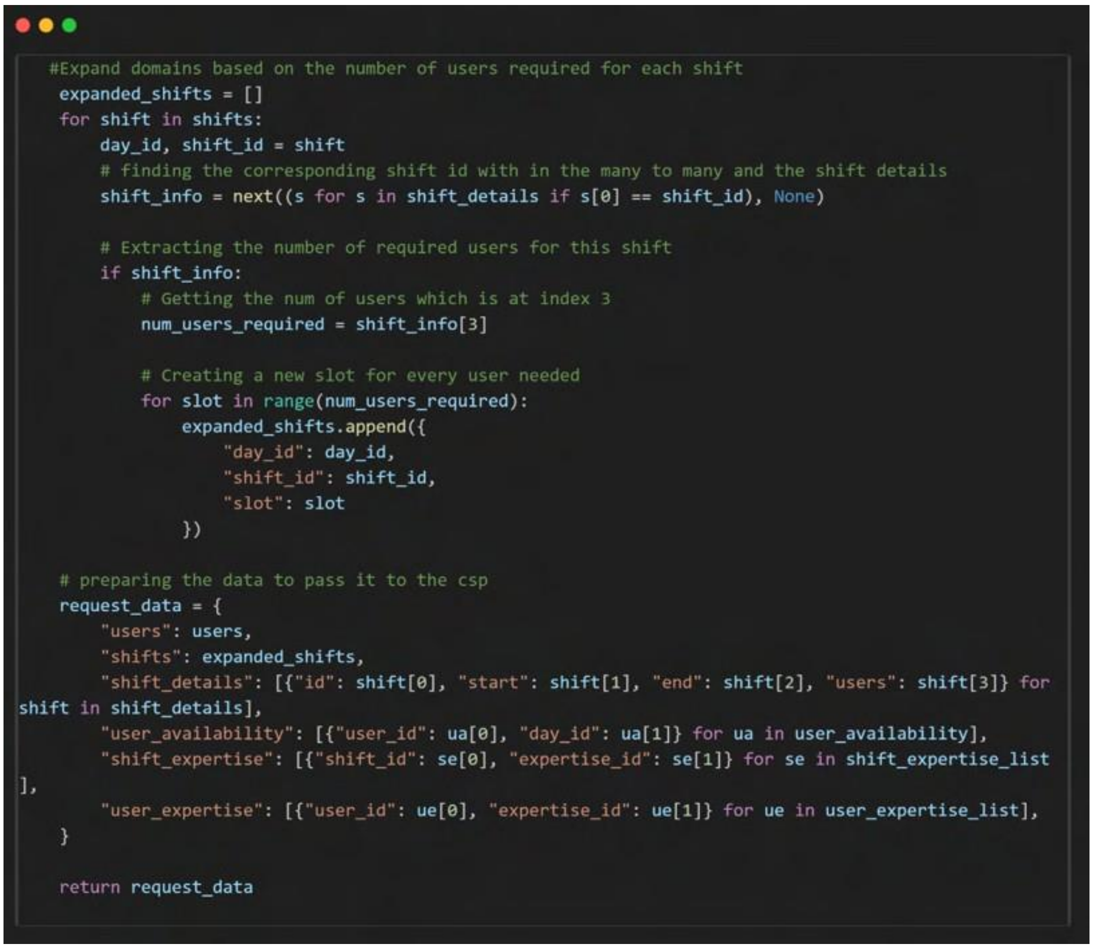
The create_schedule function prepared and returned the structured scheduling data including all relevant data by querying each table to the `team_id`. This structured data was now prepared to be sent to the CSP solver.

Some data had to be modified to support the python constraint library, such as the users were converted from a list to a tuple as the CSP required immutable types to function correctly. The concept of expanded shifts added slot field to the existing shift details, as some shifts required more than one user, adding the two variables to a single domain it would raise an error within the CSP. This is because within CSPs a domain cannot be added to two variables. By implementing slots, it essentially multiplied the shift by the number of users and assigned it a slot of the index of the loop. Once the issues were solved, the updated requests were returned to the CSP. The implementation of the crud can be seen below.



```python
from sqlalchemy.orm import Session
from app.models import User, UserAvailability, Shift, Week
from app.association import day_shift_team, user_expertise, shift_expertise
from fastapi import HTTPException

def create_schedule(db: Session, team_id: int, week_id: int):
    # checking if week exists
    week = db.query(Week).filter(Week.id == week_id).first()
    if not week:
        raise HTTPException(status_code=404, detail="Week not found")
    # Get all users in the team (filter by team_id)
    users = db.query(User.id).filter(User.team_id == team_id).all()
    # Convert the list of tuples into a simple list of user IDs
    users = [user[0] for user in users]

    # Get all day-shift pairs for the team (day_id, shift_id)
    shifts = db.query(day_shift_team.c.day_id, day_shift_team.c.shift_id).filter(
        day_shift_team.c.team_id == team_id
    ).all()

    # Get the availability of users (only approved availability entries)
    user_availability = db.query(UserAvailability.user_id, UserAvailability.day_id).filter(
        UserAvailability.team_id == team_id,
        UserAvailability.approved == True
    ).all()

    # Get shift information
    shift_details = db.query(
        Shift.id, Shift.time_start, Shift.time_end, Shift.no_of_users
    ).filter(Shift.team_id == team_id).all()

    # Get expertise data
    user_expertise_list = db.query(user_expertise.c.user_id, user_expertise.c.expertise_id).filter(
        user_expertise.c.user_id.in_(users)
    ).all()

    shift_ids = [shift[0] for shift in shift_details]  # Extract shift IDs
    shift_expertise_list = db.query(shift_expertise.c.shift_id, shift_expertise.c.expertise_id).filter
(
        shift_expertise.c.shift_id.in_(shift_ids)
    ).all()
```

Figure 84, CRUD CSP 1



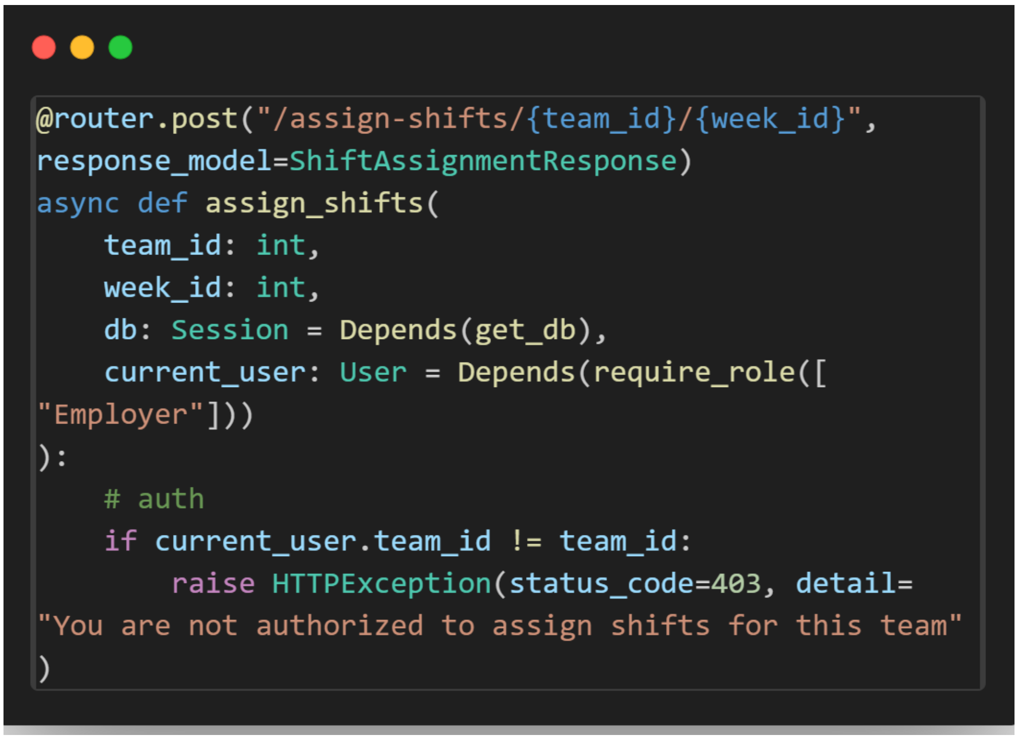
```python
    #Expand domains based on the number of users required for each shift
    expanded_shifts = []
    for shift in shifts:
        day_id, shift_id = shift
        # finding the corresponding shift id with in the many to many and the shift details
        shift_info = next((s for s in shift_details if s[0] == shift_id), None)

        # Extracting the number of required users for this shift
        if shift_info:
            # Getting the num of users which is at index 3
            num_users_required = shift_info[3]

            # Creating a new slot for every user needed
            for slot in range(num_users_required):
                expanded_shifts.append({
                    "day_id": day_id,
                    "shift_id": shift_id,
                    "slot": slot
                })

    # preparing the data to pass it to the csp
    request_data = {
        "users": users,
        "shifts": expanded_shifts,
        "shift_details": [{"id": shift[0], "start": shift[1], "end": shift[2], "users": shift[3]} for
shift in shift_details],
        "user_availability": [{"user_id": ua[0], "day_id": ua[1]} for ua in user_availability],
        "shift_expertise": [{"shift_id": se[0], "expertise_id": se[1]} for se in shift_expertise_list
],
        "user_expertise": [{"user_id": ue[0], "expertise_id": ue[1]} for ue in user_expertise_list],
    }

    return request_data
```

Figure 85, CRUD CSP 2

5.5.3 Route

The `assign_shifts` function handles the logic of generating and storing the return from the CSP. The route is responsible for passing the formatted data from the `create_schedule` function and stores the result from the solver into the solutions table and loops through each assignment returned, then stores each to the assignments table.

The route takes two parameters, the team_id which is passed from the current user team_id and the `week_id` which needs to be specified as the solution will be created dependant on the week. Conditions are placed to check the user and retrieve the user's information, while checking if the user has the required role of Employer. The create_schedule function is initiated by passing the `team_id`, `week_id` and the database, which is stored in a variable called `request_data.`



```
@router.post("/assign-shifts/{team_id}/{week_id}",
response_model=ShiftAssignmentResponse)
async def assign_shifts(
    team_id: int,
    week_id: int,
    db: Session = Depends(get_db),
    current_user: User = Depends(require_role([
"Employer"]))
):
    # auth
    if current_user.team_id != team_id:
        raise HTTPException(status_code=403, detail=
"You are not authorized to assign shifts for this team"
)
```

Figure 86, CSP Route #1

The solver is initialised by passing all relevant details from request_data to the CSP constructor, which is stored in the solver instance. The CSP logic is summarised in the solve method within the solver, which is called within the route. The result from the solve method is stored in the result instance, this instance has conditions to check if the solver has no assignments to then return a message response, “no valid assignment found”.

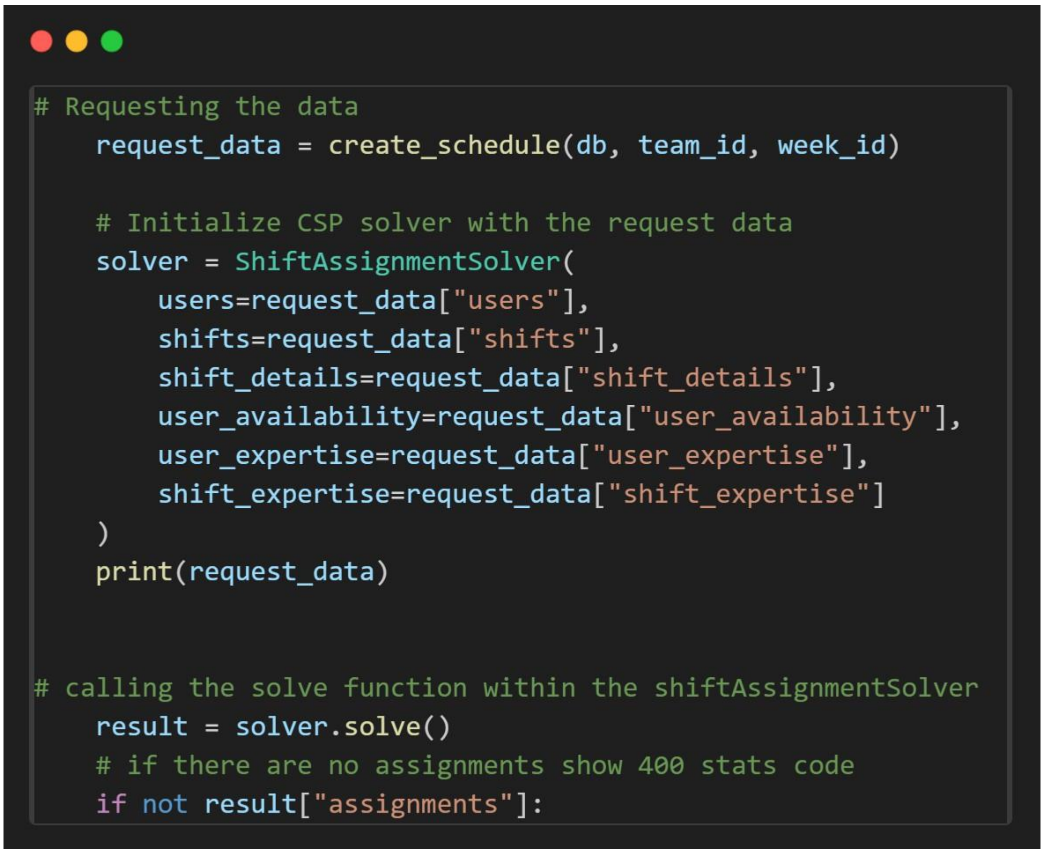
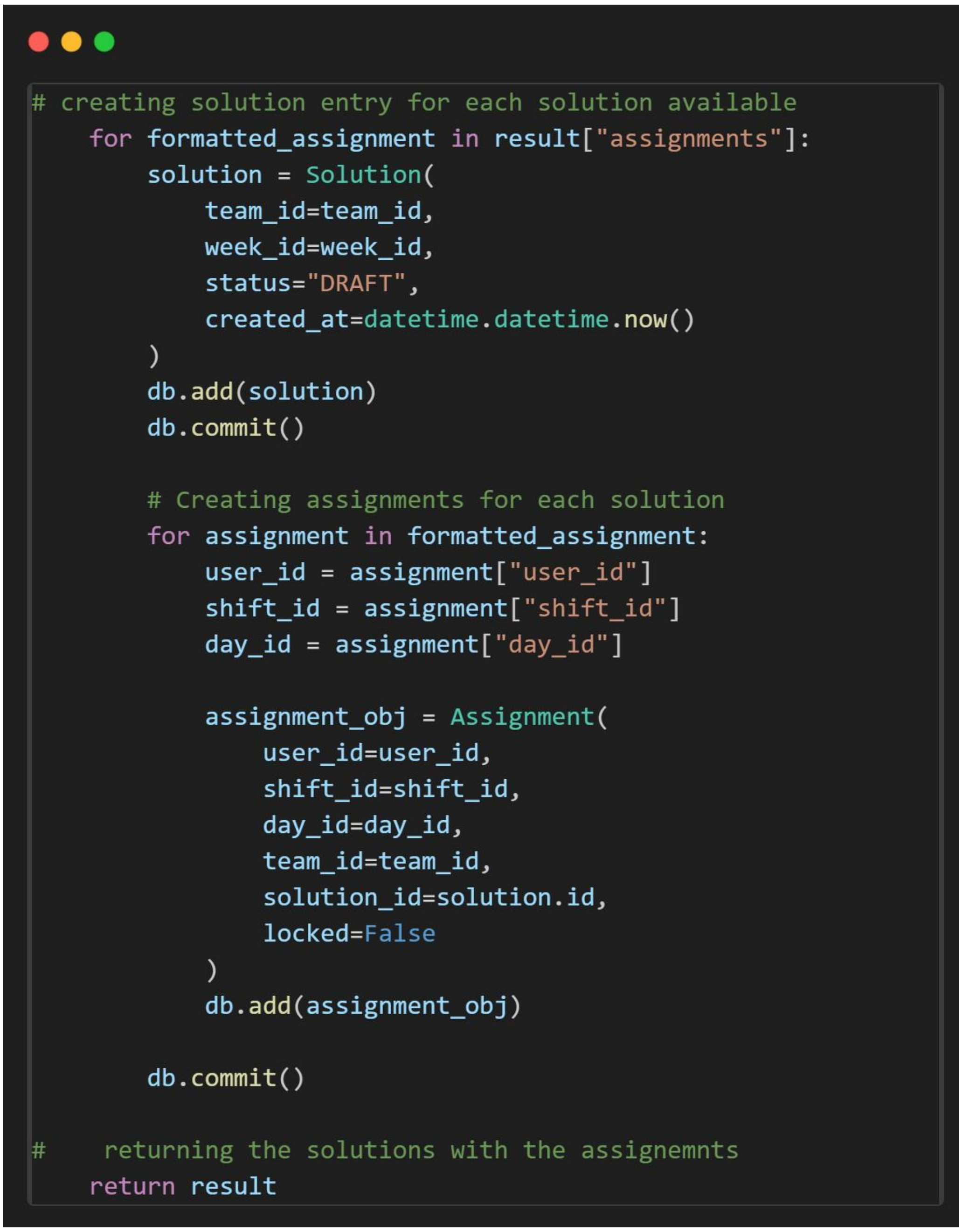

```python
# Requesting the data
    request_data = create_schedule(db, team_id, week_id)

    # Initialize CSP solver with the request data
    solver = ShiftAssignmentSolver(
        users=request_data["users"],
        shifts=request_data["shifts"],
        shift_details=request_data["shift_details"],
        user_availability=request_data["user_availability"],
        user_expertise=request_data["user_expertise"],
        shift_expertise=request_data["shift_expertise"]
    )
    print(request_data)


# calling the solve function within the shiftAssignmentSolver
    result = solver.solve()
    # if there are no assignments show 400 stats code
    if not result["assignments"]:
```

Figure 87, CSP Route #2

If there are no errors, the results which are stored in the result instance are formatted to populate the solution table with relevant data. The assignments are placed within a loop to pass the relevant details for each into the assignments table, which then displays the return of the result.



```python
# creating solution entry for each solution available
    for formatted_assignment in result["assignments"]:
        solution = Solution(
            team_id=team_id,
            week_id=week_id,
            status="DRAFT",
            created_at=datetime.datetime.now()
        )
        db.add(solution)
        db.commit()

        # Creating assignments for each solution
        for assignment in formatted_assignment:
            user_id = assignment["user_id"]
            shift_id = assignment["shift_id"]
            day_id = assignment["day_id"]

            assignment_obj = Assignment(
                user_id=user_id,
                shift_id=shift_id,
                day_id=day_id,
                team_id=team_id,
                solution_id=solution.id,
                locked=False
            )
            db.add(assignment_obj)

        db.commit()

#   returning the solutions with the assignemnts
    return result
```

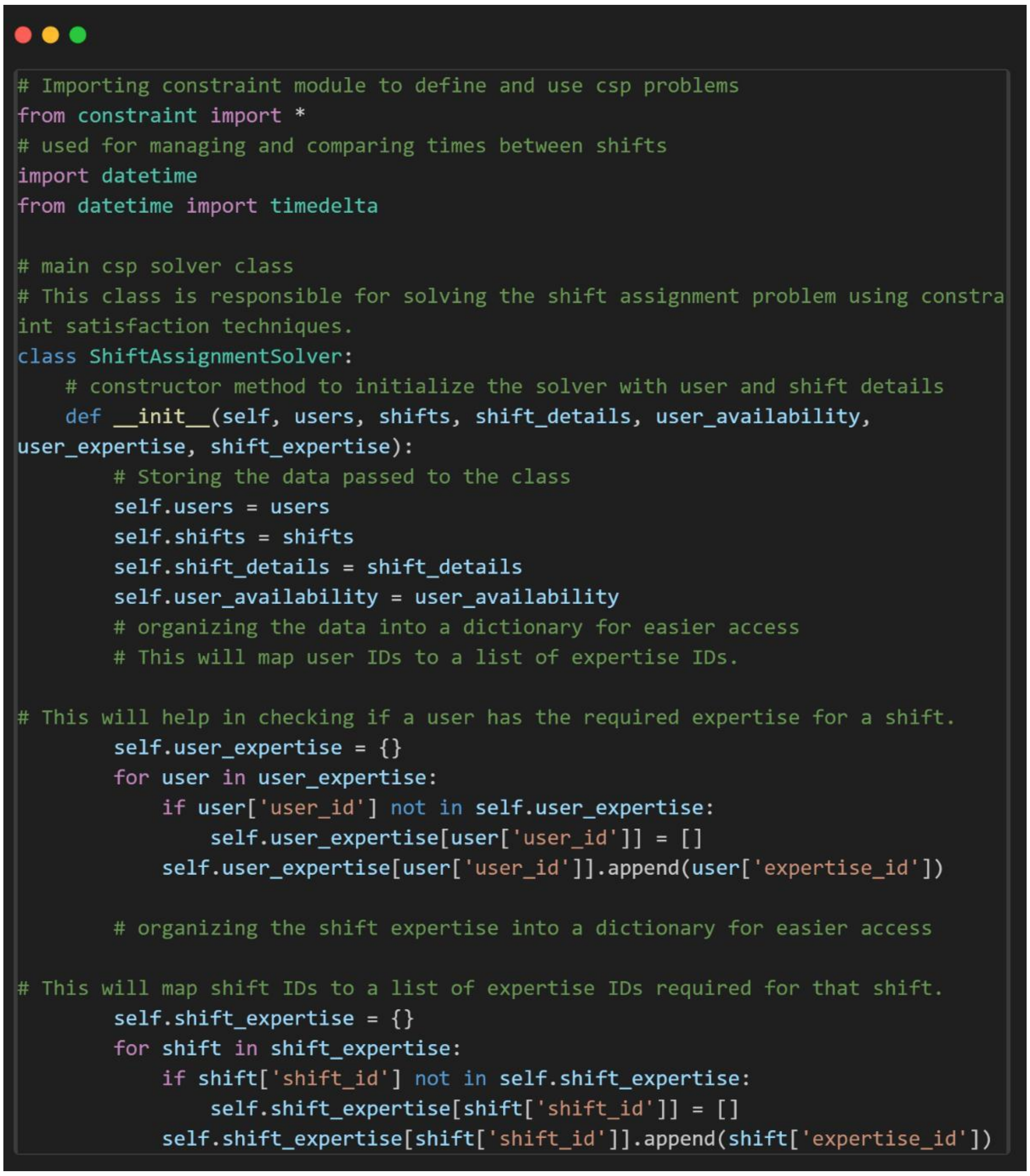
Figure 88, CSP Route #3

5.5.4 Solver

5.5.4.1 Init function

The ShiftAssignmentSolver class encapsulates the core logic for assigning users to shifts. The class uses the python constraint library to define variables, domains and constraints. The library also offers various methods which allow to develop the CSP.

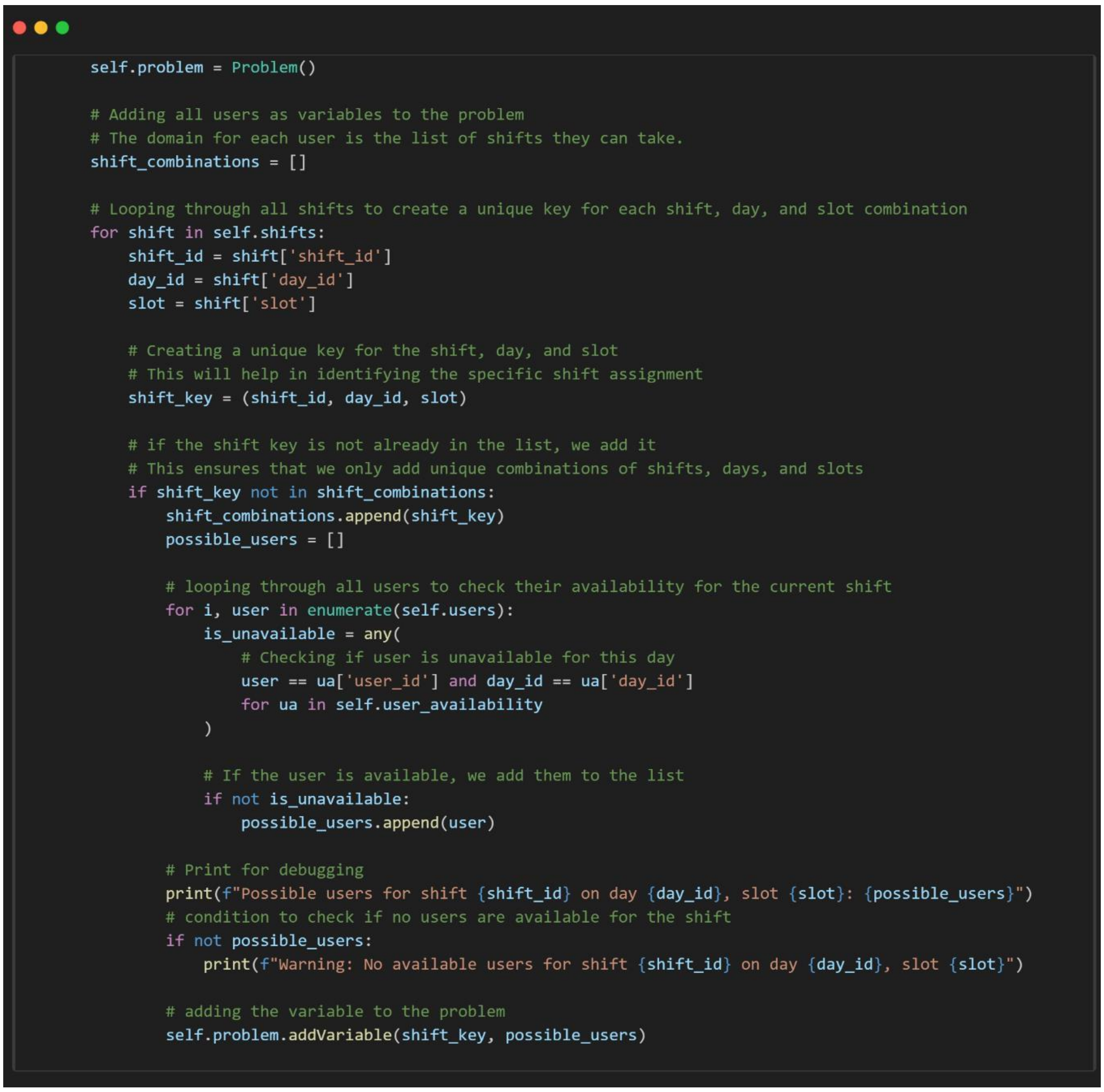
As shown in the figure below, the solver is initialised within the `__init__` function. This function takes in various parameters which is passed from the route within its constructor, such as `shifts`, `users`, and `user_availability`. Some of the lists which are passed are converted into dictionaries to enable more efficient access during constraint checking, this is done for `user_expertise` and `shift_expertise`.



```python
# Importing constraint module to define and use csp problems
from constraint import *
# used for managing and comparing times between shifts
import datetime
from datetime import timedelta

# main csp solver class
# This class is responsible for solving the shift assignment problem using constra
int satisfaction techniques.
class ShiftAssignmentSolver:
    # constructor method to initialize the solver with user and shift details
    def __init__(self, users, shifts, shift_details, user_availability,
user_expertise, shift_expertise):
        # Storing the data passed to the class
        self.users = users
        self.shifts = shifts
        self.shift_details = shift_details
        self.user_availability = user_availability
        # organizing the data into a dictionary for easier access
        # This will map user IDs to a list of expertise IDs.

# This will help in checking if a user has the required expertise for a shift.
        self.user_expertise = {}
        for user in user_expertise:
            if user['user_id'] not in self.user_expertise:
                self.user_expertise[user['user_id']] = []
            self.user_expertise[user['user_id']].append(user['expertise_id'])

        # organizing the shift expertise into a dictionary for easier access

# This will map shift IDs to a list of expertise IDs required for that shift.
        self.shift_expertise = {}
        for shift in shift_expertise:
            if shift['shift_id'] not in self.shift_expertise:
                self.shift_expertise[shift['shift_id']] = []
            self.shift_expertise[shift['shift_id']].append(shift['expertise_id'])
```

Figure 89, Init Function #1

The problem is than initialised using the problem method which is provided by python constraint. A list is defined called `shift_combinations` to store the unique shift combinations then a list is defined called `possible_users` to define the domain of each shift. The list of possible users is specified by the user's availability which was defined in the constructor. If a user is available for a shift, the python constraint `addVariable` method is called with the shift and the user. The function is then closed with the addConstraints function, which is adds all constraints which are created to the variables.



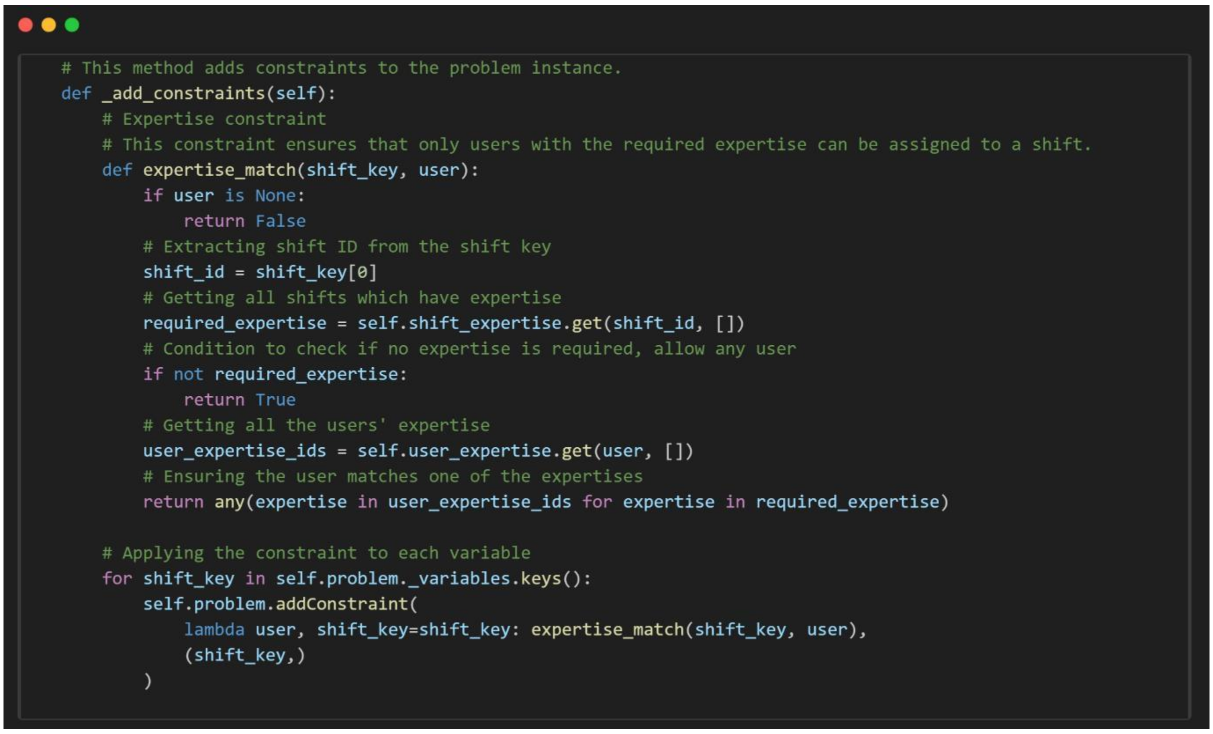
```python
        self.problem = Problem()

        # Adding all users as variables to the problem
        # The domain for each user is the list of shifts they can take.
        shift_combinations = []

        # Looping through all shifts to create a unique key for each shift, day, and slot combination
        for shift in self.shifts:
            shift_id = shift['shift_id']
            day_id = shift['day_id']
            slot = shift['slot']

            # Creating a unique key for the shift, day, and slot
            # This will help in identifying the specific shift assignment
            shift_key = (shift_id, day_id, slot)

            # if the shift key is not already in the list, we add it
            # This ensures that we only add unique combinations of shifts, days, and slots
            if shift_key not in shift_combinations:
                shift_combinations.append(shift_key)
                possible_users = []

                # looping through all users to check their availability for the current shift
                for i, user in enumerate(self.users):
                    is_unavailable = any(
                        # Checking if user is unavailable for this day
                        user == ua['user_id'] and day_id == ua['day_id']
                        for ua in self.user_availability
                    )

                    # If the user is available, we add them to the list
                    if not is_unavailable:
                        possible_users.append(user)

                # Print for debugging
                print(f"Possible users for shift {shift_id} on day {day_id}, slot {slot}: {possible_users}")
                # condition to check if no users are available for the shift
                if not possible_users:
                    print(f"Warning: No available users for shift {shift_id} on day {day_id}, slot {slot}")

                # adding the variable to the problem
                self.problem.addVariable(shift_key, possible_users)
```

Figure 90, Init Function #2

5.5.4.2 Constraints Function

The solvers main responsibility was to assign shifts to users without breaking any constraints. Some constraints were created from the data that was passed such as expertise. However, some global constraints needed to be added to prevent a single user being assigned all shifts.

The constraints are all defined within the `_add_constraints` function, which takes the self-parameter specifying the current data. The first constraint checks if there is expertise relationship between the shift and expertise, if there is an expertise attached to the shift then it does the same with the user. It uses the python constraint method, `addConstraint`, to match the expertise's between shift and user. Any assignments that don't match are removed from the assignments list selection. Global constraints were implemented to ensure fair and logical shift distribution across users, which were applied for the whole problem space.

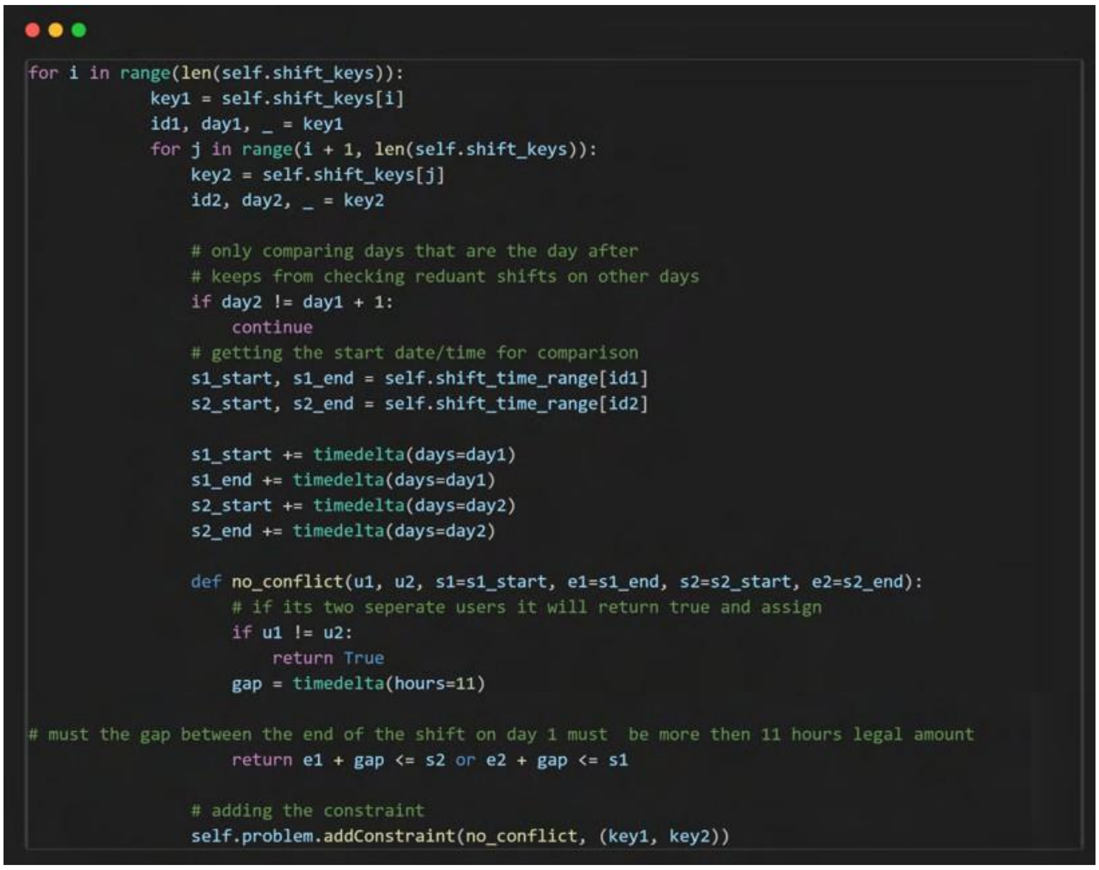


```python
# This method adds constraints to the problem instance.
def _add_constraints(self):
    # Expertise constraint
    # This constraint ensures that only users with the required expertise can be assigned to a shift.
    def expertise_match(shift_key, user):
        if user is None:
            return False
        # Extracting shift ID from the shift key
        shift_id = shift_key[0]
        # Getting all shifts which have expertise
        required_expertise = self.shift_expertise.get(shift_id, [])
        # Condition to check if no expertise is required, allow any user
        if not required_expertise:
            return True
        # Getting all the users' expertise
        user_expertise_ids = self.user_expertise.get(user, [])
        # Ensuring the user matches one of the expertises
        return any(expertise in user_expertise_ids for expertise in required_expertise)

    # Applying the constraint to each variable
    for shift_key in self.problem._variables.keys():
        self.problem.addConstraint(
            lambda user, shift_key=shift_key: expertise_match(shift_key, user),
            (shift_key,)
        )
```

Figure 91, Constraint function #1

The `no_conflict` and `max_day_constraint` functions enforced mandatory rest period for users between shifts. The `no_conflict` was responsible for checking which shifts were compatible with each other. The `max_day_constraint` checked how many days each user has worked, ensuring no single user is assigned all the shifts.

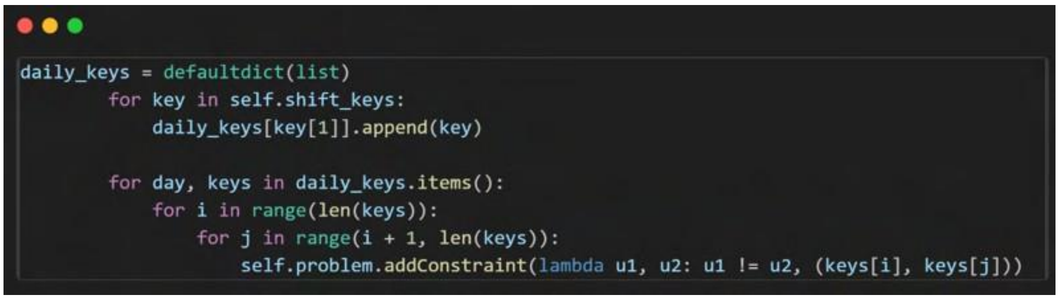
Checking which shifts were compatible could become very computational especially for larger problems. Optimisation was implemented to make sure that the checks were only made with relevant shifts. Relevant shifts were seen as shifts that are the next day from the current shift in the loop and used a condition to check if the shifts have a gap of eleven hours between one another, eleven hours is legal rest requirement within Ireland. A nested loop was implemented to take the start and end time of both shifts only if the second shift is a day ahead. These shifts are placed into the no conflict function, where it checks the two shifts' users and if they don't match then it can continue. If the user is the same on both shifts it checks if the time gap is more eleven hours.



```
for i in range(len(self.shift_keys)):
            key1 = self.shift_keys[i]
            id1, day1, _ = key1
            for j in range(i + 1, len(self.shift_keys)):
                key2 = self.shift_keys[j]
                id2, day2, _ = key2

                # only comparing days that are the day after
                # keeps from checking reduant shifts on other days
                if day2 != day1 + 1:
                    continue
                # getting the start date/time for comparison
                s1_start, s1_end = self.shift_time_range[id1]
                s2_start, s2_end = self.shift_time_range[id2]

                s1_start += timedelta(days=day1)
                s1_end += timedelta(days=day1)
                s2_start += timedelta(days=day2)
                s2_end += timedelta(days=day2)

                def no_conflict(u1, u2, s1=s1_start, e1=s1_end, s2=s2_start, e2=s2_end):
                    # if its two seperate users it will return true and assign
                    if u1 != u2:
                        return True
                    gap = timedelta(hours=11)

# must the gap between the end of the shift on day 1 must  be more then 11 hours legal amount
                    return e1 + gap <= s2 or e2 + gap <= s1

                # adding the constraint
                self.problem.addConstraint(no_conflict, (key1, key2))
```

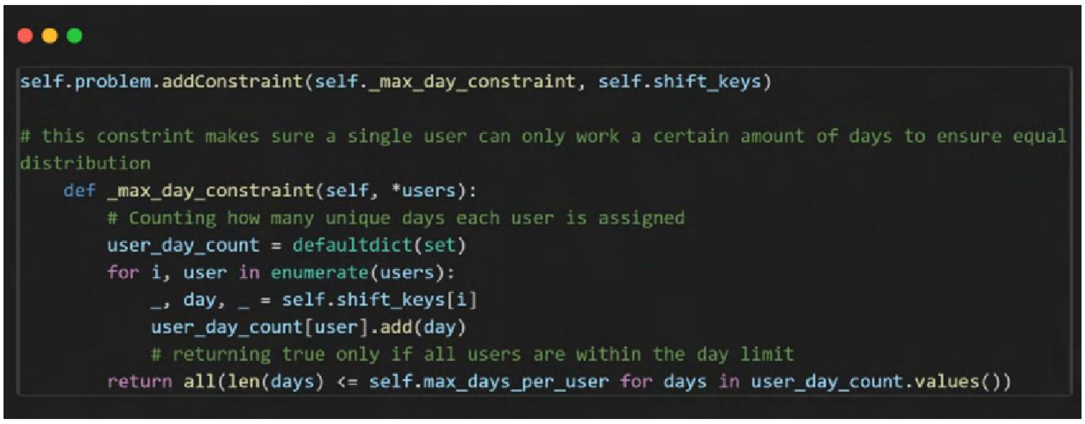
Figure 92, No Conflict Function

Additionally, another function was called using the data, which was gathered from the loop. This function put each shift into a separate dictionary depending on the day. This would place constraints against each shift which have share the same day, ensuring that a user is not placed on two shifts on a single day.



```
daily_keys = defaultdict(list)
        for key in self.shift_keys:
            daily_keys[key[1]].append(key)

        for day, keys in daily_keys.items():
            for i in range(len(keys)):
                for j in range(i + 1, len(keys)):
                    self.problem.addConstraint(lambda u1, u2: u1 != u2, (keys[i], keys[j]))
```

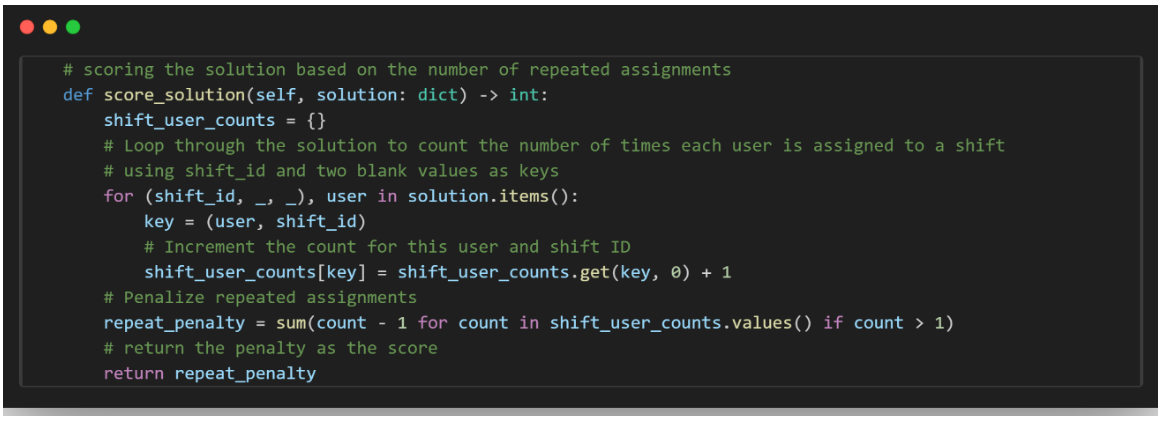
The `max_day_constraint` takes the assignments from each user and the length of assignments each user has. It conditions to make sure the user doesn't have more then five assignments. This was hardcoded to ensure no user works more than five days a week. For future implementation this can be set within the team table, so each user has their own max days they can work or implement it using hours.



```python
self.problem.addConstraint(self._max_day_constraint, self.shift_keys)

# this constrint makes sure a single user can only work a certain amount of days to ensure equal
distribution
    def _max_day_constraint(self, *users):
        # Counting how many unique days each user is assigned
        user_day_count = defaultdict(set)
        for i, user in enumerate(users):
            _, day, _ = self.shift_keys[i]
            user_day_count[user].add(day)
            # returning true only if all users are within the day limit
        return all(len(days) <= self.max_days_per_user for days in user_day_count.values())
```

Figure 93, Max Day Constraint

Additionally, a function was implemented to rank solutions based on the fairness of shift distribution. The `score_solution` method looped through each solution and tracks how many times a user is assigned to the same shift_id. If a user appears multiple times for the same shift across multiple days, a penalty point is assigned for each repetition. The lower the score, the less unfair the roster becomes in theory. The solutions return within the solve method are sorted by the best score solution, `scored_solutions = sorted (solutions, key=self. score_solution`. The Figure below illustrates how this function was created. This function had to be removed as it was too computational within generate and was only implemented within the regeneration solver.

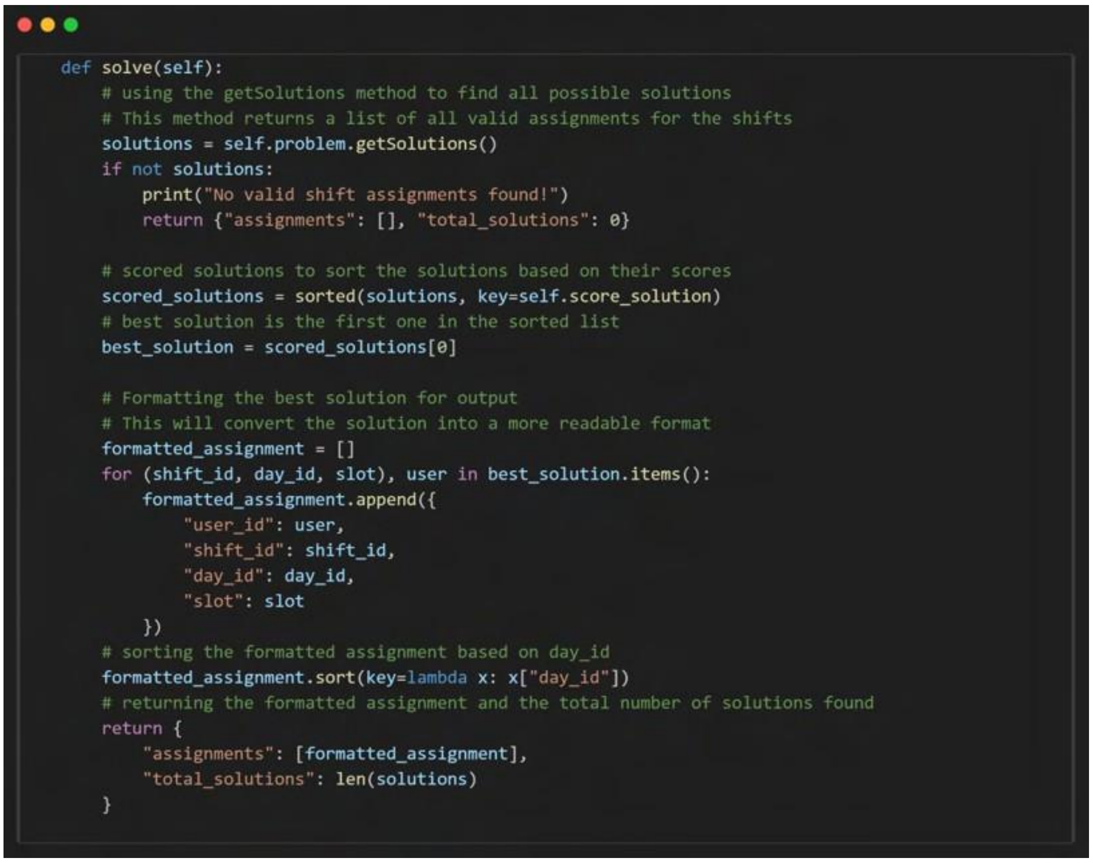


```python
# scoring the solution based on the number of repeated assignments
def score_solution(self, solution: dict) -> int:
    shift_user_counts = {}
    # Loop through the solution to count the number of times each user is assigned to a shift
    # using shift_id and two blank values as keys
    for (shift_id, _, _), user in solution.items():
        key = (user, shift_id)
        # Increment the count for this user and shift ID
        shift_user_counts[key] = shift_user_counts.get(key, 0) + 1
    # Penalize repeated assignments
    repeat_penalty = sum(count - 1 for count in shift_user_counts.values() if count > 1)
    # return the penalty as the score
    return repeat_penalty
```

Figure 94, Scoring Solutions Function

5.5.4.3 Solving Function

The solve method gathered the specific problems constraints and assignments which were defined. The solve method initialised by using the python constraint method called `getsolutions` to gather all existing solutions. If solutions existed, they would be formatted and filtered by `day_id` to show the assignments in order of each solution beginning with Monday or the first day shifts beginning with. The response also provides the number of solutions which are available and if no solutions are available the method provides an error that no solutions were found and why. The figure below illustrates how it was implemented within the code.



```python
    def solve(self):
        # using the getSolutions method to find all possible solutions
        # This method returns a list of all valid assignments for the shifts
        solutions = self.problem.getSolutions()
        if not solutions:
            print("No valid shift assignments found!")
            return {"assignments": [], "total_solutions": 0}

        # scored solutions to sort the solutions based on their scores
        scored_solutions = sorted(solutions, key=self.score_solution)
        # best solution is the first one in the sorted list
        best_solution = scored_solutions[0]

        # Formatting the best solution for output
        # This will convert the solution into a more readable format
        formatted_assignment = []
        for (shift_id, day_id, slot), user in best_solution.items():
            formatted_assignment.append({
                "user_id": user,
                "shift_id": shift_id,
                "day_id": day_id,
                "slot": slot
            })
        # sorting the formatted assignment based on day_id
        formatted_assignment.sort(key=lambda x: x["day_id"])
        # returning the formatted assignment and the total number of solutions found
        return {
            "assignments": [formatted_assignment],
            "total_solutions": len(solutions)
        }
```

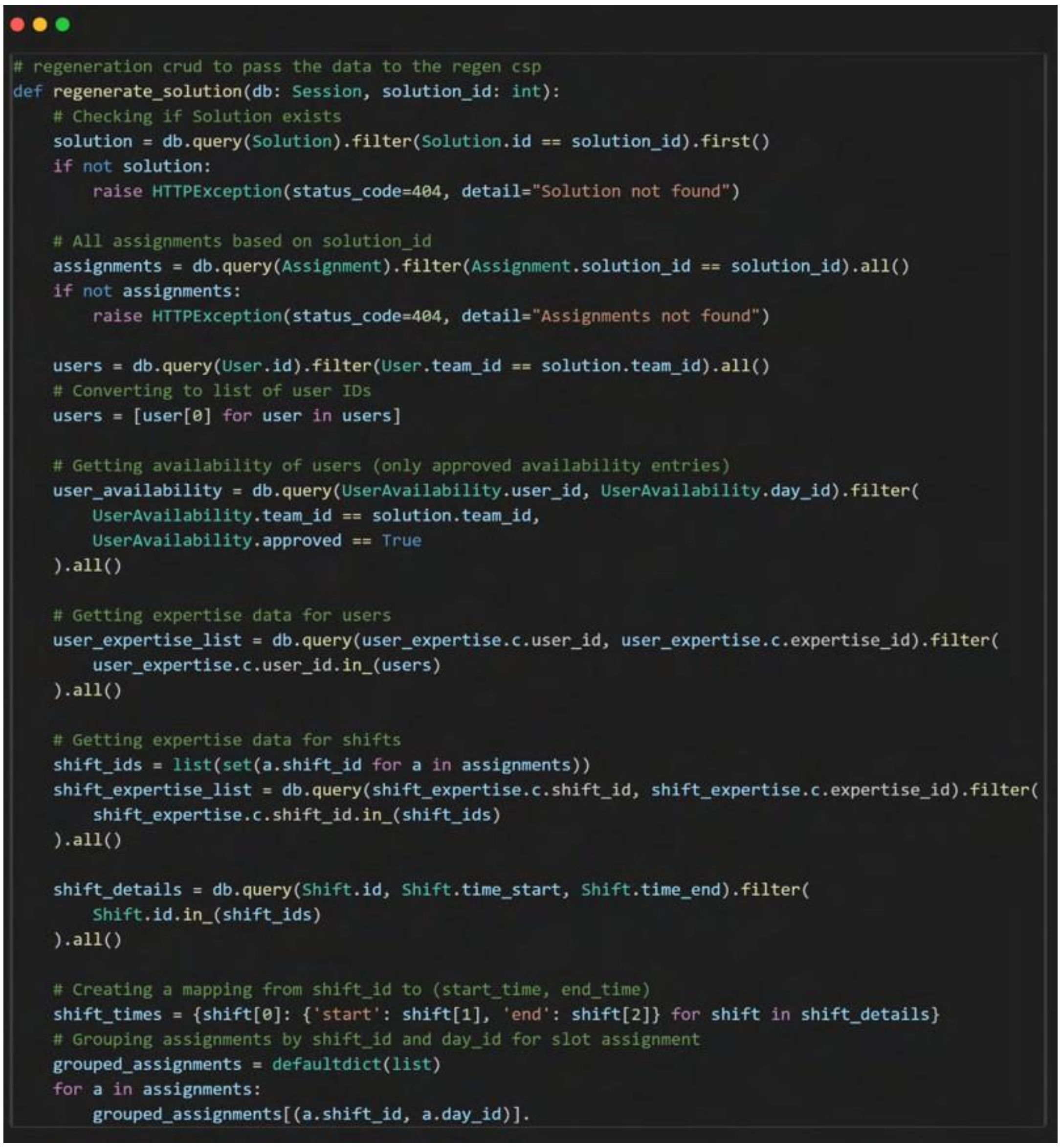
Figure 95, Solve Method

5.5.5 Regeneration Solver

The regeneration solver is a separate CSP solver class, which focuses on using existing solutions and changing certain assignments dependent on additional constraints. This solver allows the employer to have more flexibility when choosing their schedule, by locking certain assignments and regenerating them. The following sections provide an understanding of how this regeneration roster was implemented and the difference between the two CSPs.

5.5.5.1 CRUD

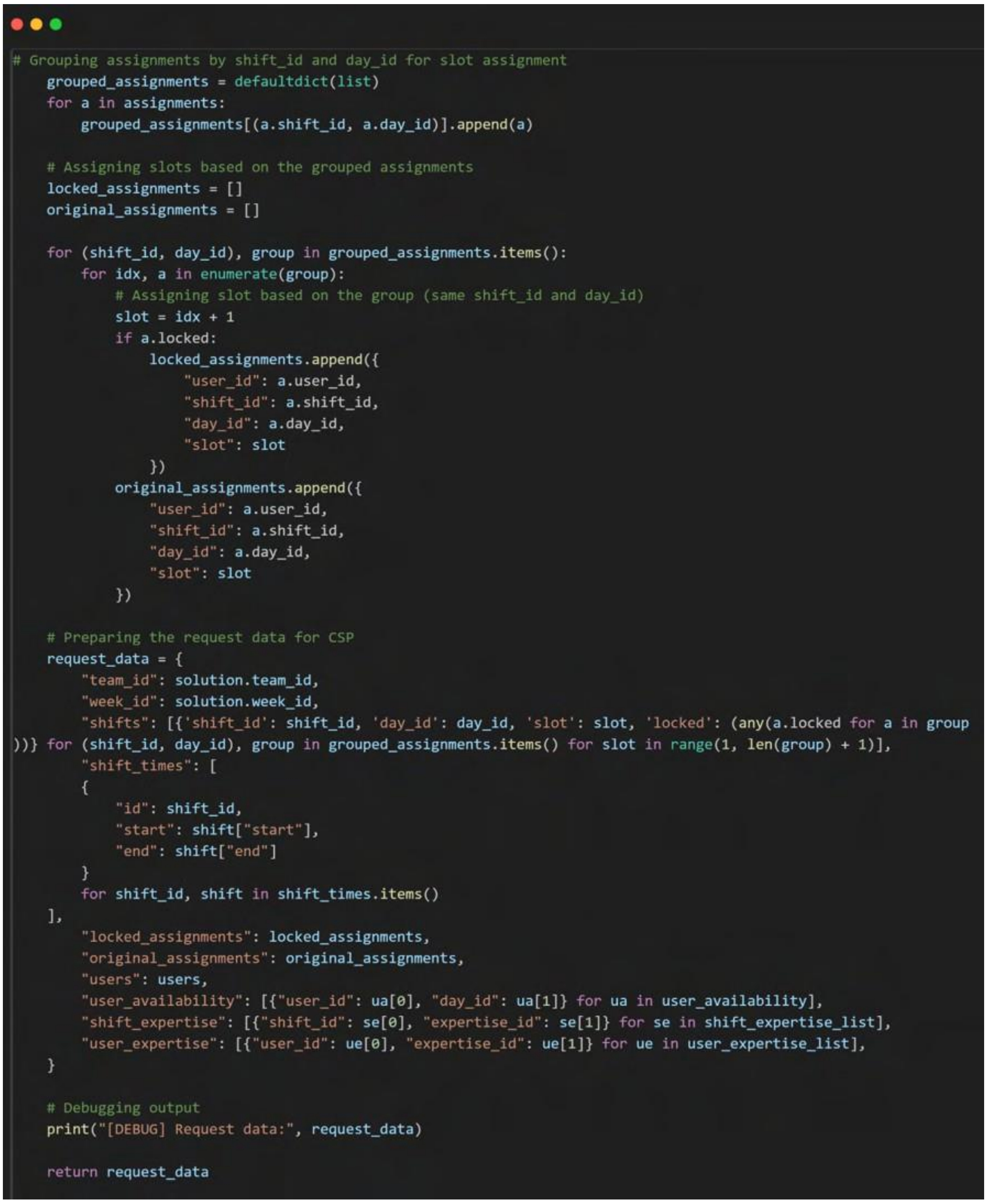
This CSPs logic was dependent on the solution provided. As each table within the database is connected to the id of the team, it becomes easy to gather all the correct data in relation to the solution and the team connected to the solution.



```python
# regeneration crud to pass the data to the regen csp
def regenerate_solution(db: Session, solution_id: int):
    # Checking if Solution exists
    solution = db.query(Solution).filter(Solution.id == solution_id).first()
    if not solution:
        raise HTTPException(status_code=404, detail="Solution not found")

    # All assignments based on solution_id
    assignments = db.query(Assignment).filter(Assignment.solution_id == solution_id).all()
    if not assignments:
        raise HTTPException(status_code=404, detail="Assignments not found")

    users = db.query(User.id).filter(User.team_id == solution.team_id).all()
    # Converting to list of user IDs
    users = [user[0] for user in users]

    # Getting availability of users (only approved availability entries)
    user_availability = db.query(UserAvailability.user_id, UserAvailability.day_id).filter(
        UserAvailability.team_id == solution.team_id,
        UserAvailability.approved == True
    ).all()

    # Getting expertise data for users
    user_expertise_list = db.query(user_expertise.c.user_id, user_expertise.c.expertise_id).filter(
        user_expertise.c.user_id.in_(users)
    ).all()

    # Getting expertise data for shifts
    shift_ids = list(set(a.shift_id for a in assignments))
    shift_expertise_list = db.query(shift_expertise.c.shift_id, shift_expertise.c.expertise_id).filter(
        shift_expertise.c.shift_id.in_(shift_ids)
    ).all()

    shift_details = db.query(Shift.id, Shift.time_start, Shift.time_end).filter(
        Shift.id.in_(shift_ids)
    ).all()

    # Creating a mapping from shift_id to (start_time, end_time)
    shift_times = {shift[0]: {'start': shift[1], 'end': shift[2]} for shift in shift_details}
    # Grouping assignments by shift_id and day_id for slot assignment
    grouped_assignments = defaultdict(list)
    for a in assignments:
        grouped_assignments[(a.shift_id, a.day_id)].
```

Figure 96, Regenerate Crud #1

Unlike the previous CSP, it allows gathers data about each assignment specific to the solution, dividing the assignments into groups of locked and unlocked. This simplifies prefiling the solution within CSP and only changing the assignments which are unlocked.

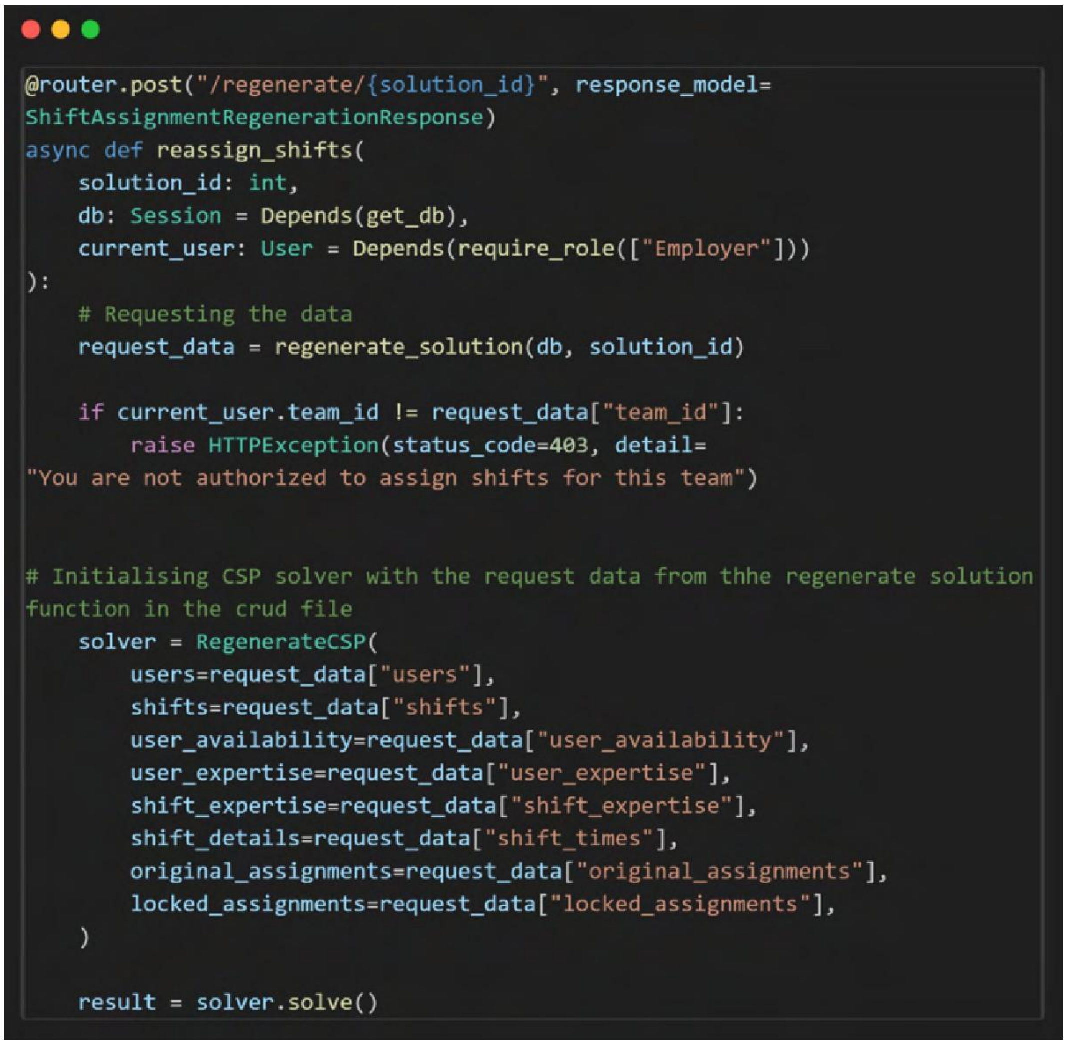


```python
# Grouping assignments by shift_id and day_id for slot assignment
    grouped_assignments = defaultdict(list)
    for a in assignments:
        grouped_assignments[(a.shift_id, a.day_id)].append(a)

    # Assigning slots based on the grouped assignments
    locked_assignments = []
    original_assignments = []

    for (shift_id, day_id), group in grouped_assignments.items():
        for idx, a in enumerate(group):
            # Assigning slot based on the group (same shift_id and day_id)
            slot = idx + 1
            if a.locked:
                locked_assignments.append({
                    "user_id": a.user_id,
                    "shift_id": a.shift_id,
                    "day_id": a.day_id,
                    "slot": slot
                })
            original_assignments.append({
                "user_id": a.user_id,
                "shift_id": a.shift_id,
                "day_id": a.day_id,
                "slot": slot
            })

    # Preparing the request data for CSP
    request_data = {
        "team_id": solution.team_id,
        "week_id": solution.week_id,
        "shifts": [{'shift_id': shift_id, 'day_id': day_id, 'slot': slot, 'locked': (any(a.locked for a in group
))} for (shift_id, day_id), group in grouped_assignments.items() for slot in range(1, len(group) + 1)],
        "shift_times": [
        {
            "id": shift_id,
            "start": shift["start"],
            "end": shift["end"]
        }
        for shift_id, shift in shift_times.items()
    ],
        "locked_assignments": locked_assignments,
        "original_assignments": original_assignments,
        "users": users,
        "user_availability": [{"user_id": ua[0], "day_id": ua[1]} for ua in user_availability],
        "shift_expertise": [{"shift_id": se[0], "expertise_id": se[1]} for se in shift_expertise_list],
        "user_expertise": [{"user_id": ue[0], "expertise_id": ue[1]} for ue in user_expertise_list],
    }

    # Debugging output
    print("[DEBUG] Request data:", request_data)

    return request_data
```

Figure 97, Regenerate Crud #2

5.5.5.2 Route

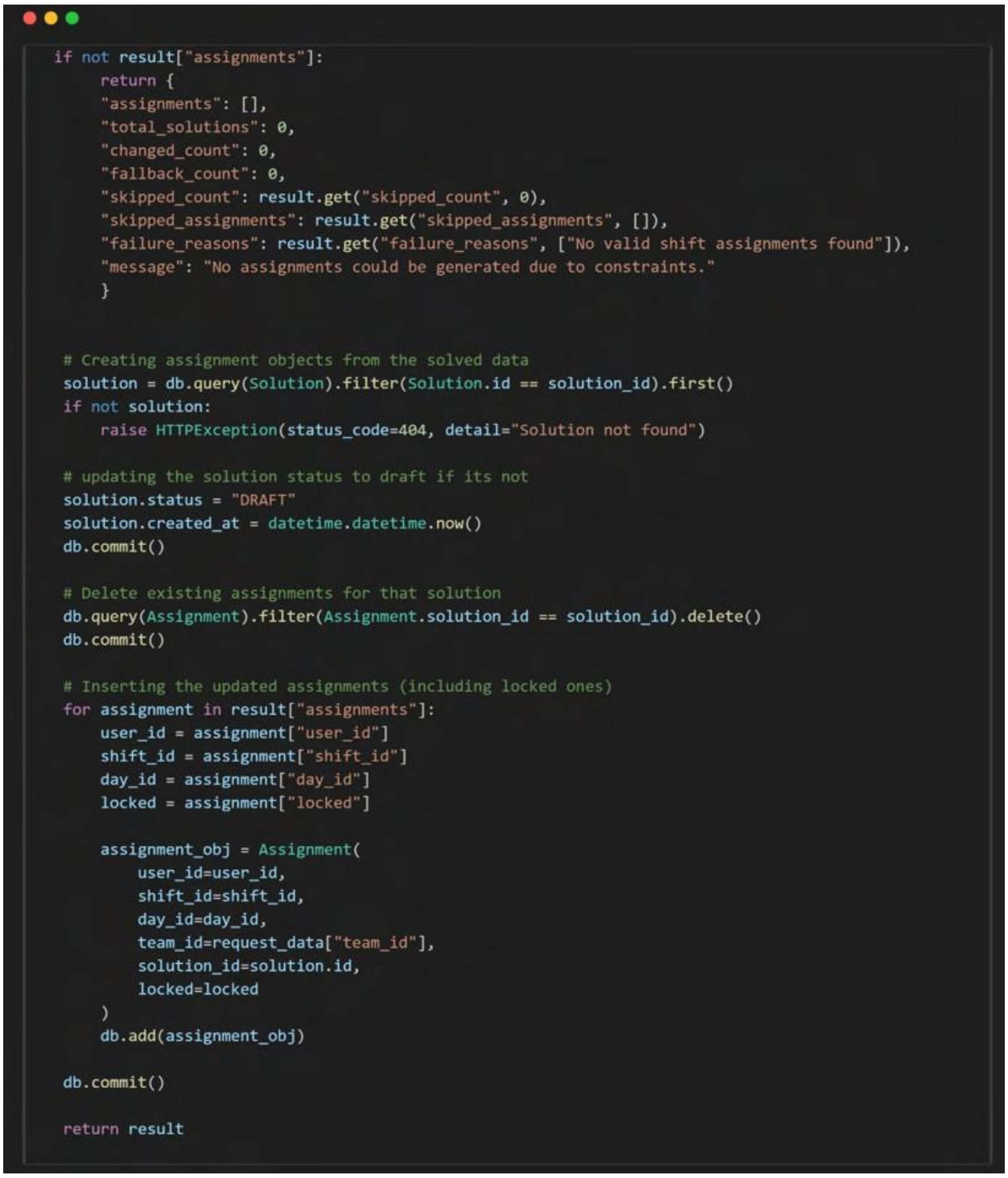
The route holds the solution id parameter which specifies which solution will be regenerated. This solution is passed to the `generate_solution` function which was created in the crud file. The response from this function is then stored in a variable called `request_data`, this data is then passed to the regenerate CSP. The solve function is then called to activate the CSP solver.



```
@router.post("/regenerate/{solution_id}", response_model=
ShiftAssignmentRegenerationResponse)
async def reassign_shifts(
    solution_id: int,
    db: Session = Depends(get_db),
    current_user: User = Depends(require_role(["Employer"]))
):
    # Requesting the data
    request_data = regenerate_solution(db, solution_id)

    if current_user.team_id != request_data["team_id"]:
        raise HTTPException(status_code=403, detail=
"You are not authorized to assign shifts for this team")


# Initialising CSP solver with the request data from thhe regenerate solution
function in the crud file
    solver = RegenerateCSP(
        users=request_data["users"],
        shifts=request_data["shifts"],
        user_availability=request_data["user_availability"],
        user_expertise=request_data["user_expertise"],
        shift_expertise=request_data["shift_expertise"],
        shift_details=request_data["shift_times"],
        original_assignments=request_data["original_assignments"],
        locked_assignments=request_data["locked_assignments"],
    )

    result = solver.solve()
```

Figure 98, Regenerate Route #1

Using this `result` variable, a condition is placed to check if there were any solutions found. If there was a solution, this solution is now placed now updated with the new solution data and each assignment corresponding with that previous solution is deleted and new assignments are assigned.

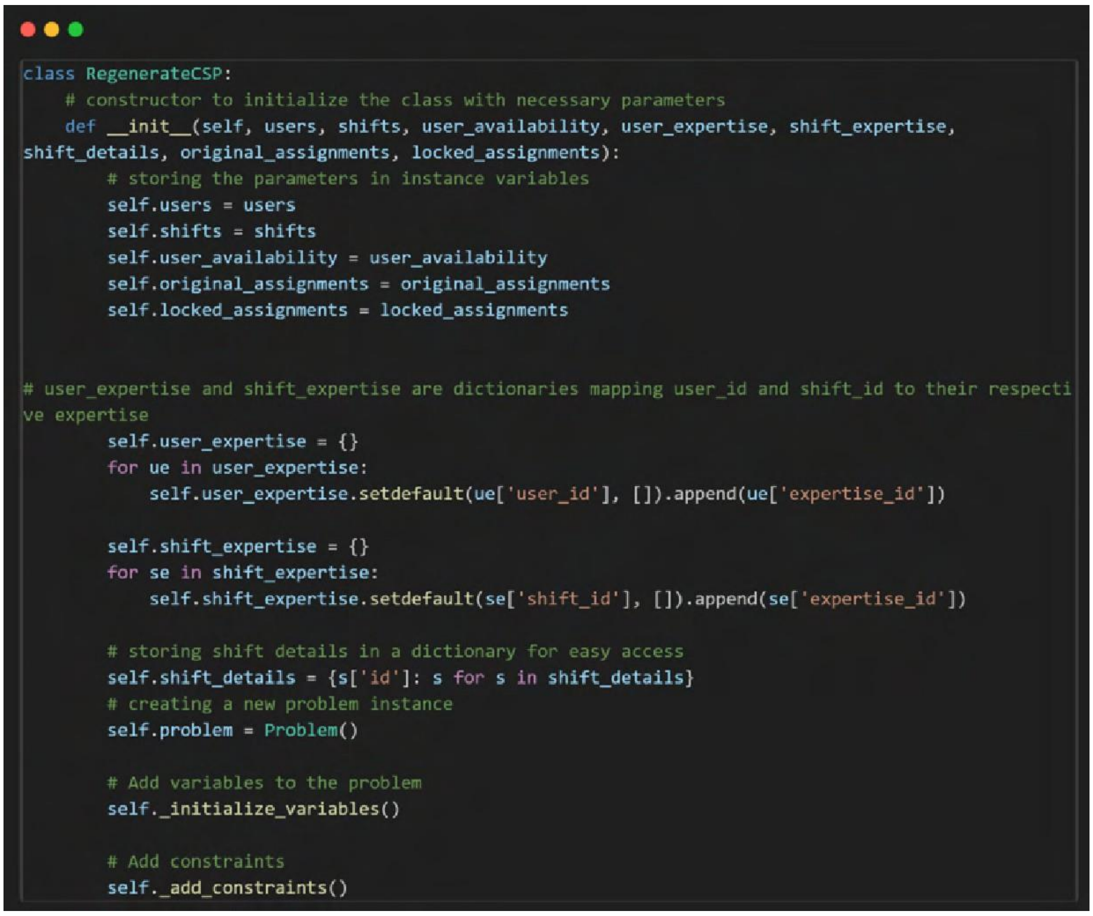


```python
if not result["assignments"]:
    return {
    "assignments": [],
    "total_solutions": 0,
    "changed_count": 0,
    "fallback_count": 0,
    "skipped_count": result.get("skipped_count", 0),
    "skipped_assignments": result.get("skipped_assignments", []),
    "failure_reasons": result.get("failure_reasons", ["No valid shift assignments found"]),
    "message": "No assignments could be generated due to constraints."
    }


# Creating assignment objects from the solved data
solution = db.query(Solution).filter(Solution.id == solution_id).first()
if not solution:
    raise HTTPException(status_code=404, detail="Solution not found")

# updating the solution status to draft if its not
solution.status = "DRAFT"
solution.created_at = datetime.datetime.now()
db.commit()

# Delete existing assignments for that solution
db.query(Assignment).filter(Assignment.solution_id == solution_id).delete()
db.commit()

# Inserting the updated assignments (including locked ones)
for assignment in result["assignments"]:
    user_id = assignment["user_id"]
    shift_id = assignment["shift_id"]
    day_id = assignment["day_id"]
    locked = assignment["locked"]

    assignment_obj = Assignment(
        user_id=user_id,
        shift_id=shift_id,
        day_id=day_id,
        team_id=request_data["team_id"],
        solution_id=solution.id,
        locked=locked
    )
    db.add(assignment_obj)

db.commit()

return result
```

Figure 99, Regenerate Route #2

5.5.5.3 Solver

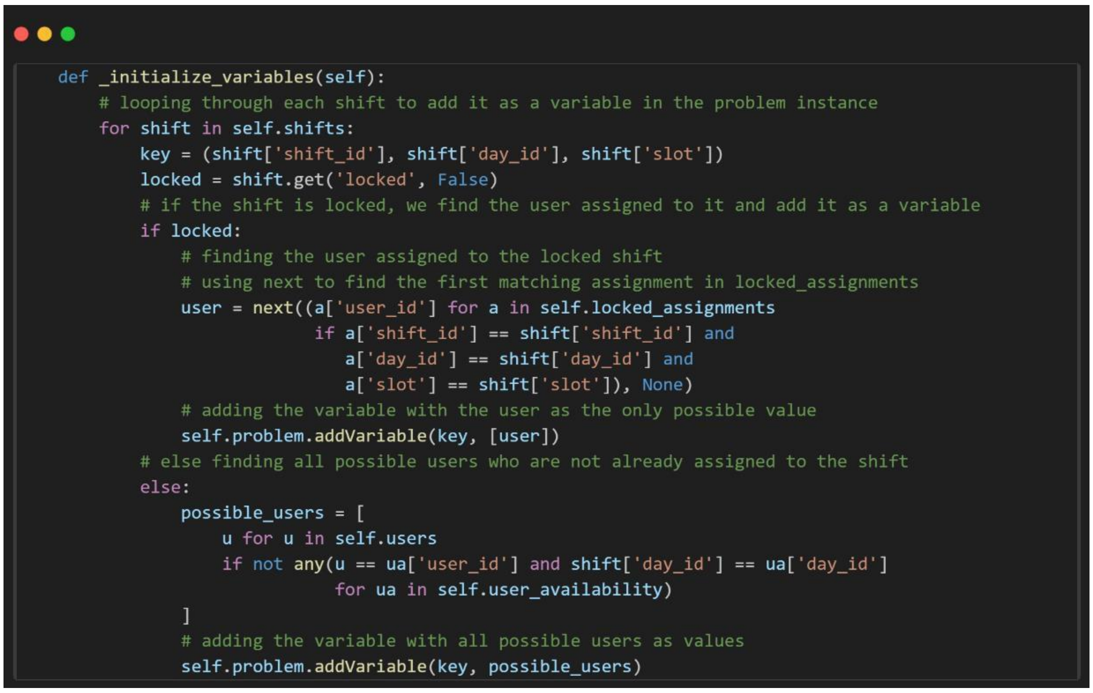
The regenerate solver is like the generation solver, it takes in the same variables within the constructor with additional locked and original assignment variables. Once the variables that were passed into the constructor are initialized, there are a few adjustments which are made to these variables. The user and shift expertise data are converted into dictionaries to allow the CSP to access these variables properly. The shift details are also changed to a dictionary to access easier. An instance of the problem is created, and the two main functions are added to the problem.



```
class RegenerateCSP:
    # constructor to initialize the class with necessary parameters
    def __init__(self, users, shifts, user_availability, user_expertise, shift_expertise,
shift_details, original_assignments, locked_assignments):
        # storing the parameters in instance variables
        self.users = users
        self.shifts = shifts
        self.user_availability = user_availability
        self.original_assignments = original_assignments
        self.locked_assignments = locked_assignments


# user_expertise and shift_expertise are dictionaries mapping user_id and shift_id to their respecti
ve expertise
        self.user_expertise = {}
        for ue in user_expertise:
            self.user_expertise.setdefault(ue['user_id'], []).append(ue['expertise_id'])

        self.shift_expertise = {}
        for se in shift_expertise:
            self.shift_expertise.setdefault(se['shift_id'], []).append(se['expertise_id'])

        # storing shift details in a dictionary for easy access
        self.shift_details = {s['id']: s for s in shift_details}
        # creating a new problem instance
        self.problem = Problem()

        # Add variables to the problem
        self._initialize_variables()

        # Add constraints
        self._add_constraints()
```

Figure 100, Regenerate solver #1

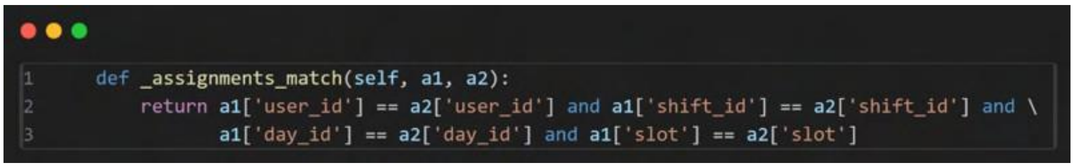
A new method has been created to add variables to the problem instance. This was done differently compared to the other CSP. `_initialize_variables` began with looping though the shifts which were already locked. If a shift was locked with would pass this assignment through to the solution and add a new variable. If an assignment was not locked it would clear the assignment and add the shift with a list of the possible users which are available.



```python
def _initialize_variables(self):
    # looping through each shift to add it as a variable in the problem instance
    for shift in self.shifts:
        key = (shift['shift_id'], shift['day_id'], shift['slot'])
        locked = shift.get('locked', False)
        # if the shift is locked, we find the user assigned to it and add it as a variable
        if locked:
            # finding the user assigned to the locked shift
            # using next to find the first matching assignment in locked_assignments
            user = next((a['user_id'] for a in self.locked_assignments
                         if a['shift_id'] == shift['shift_id'] and
                            a['day_id'] == shift['day_id'] and
                            a['slot'] == shift['slot']), None)
            # adding the variable with the user as the only possible value
            self.problem.addVariable(key, [user])
        # else finding all possible users who are not already assigned to the shift
        else:
            possible_users = [
                u for u in self.users
                if not any(u == ua['user_id'] and shift['day_id'] == ua['day_id']
                           for ua in self.user_availability)
            ]
            # adding the variable with all possible users as values
            self.problem.addVariable(key, possible_users)
```

Figure 101, Regenerate solver #2

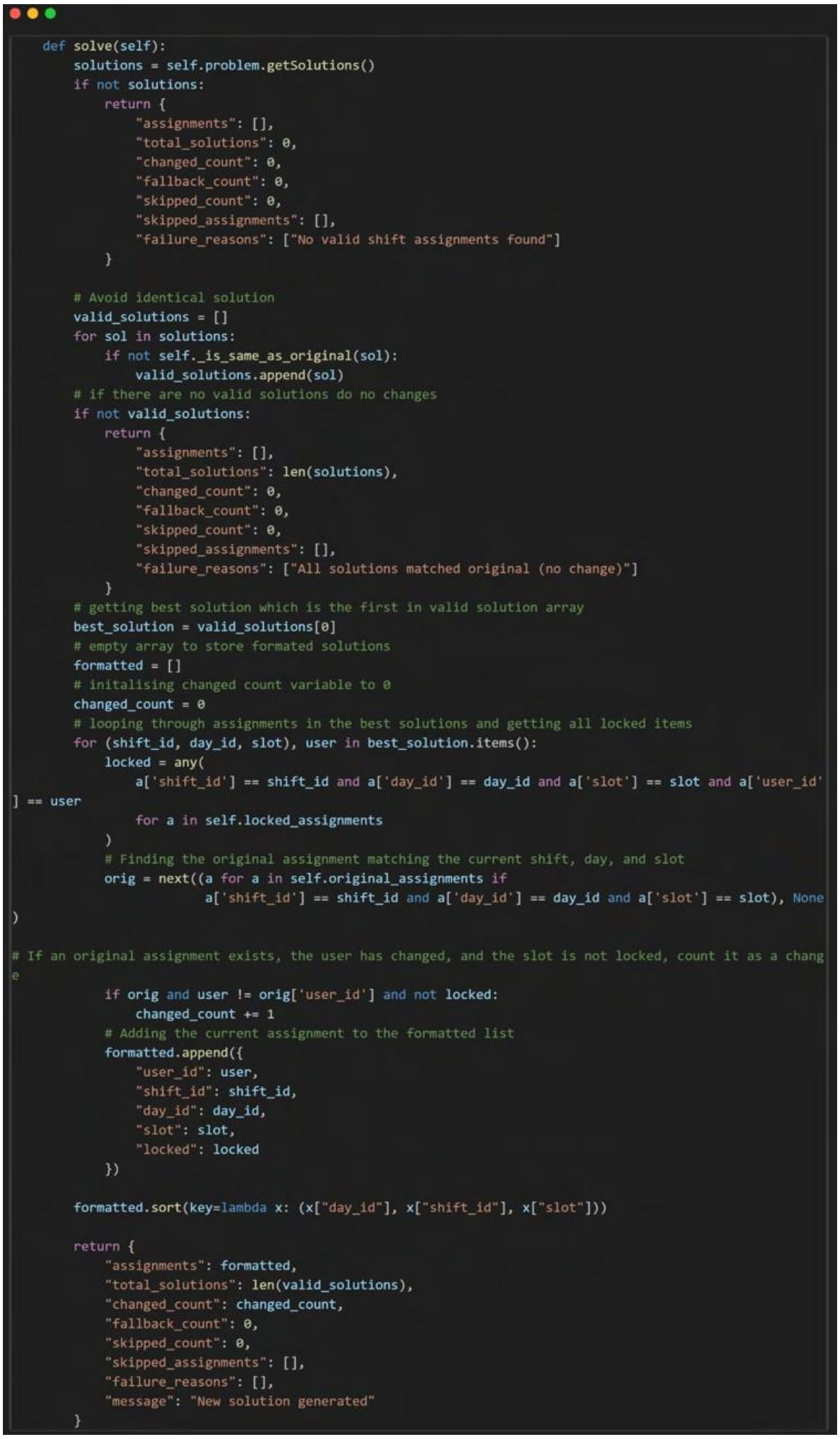
The constraints method adds various constraints to the problem, the constraints are identical to the previous CSP and can be seen on Constraints function, (5.6.4.2). However, an additional constraint is added to make sure that the previous solutions assignments are not repeated. It loops through the original assignments and the current assignments and checks if they are identical.



```python
def _assignments_match(self, a1, a2):
    return a1['user_id'] == a2['user_id'] and a1['shift_id'] == a2['shift_id'] and \
        a1['day_id'] == a2['day_id'] and a1['slot'] == a2['slot']
```

Figure 102, Regenerate Solver #3

The `solve` method gets three valid solutions if they are available. It has conditions to check if no solutions are available, it provides a message indicating why no new solutions were available. In the response, it also provides the `changed_count` which is the number of assignments which were changed in relation to the first solution.



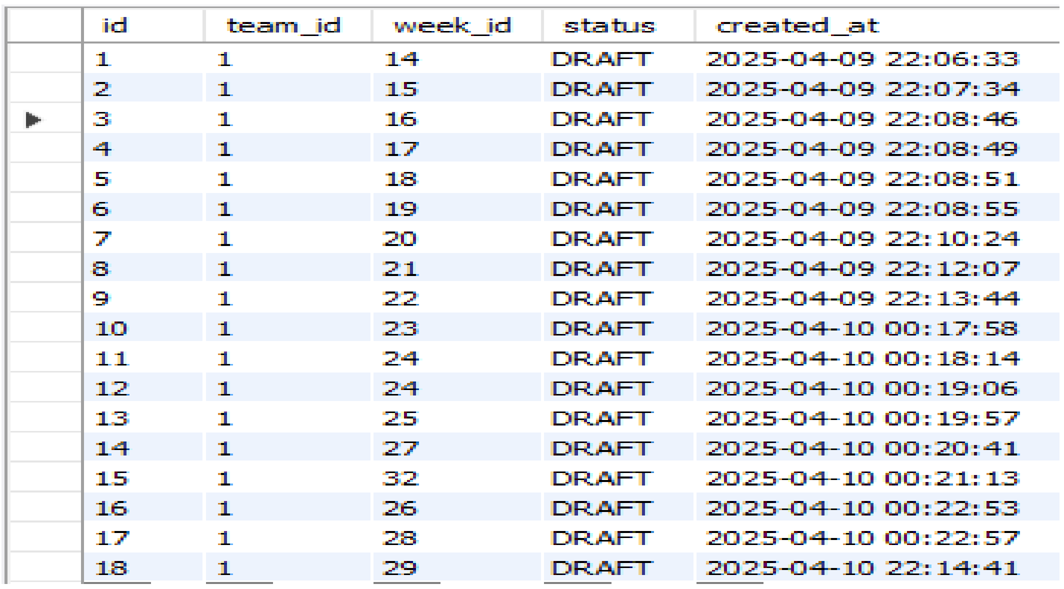
```
    def solve(self):
        solutions = self.problem.getSolutions()
        if not solutions:
            return {
                "assignments": [],
                "total_solutions": 0,
                "changed_count": 0,
                "fallback_count": 0,
                "skipped_count": 0,
                "skipped_assignments": [],
                "failure_reasons": ["No valid shift assignments found"]
            }

        # Avoid identical solution
        valid_solutions = []
        for sol in solutions:
            if not self._is_same_as_original(sol):
                valid_solutions.append(sol)
        # if there are no valid solutions do no changes
        if not valid_solutions:
            return {
                "assignments": [],
                "total_solutions": len(solutions),
                "changed_count": 0,
                "fallback_count": 0,
                "skipped_count": 0,
                "skipped_assignments": [],
                "failure_reasons": ["All solutions matched original (no change)"]
            }
        # getting best solution which is the first in valid solution array
        best_solution = valid_solutions[0]
        # empty array to store formated solutions
        formatted = []
        # initalising changed count variable to 0
        changed_count = 0
        # looping through assignments in the best solutions and getting all locked items
        for (shift_id, day_id, slot), user in best_solution.items():
            locked = any(
                a['shift_id'] == shift_id and a['day_id'] == day_id and a['slot'] == slot and a['user_id'
] == user
                for a in self.locked_assignments
            )
            # Finding the original assignment matching the current shift, day, and slot
            orig = next((a for a in self.original_assignments if
                         a['shift_id'] == shift_id and a['day_id'] == day_id and a['slot'] == slot), None
)

# If an original assignment exists, the user has changed, and the slot is not locked, count it as a chang
e
            if orig and user != orig['user_id'] and not locked:
                changed_count += 1
            # Adding the current assignment to the formatted list
            formatted.append({
                "user_id": user,
                "shift_id": shift_id,
                "day_id": day_id,
                "slot": slot,
                "locked": locked
            })

        formatted.sort(key=lambda x: (x["day_id"], x["shift_id"], x["slot"]))

        return {
            "assignments": formatted,
            "total_solutions": len(valid_solutions),
            "changed_count": changed_count,
            "fallback_count": 0,
            "skipped_count": 0,
            "skipped_assignments": [],
            "failure_reasons": [],
            "message": "New solution generated"
        }
```

Figure 103, Regenerate Solver #4

5.5.6 Storage and Retrieval

As mentioned within Route section (5.6.3), all assignments and solutions are stored directly into the database. Each solution and assignment within the database match the database schema, which was designed, as we can see from the figure below, each assignment is linking to a solution through a foreign key called `solution_id`. The `week_id` also provided that the user can view multiple solutions at once for a specific week, comparing the solutions, then choosing the solution which was most suitable for the employer.

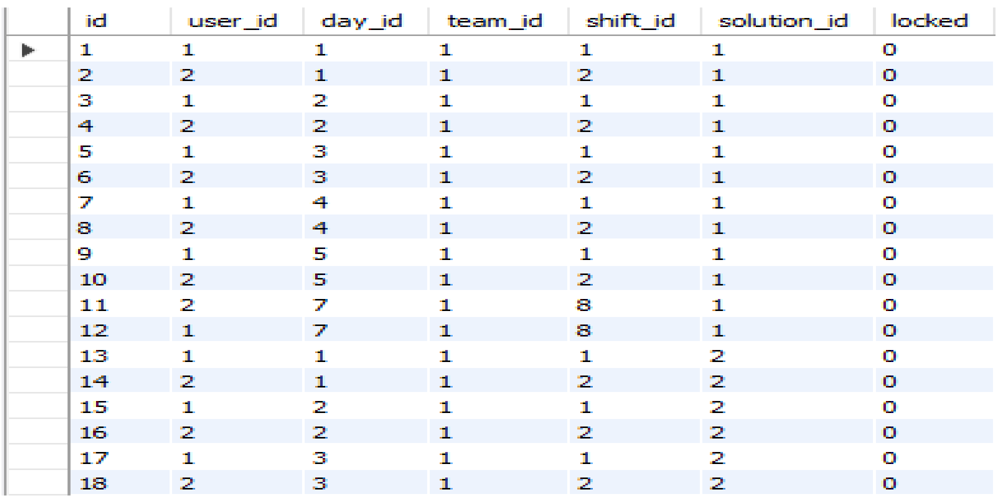
Solutions:



	id	team_id	week_id	status	created_at
	1	1	14	DRAFT	2025-04-09 22:06:33
	2	1	15	DRAFT	2025-04-09 22:07:34
▶	3	1	16	DRAFT	2025-04-09 22:08:46
	4	1	17	DRAFT	2025-04-09 22:08:49
	5	1	18	DRAFT	2025-04-09 22:08:51
	6	1	19	DRAFT	2025-04-09 22:08:55
	7	1	20	DRAFT	2025-04-09 22:10:24
	8	1	21	DRAFT	2025-04-09 22:12:07
	9	1	22	DRAFT	2025-04-09 22:13:44
	10	1	23	DRAFT	2025-04-10 00:17:58
	11	1	24	DRAFT	2025-04-10 00:18:14
	12	1	24	DRAFT	2025-04-10 00:19:06
	13	1	25	DRAFT	2025-04-10 00:19:57
	14	1	27	DRAFT	2025-04-10 00:20:41
	15	1	32	DRAFT	2025-04-10 00:21:13
	16	1	26	DRAFT	2025-04-10 00:22:53
	17	1	28	DRAFT	2025-04-10 00:22:57
	18	1	29	DRAFT	2025-04-10 22:14:41

Figure 104, Solutions SQL table

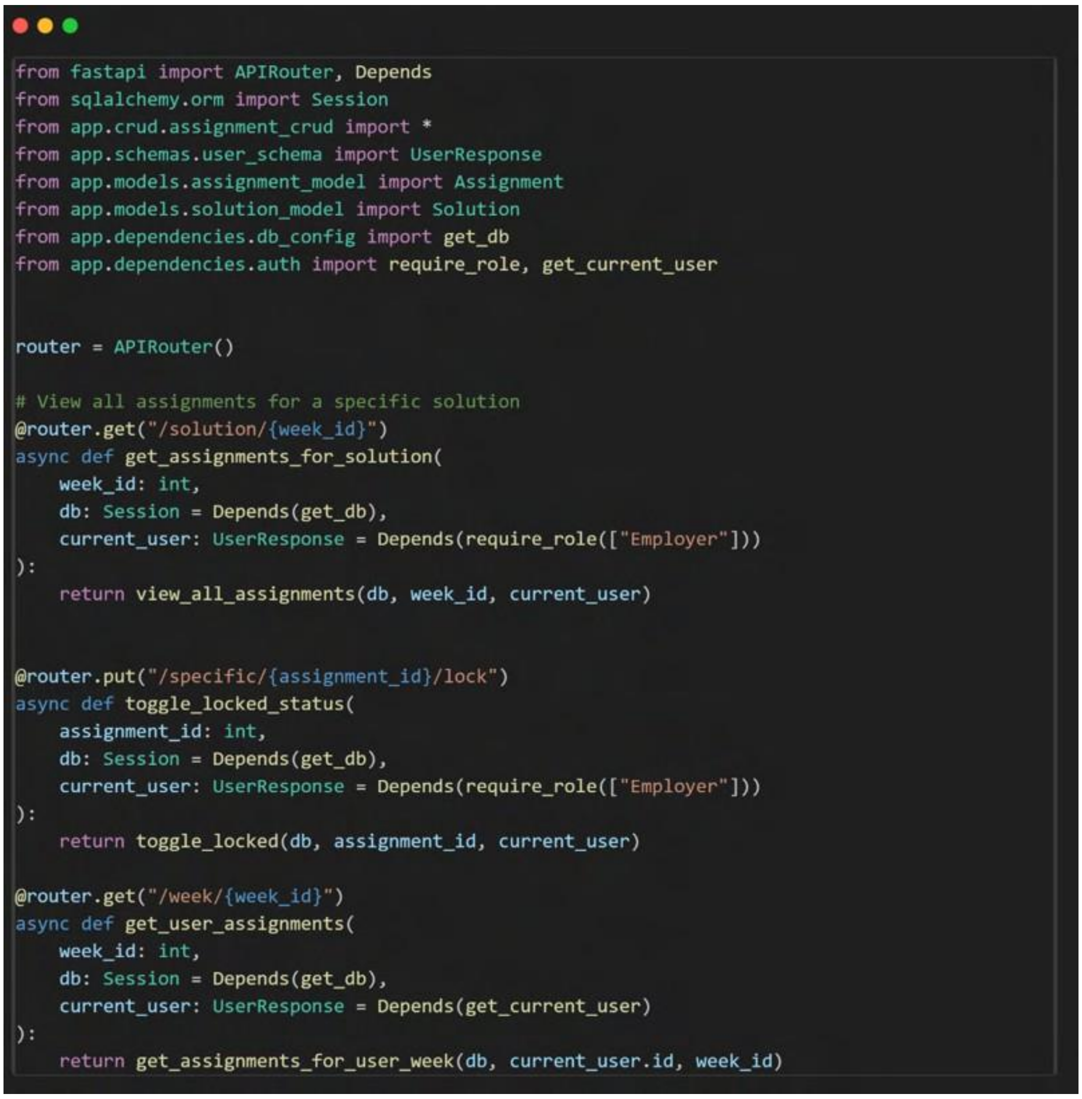
Assignments:



id	user_id	day_id	team_id	shift_id	solution_id	locked
1	1	1	1	1	1	0
2	2	1	1	2	1	0
3	1	2	1	1	1	0
4	2	2	1	2	1	0
5	1	3	1	1	1	0
6	2	3	1	2	1	0
7	1	4	1	1	1	0
8	2	4	1	2	1	0
9	1	5	1	1	1	0
10	2	5	1	2	1	0
11	2	7	1	8	1	0
12	1	7	1	8	1	0
13	1	1	1	1	2	0
14	2	1	1	2	2	0
15	1	2	1	1	2	0
16	2	2	1	2	2	0
17	1	3	1	1	2	0
18	2	3	1	2	2	0

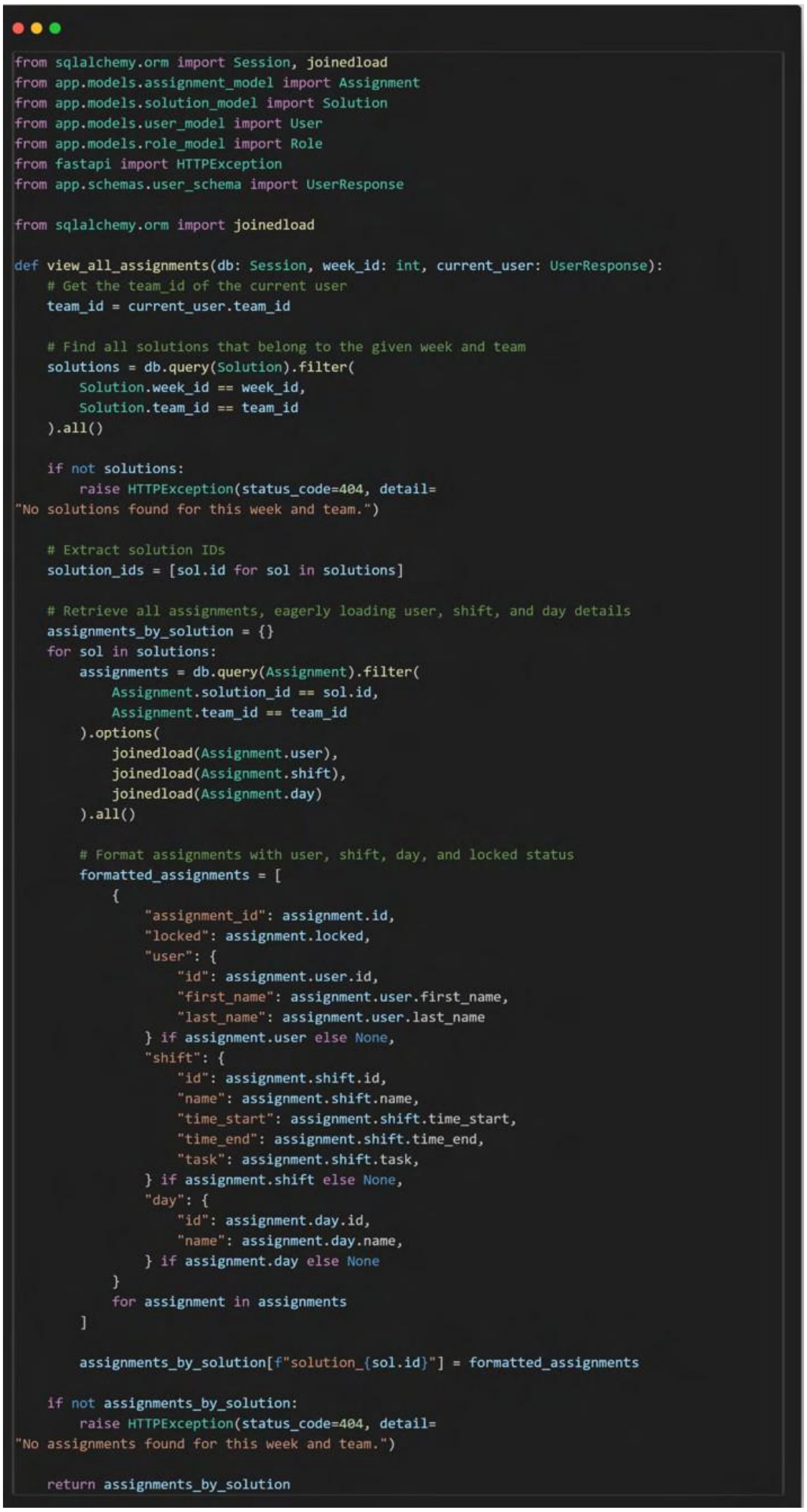
Figure 105, Assignments SQL Table

The logic behind the retrieval of solutions and related assignments are done within `assignment_crud.py` file, the routes assigned to each of these functions are stored within the `assignment_route.py` file. The figures below show how an employer can access their solutions, whether they are status draft or complete, and how they can update specific assignments to toggle their locked Boolean and prepare the solution for reassignment if needed.



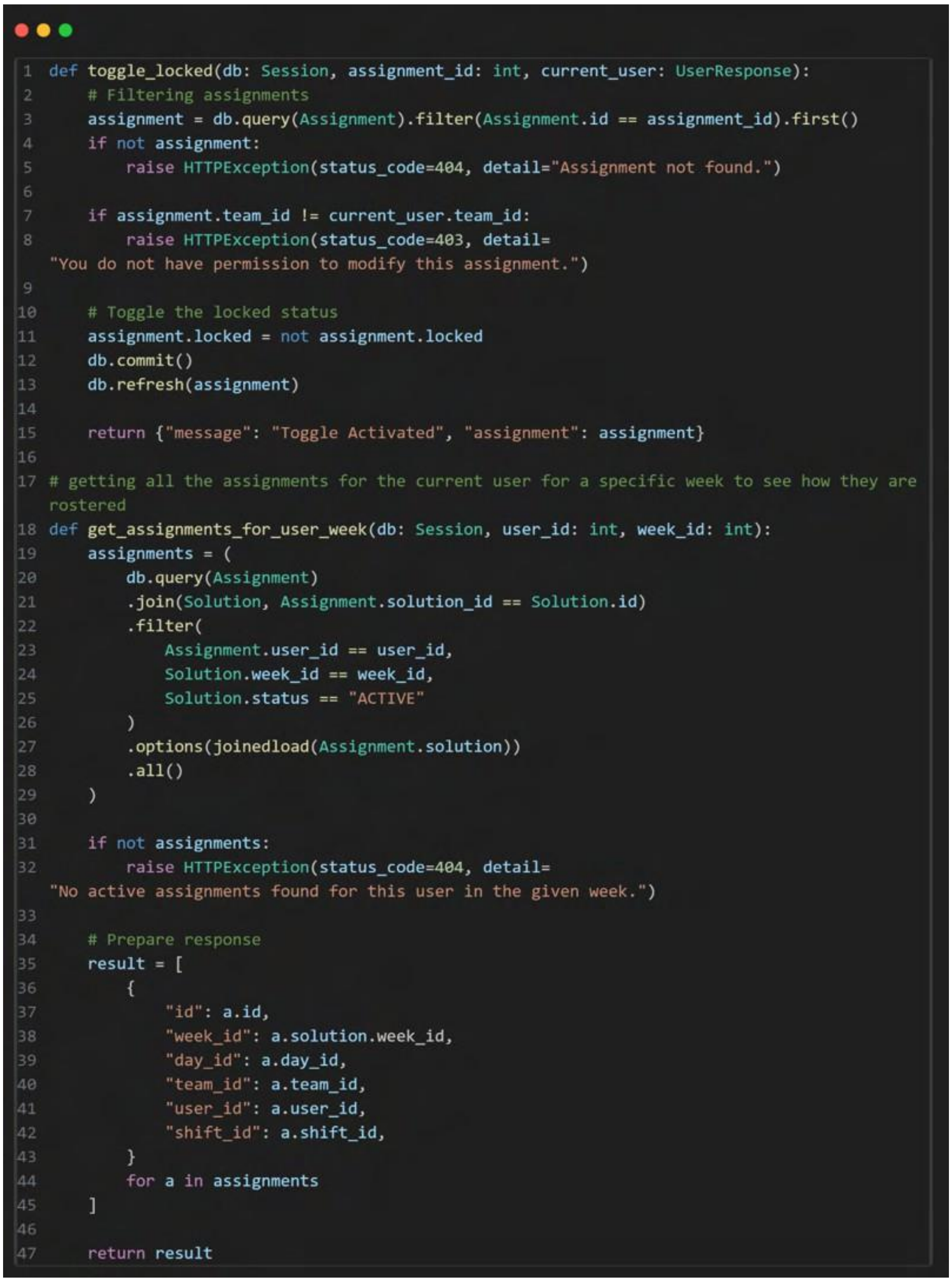
```python
from fastapi import APIRouter, Depends
from sqlalchemy.orm import Session
from app.crud.assignment_crud import *
from app.schemas.user_schema import UserResponse
from app.models.assignment_model import Assignment
from app.models.solution_model import Solution
from app.dependencies.db_config import get_db
from app.dependencies.auth import require_role, get_current_user


router = APIRouter()

# View all assignments for a specific solution
@router.get("/solution/{week_id}")
async def get_assignments_for_solution(
    week_id: int,
    db: Session = Depends(get_db),
    current_user: UserResponse = Depends(require_role(["Employer"]))
):
    return view_all_assignments(db, week_id, current_user)


@router.put("/specific/{assignment_id}/lock")
async def toggle_locked_status(
    assignment_id: int,
    db: Session = Depends(get_db),
    current_user: UserResponse = Depends(require_role(["Employer"]))
):
    return toggle_locked(db, assignment_id, current_user)

@router.get("/week/{week_id}")
async def get_user_assignments(
    week_id: int,
    db: Session = Depends(get_db),
    current_user: UserResponse = Depends(get_current_user)
):
    return get_assignments_for_user_week(db, current_user.id, week_id)
```

Figure 106, Routes for solutions and assignments



```python
from sqlalchemy.orm import Session, joinedload
from app.models.assignment_model import Assignment
from app.models.solution_model import Solution
from app.models.user_model import User
from app.models.role_model import Role
from fastapi import HTTPException
from app.schemas.user_schema import UserResponse

from sqlalchemy.orm import joinedload

def view_all_assignments(db: Session, week_id: int, current_user: UserResponse):
    # Get the team_id of the current user
    team_id = current_user.team_id

    # Find all solutions that belong to the given week and team
    solutions = db.query(Solution).filter(
        Solution.week_id == week_id,
        Solution.team_id == team_id
    ).all()

    if not solutions:
        raise HTTPException(status_code=404, detail=
"No solutions found for this week and team.")

    # Extract solution IDs
    solution_ids = [sol.id for sol in solutions]

    # Retrieve all assignments, eagerly loading user, shift, and day details
    assignments_by_solution = {}
    for sol in solutions:
        assignments = db.query(Assignment).filter(
            Assignment.solution_id == sol.id,
            Assignment.team_id == team_id
        ).options(
            joinedload(Assignment.user),
            joinedload(Assignment.shift),
            joinedload(Assignment.day)
        ).all()

        # Format assignments with user, shift, day, and locked status
        formatted_assignments = [
            {
                "assignment_id": assignment.id,
                "locked": assignment.locked,
                "user": {
                    "id": assignment.user.id,
                    "first_name": assignment.user.first_name,
                    "last_name": assignment.user.last_name
                } if assignment.user else None,
                "shift": {
                    "id": assignment.shift.id,
                    "name": assignment.shift.name,
                    "time_start": assignment.shift.time_start,
                    "time_end": assignment.shift.time_end,
                    "task": assignment.shift.task,
                } if assignment.shift else None,
                "day": {
                    "id": assignment.day.id,
                    "name": assignment.day.name,
                } if assignment.day else None
            }
            for assignment in assignments
        ]

        assignments_by_solution[f"solution_{sol.id}"] = formatted_assignments

    if not assignments_by_solution:
        raise HTTPException(status_code=404, detail=
"No assignments found for this week and team.")

    return assignments_by_solution
```

Figure 107, Crud File functions for solutions and assignments



```python
def toggle_locked(db: Session, assignment_id: int, current_user: UserResponse):
    # Filtering assignments
    assignment = db.query(Assignment).filter(Assignment.id == assignment_id).first()
    if not assignment:
        raise HTTPException(status_code=404, detail="Assignment not found.")

    if assignment.team_id != current_user.team_id:
        raise HTTPException(status_code=403, detail=
"You do not have permission to modify this assignment.")

    # Toggle the locked status
    assignment.locked = not assignment.locked
    db.commit()
    db.refresh(assignment)

    return {"message": "Toggle Activated", "assignment": assignment}

# getting all the assignments for the current user for a specific week to see how they are
rostered
def get_assignments_for_user_week(db: Session, user_id: int, week_id: int):
    assignments = (
        db.query(Assignment)
        .join(Solution, Assignment.solution_id == Solution.id)
        .filter(
            Assignment.user_id == user_id,
            Solution.week_id == week_id,
            Solution.status == "ACTIVE"
        )
        .options(joinedload(Assignment.solution))
        .all()
    )

    if not assignments:
        raise HTTPException(status_code=404, detail=
"No active assignments found for this user in the given week.")

    # Prepare response
    result = [
        {
            "id": a.id,
            "week_id": a.solution.week_id,
            "day_id": a.day_id,
            "team_id": a.team_id,
            "user_id": a.user_id,
            "shift_id": a.shift_id,
        }
        for a in assignments
    ]

    return result
```

Figure 108, Locking Assignments Route

5.6 Frontend

5.6.1 Overview

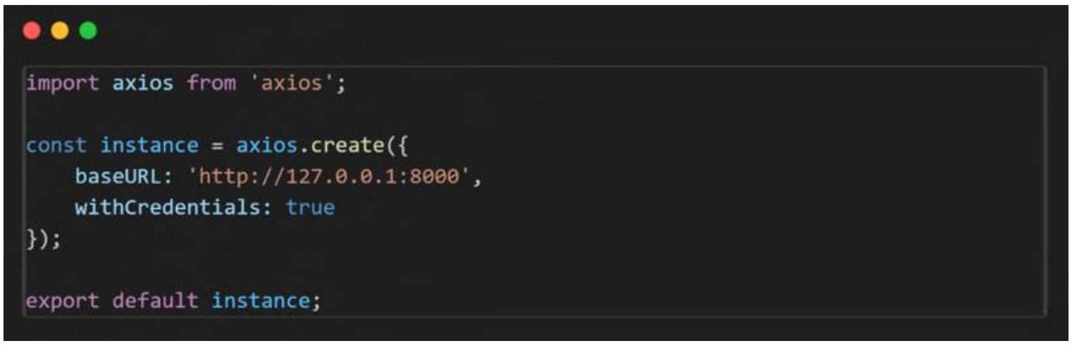
In relation to the program design section (4.2), the frontend was developed using React, a popular JavaScript library known for its efficient rendering and development of single-page applications (SPAs). Vite was selected as the build tool due to its fast development server which offered a smoother development experience through efficient rebuilds in relation to changes in code.

React also offered access to a wide variety of libraries such as React Router and React Big Callendar. React Router was used for implementing client-side routing, offering dynamic navigation between pages. React Big Calendar was integrated to provide an interactive calendar, which was used for the core functionality of the application.

To enhance the user interface (UI), the Shadcn component library was implemented, providing a consistent design. Its integration with Tailwind CSS allowed for customisation of these components including the colour palette, typography and layout. This ensured the interface aligned with the user interface design (4.3).

5.6.2 Backend Connection

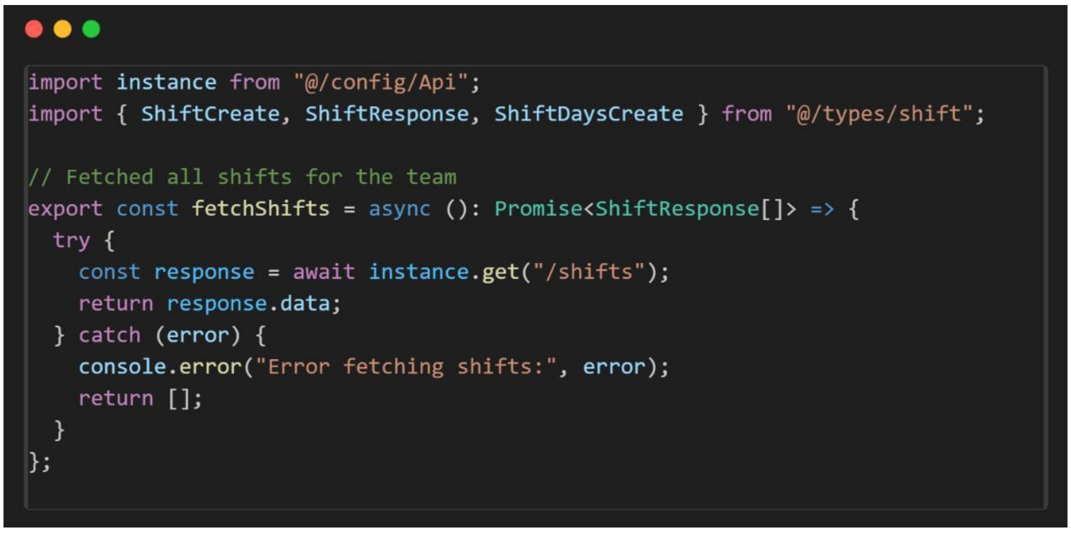
To enable communication between the frontend and backend, an axios instance was created within `Api.tsx` file. Axios is an asynchronous HTTP client that simplifies making requests from the frontend to a backend server. The figure below displays how this configuration sets the base URL as the backend server ensuring credentials is set to true, credentials make sure that the cookies are passed with the request. The hosted sites URL would be set in `.env` and called upon this file, this allows for immediate transfer after the development.



```
import axios from 'axios';

const instance = axios.create({
    baseURL: 'http://127.0.0.1:8000',
    withCredentials: true
});

export default instance;
```

Figure 109, Api.tsx file

To keep a clean and modular architecture, the backend routes are organised into separate files within services folder. Each file corresponds to a specific set of routes and contains relevant functions for interacting with those routes. Keeping all these routes in a service file allows for easy import and reuse of these functions in various pages. The figure below displays a function from the shift service file, containing a variety of asynchronous functions that ensure the application is not blocked while making the HTTP request. Each function takes specific parameters, such as an ID, and returns a promise with a defined response type, this ensures the request will eventually return data. The functions use the axios instance as the base of the URL followed with the rest of URL endpoint with the appropriate HTTP method.

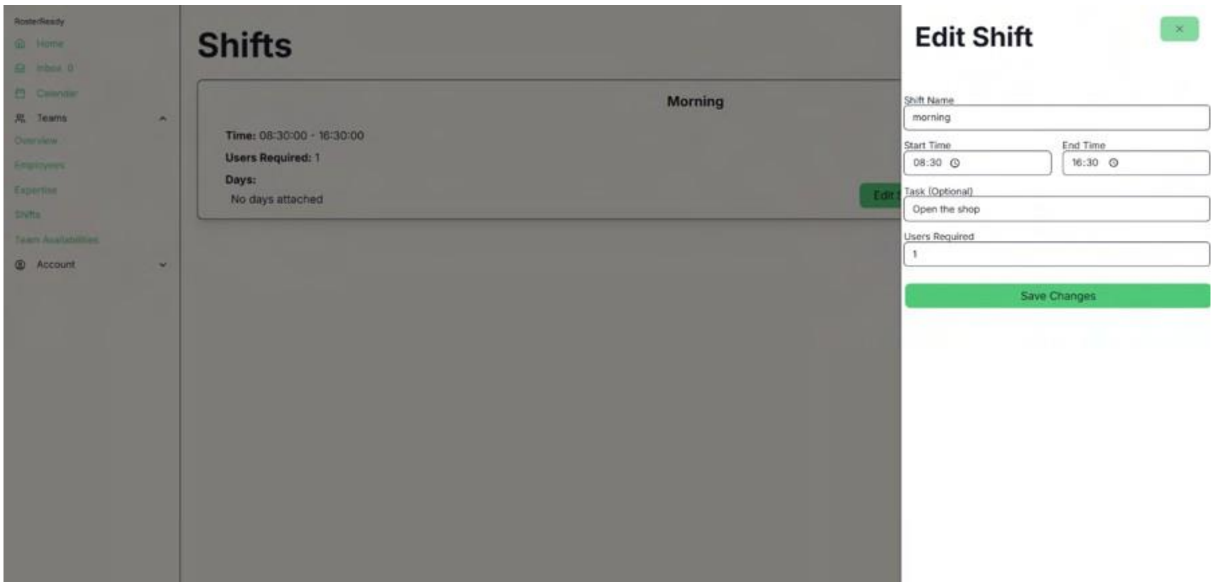


```
import instance from "@/config/Api";
import { ShiftCreate, ShiftResponse, ShiftDaysCreate } from "@/types/shift";

// Fetched all shifts for the team
export const fetchShifts = async (): Promise<ShiftResponse[]> => {
  try {
    const response = await instance.get("/shifts");
    return response.data;
  } catch (error) {
    console.error("Error fetching shifts:", error);
    return [];
  }
};
```

Figure 110, Shift Service

5.6.3 Forms

The application included various forms to manage the CRUD operations for entities like users, teams, shifts, expertise, and more. These forms were wrapped in either a dialog or a sheet, which was provided by the Shadcn component library, ensuring that all related data was organized on a single page. The implementation of Reacts, useState hook was to manage the visibility of these forms, which made the application appear seamless. The figure below illustrates how the edit sheet was displayed.



Shifts

Morning

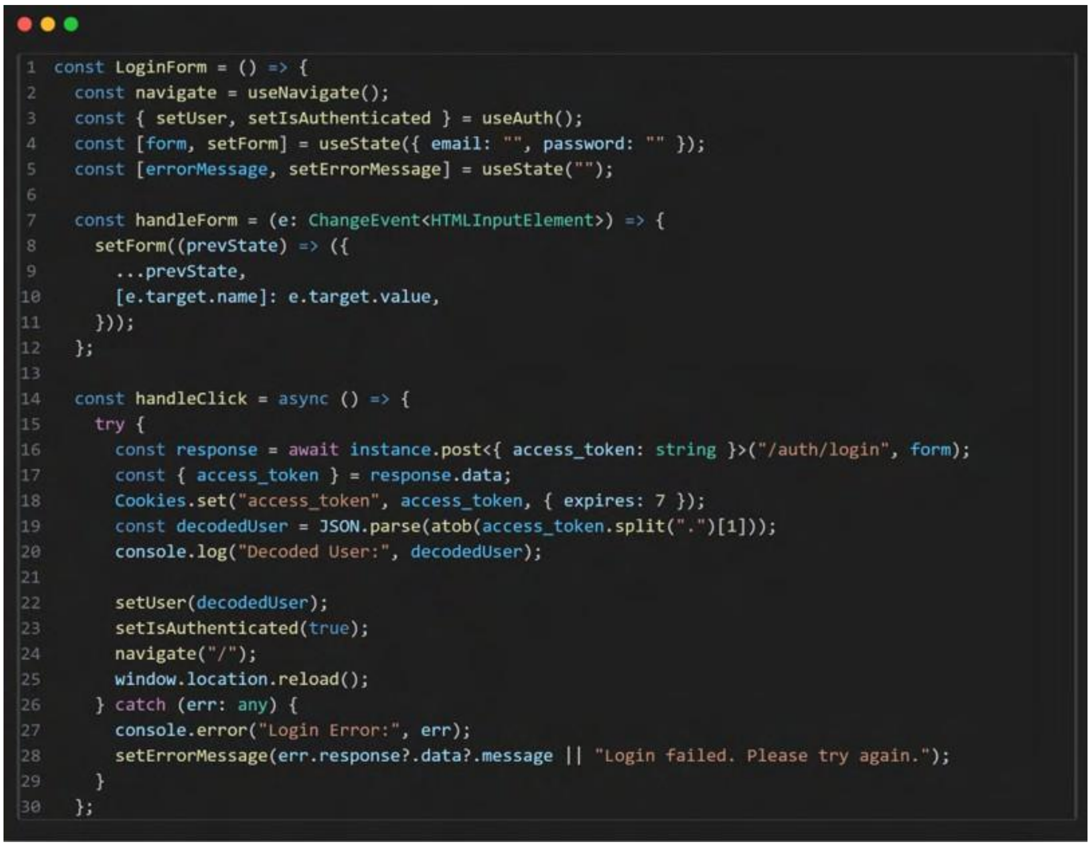
Time: 08:30:00 - 16:30:00
Users Required: 1
Days:
No days attached

Edit Shift

Shift Name: morning
Start Time: 08:30
End Time: 16:30
Task (Optional): Open the shop
Users Required: 1

Figure 111, Shift Edit Sheet

The core form was the login form, which handled the authentication for the application. It validated users and granted access to all other protected routes and features. The figure below displays how this form was implemented. The form collects users input for email and password; the input is stored in a useState called `form`. The handle click function is called once the submit button is pressed, this sends a post request using the axios instance. If the response is successful, the JWT access token is stored in a cookie using the `js-cookie` library. The token is decoded and stored in the applications authentication context, for future retrieval of other routes.

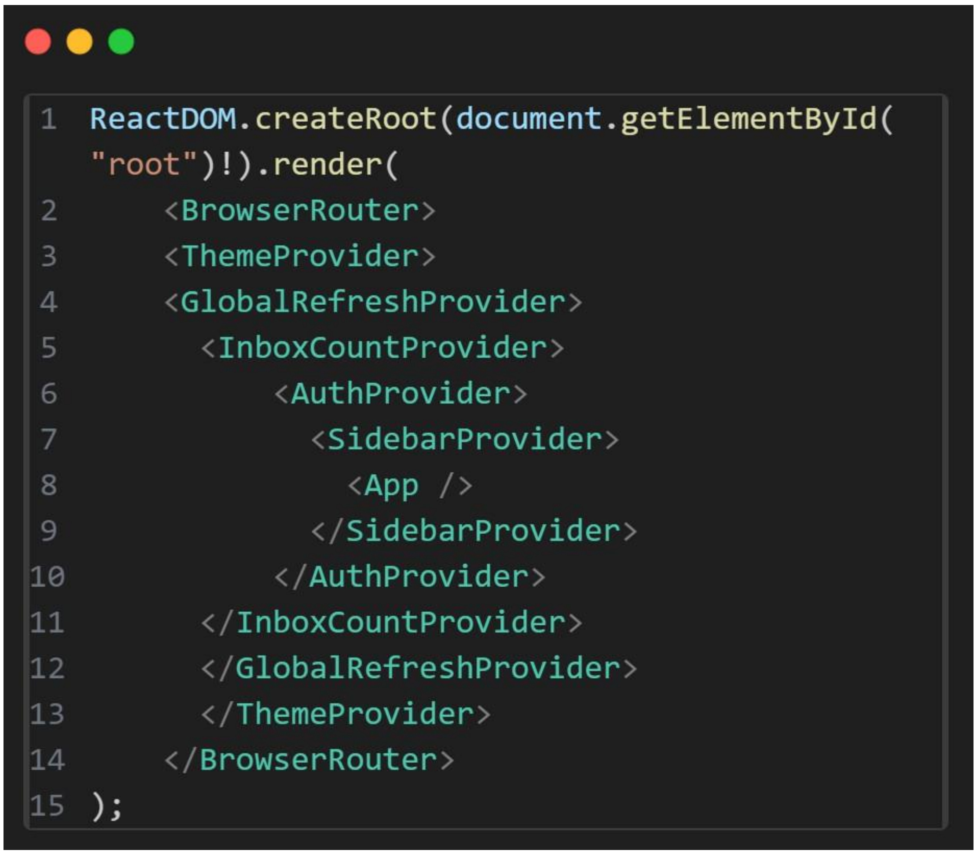


```
const LoginForm = () => {
  const navigate = useNavigate();
  const { setUser, setIsAuthenticated } = useAuth();
  const [form, setForm] = useState({ email: "", password: "" });
  const [errorMessage, setErrorMessage] = useState("");

  const handleForm = (e: ChangeEvent<HTMLInputElement>) => {
    setForm((prevState) => ({
      ...prevState,
      [e.target.name]: e.target.value,
    }));
  };

  const handleClick = async () => {
    try {
      const response = await instance.post<{ access_token: string }>("/auth/login", form);
      const { access_token } = response.data;
      Cookies.set("access_token", access_token, { expires: 7 });
      const decodedUser = JSON.parse(atob(access_token.split(".")[1]));
      console.log("Decoded User:", decodedUser);

      setUser(decodedUser);
      setIsAuthenticated(true);
      navigate("/");
      window.location.reload();
    } catch (err: any) {
      console.error("Login Error:", err);
      setErrorMessage(err.response?.data?.message || "Login failed. Please try again.");
    }
  };
```

Figure 112, Login Form Code

5.6.4 Contexts

Contexts were fundamentals for managing and sharing a state across the entire application, rather than passing props manually between each component. Context is a method provided by React to implement these global states efficiently. There are four distinct contexts within the application which are responsible for specific set of global states, authentication, global refresh, inbox count and theme management. These contexts were wrapped around the application within the `Main.tsx` file. The figure below illustrates the contexts being wrapped around the children, which is the `App.tsx` file.

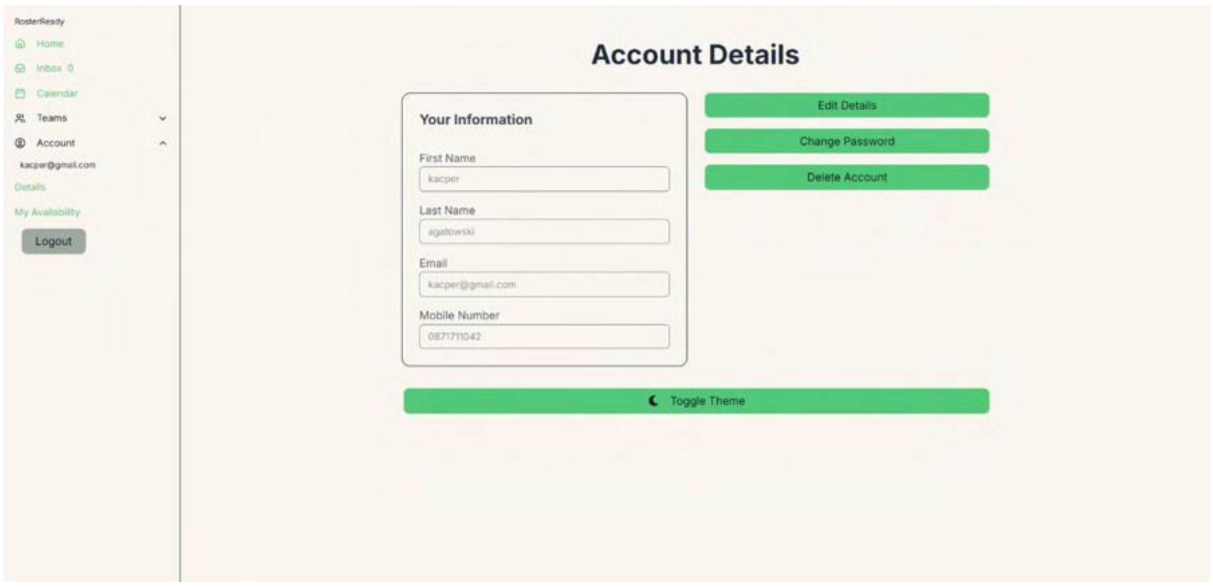


```
ReactDOM.createRoot(document.getElementById(
"root")!).render(
    <BrowserRouter>
    <ThemeProvider>
    <GlobalRefreshProvider>
      <InboxCountProvider>
          <AuthProvider>
            <SidebarProvider>
              <App />
            </SidebarProvider>
          </AuthProvider>
      </InboxCountProvider>
      </GlobalRefreshProvider>
      </ThemeProvider>
    </BrowserRouter>
);
```

Figure 113, Context Wrapping Children

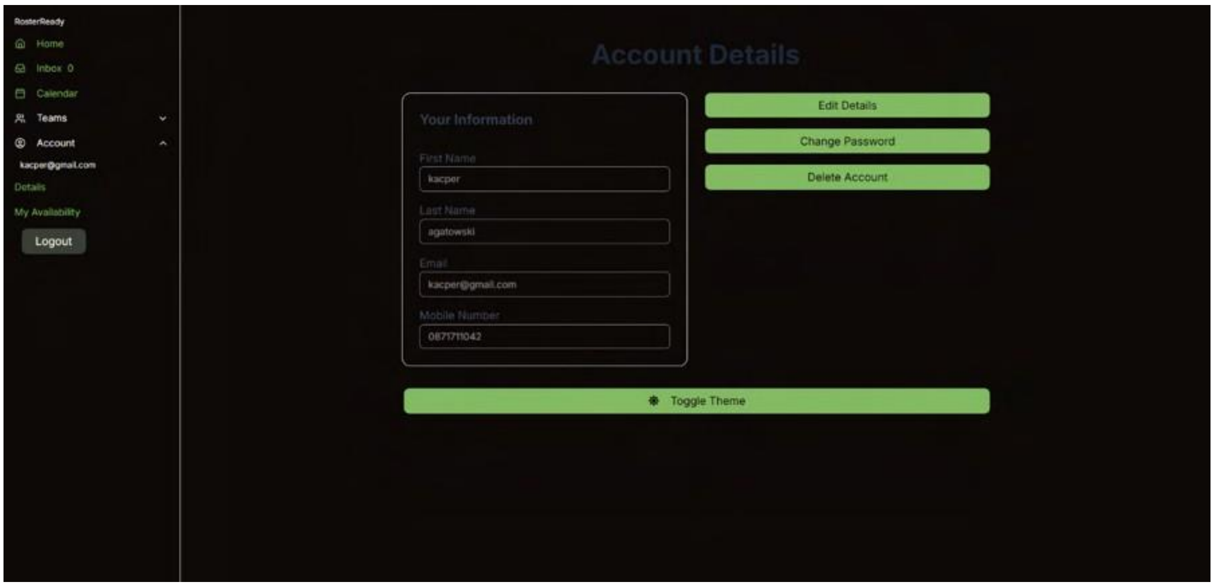
5.6.4.1 Theme Management Context

Theme management context is responsible for managing the state of the visual theme of the application. The state is set within the application page which is passed to all the children pages of the application. This theme is then stored within local storage, this ensures that if the user refreshes the page their theme is still set to chosen theme. There are two themes available, dark and light. Each theme utilises the colour palette mentioned in user interface design section (4.3.3). Figures 114 and 115, display both themes on the account page.



First Name: kacper
Last Name: agatowski
Email: kacper@gmail.com
Mobile Number: 0871711042

Figure 114, Light Theme

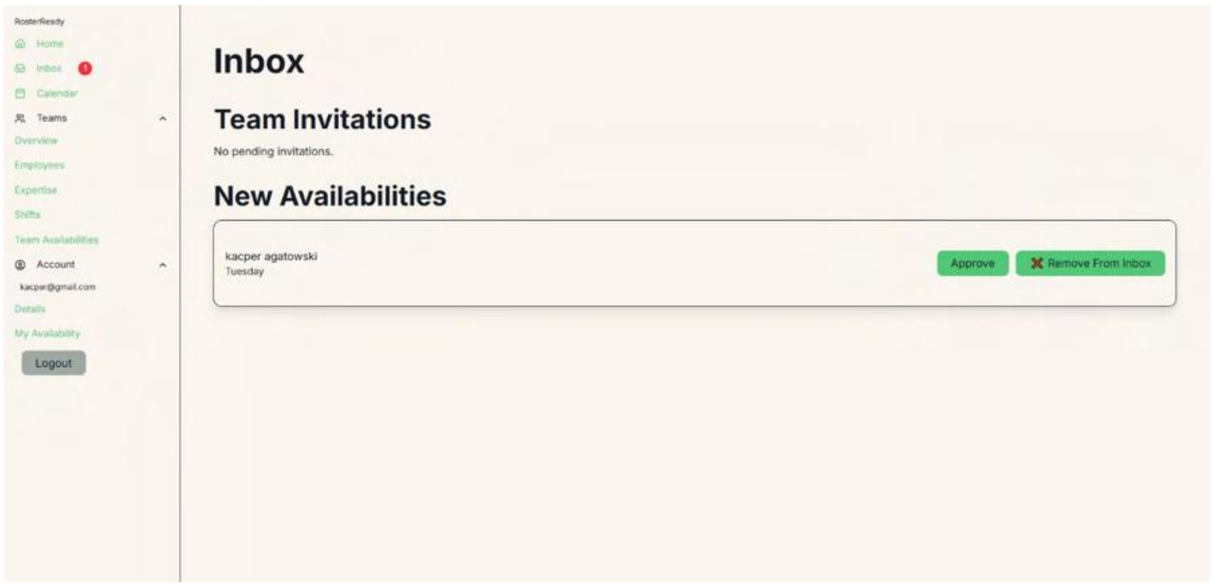


First Name: kacper
Last Name: agatowski
Email: kacper@gmail.com
Mobile Number: 0871711042

Figure 115, Dark Theme

5.6.4.2 Inbox Count Context

Inbox count context is responsible for managing the state of the integer within the sidebar beside the inbox. This stores the length of the invites and availabilities in a state which is set within the inbox file. Any changes made to the inbox such as accepting an invitation will dynamically update and reduce the count, providing real-time feedback to the user. Figure 116 illustrates how this inbox count was displayed with the inbox page.



Inbox

Team Invitations

No pending invitations.

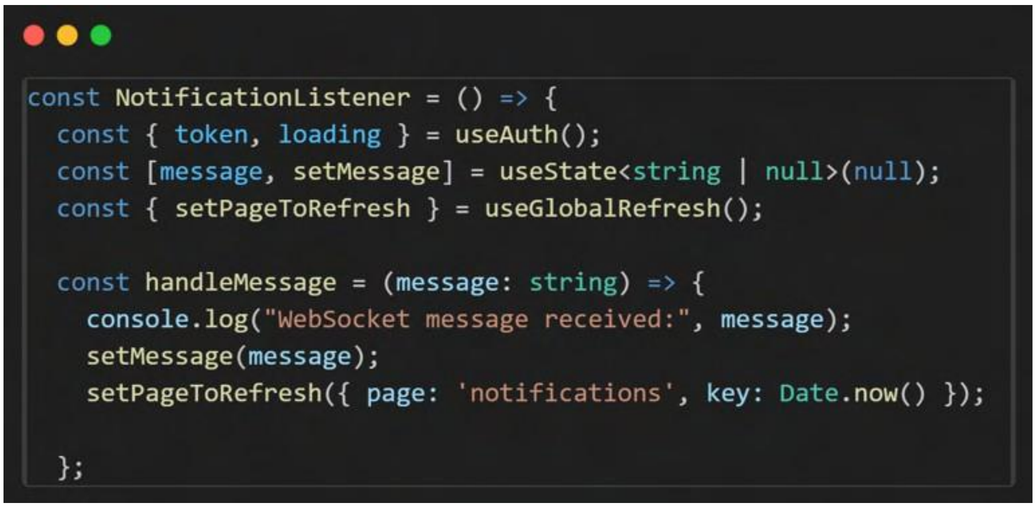
New Availabilities

kacper agatowski
Tuesday

Figure 116, Inbox Count Visual

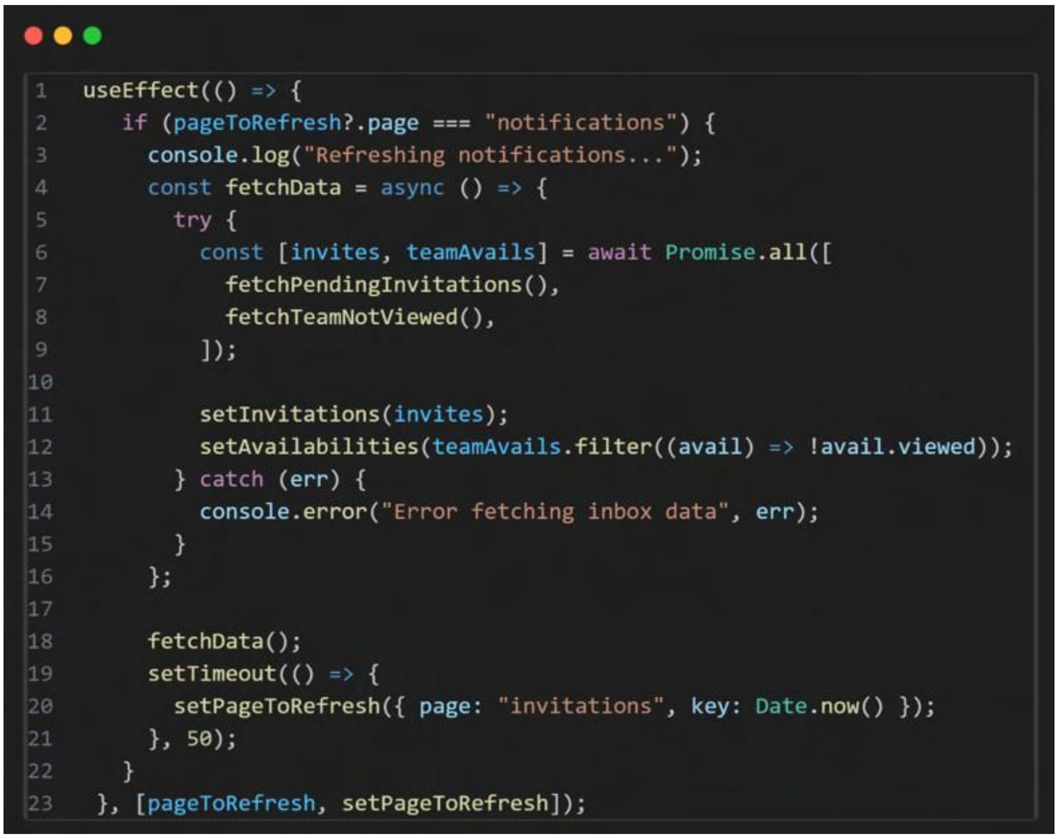
5.6.4.3 Global Refresh Context

The global refresh context was responsible for refreshing specific pages which were dependant on a variety of factors. Its prime connection was with the WebSocket server, it would listen for a message activity. If the message displayed information about availability or invitations, the global refresh state would be set to this text. Other children, such as pages would wait for these state key words and refresh specific functions in relation to the state. Figures 117 illustrates how this state was set upon a WebSocket message delivery, the global state is now set to "Notifications", figure 118 displays a condition which checks if the state is "Notifications" then it runs the function and sets the state to another function which needs to be refreshed. This made the application reload with WebSocket messages and data in real-time in relation to the database.



```
const NotificationListener = () => {
  const { token, loading } = useAuth();
  const [message, setMessage] = useState<string | null>(null);
  const { setPageToRefresh } = useGlobalRefresh();

  const handleMessage = (message: string) => {
    console.log("WebSocket message received:", message);
    setMessage(message);
    setPageToRefresh({ page: 'notifications', key: Date.now() });

  };
```

Figure 117, Setting the state upon notification

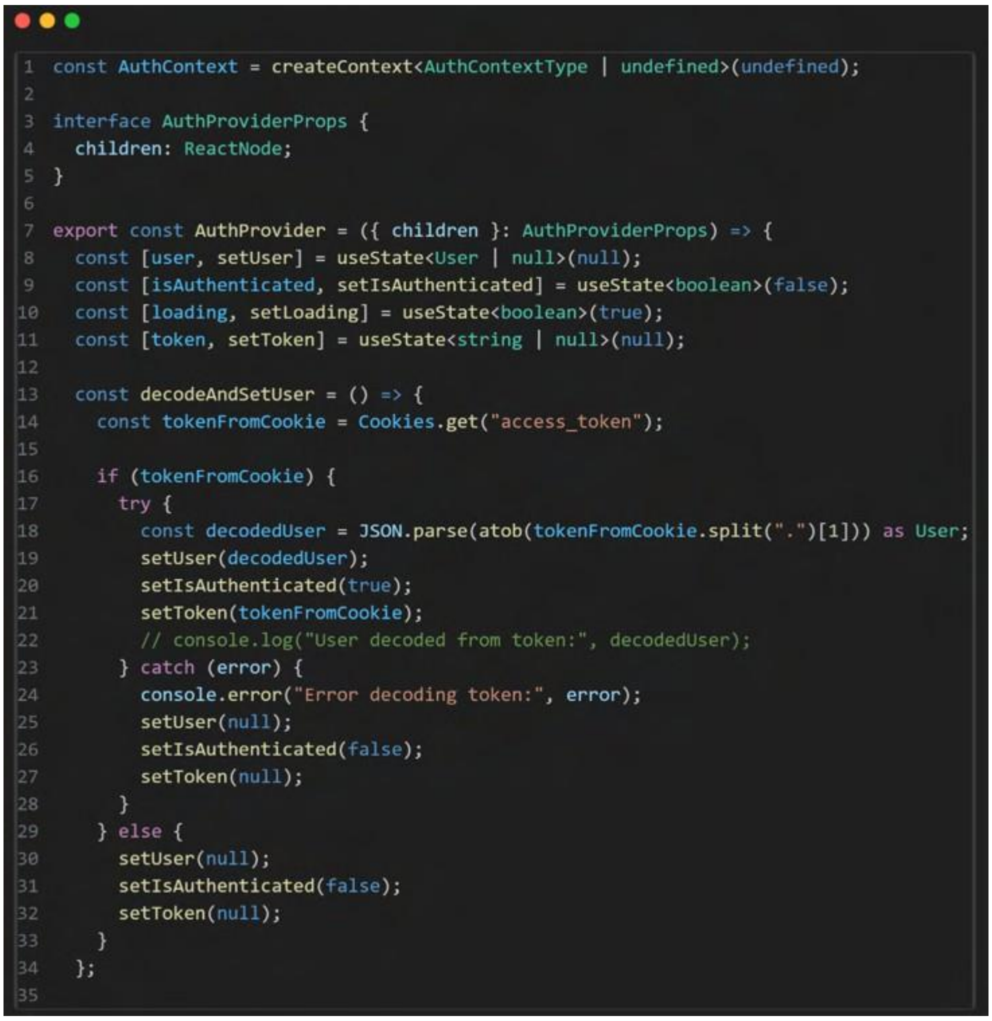


```
useEffect(() => {
   if (pageToRefresh?.page === "notifications") {
     console.log("Refreshing notifications...");
     const fetchData = async () => {
       try {
         const [invites, teamAvails] = await Promise.all([
           fetchPendingInvitations(),
           fetchTeamNotViewed(),
         ]);

         setInvitations(invites);
         setAvailabilities(teamAvails.filter((avail) => !avail.viewed));
       } catch (err) {
         console.error("Error fetching inbox data", err);
       }
     };

     fetchData();
     setTimeout(() => {
       setPageToRefresh({ page: "invitations", key: Date.now() });
     }, 50);
   }
 }, [pageToRefresh, setPageToRefresh]);
```

Figure 118, Using the state to run a function

5.6.4.4 Authentication Context

The authentication context was responsible for setting the state of the current user. The state was set upon login, which held multiple different states which would be reused throughout the application. This context would get the access token which was returned as a cookie through the login response. If the cookie existed, it would store the user information in a state called user and would set authenticated to true. Additionally, the token from the cookie was stored. Storing the cookie within a state had to be implemented for the WebSocket route, as the WebSocket route could not access the cookie due to a bug. These states were used across the whole application, sending user information within routes or checking if the user is authenticated to view certain pages. Implementing authentication as context simplified role-based access control and authentication control across the application. Figure 119 displays the file structure of this context.



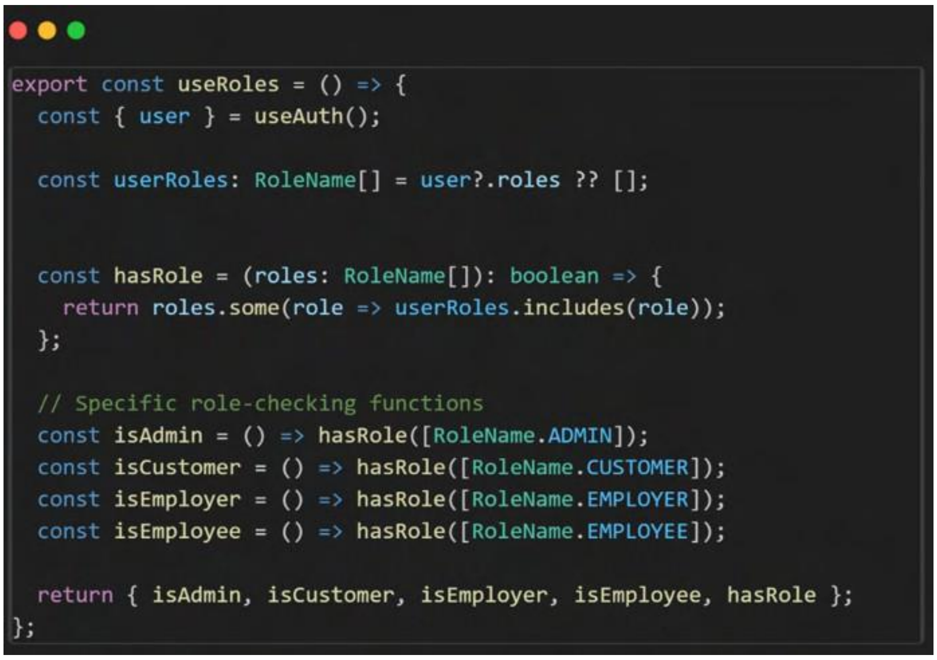
```
const AuthContext = createContext<AuthContextType | undefined>(undefined);

interface AuthProviderProps {
  children: ReactNode;
}

export const AuthProvider = ({ children }: AuthProviderProps) => {
  const [user, setUser] = useState<User | null>(null);
  const [isAuthenticated, setIsAuthenticated] = useState<boolean>(false);
  const [loading, setLoading] = useState<boolean>(true);
  const [token, setToken] = useState<string | null>(null);

  const decodeAndSetUser = () => {
    const tokenFromCookie = Cookies.get("access_token");

    if (tokenFromCookie) {
      try {
        const decodedUser = JSON.parse(atob(tokenFromCookie.split(".")[1])) as User;
        setUser(decodedUser);
        setIsAuthenticated(true);
        setToken(tokenFromCookie);
        // console.log("User decoded from token:", decodedUser);
      } catch (error) {
        console.error("Error decoding token:", error);
        setUser(null);
        setIsAuthenticated(false);
        setToken(null);
      }
    } else {
      setUser(null);
      setIsAuthenticated(false);
      setToken(null);
    }
  };

```

Figure 119, File structure of Authentication Context

5.6.5 Hooks

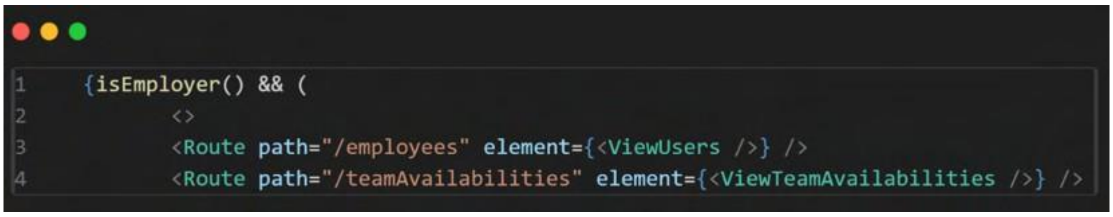
Hooks were implemented to encapsulate logic that can be reused across the application, this made code more modular and eliminated any repeating code. Two hooks were implemented, `useRoles` and `useWebSocket`. Each hook was responsible to manage a specific functionality.

The `useRoles.tsx` file was responsible for checking a specific role using the authentication context user state, ensuring that the role required for certain pages and functionality were correctly validated across the application. Figure 120 illustrates how this hook was implemented and how each arrow function represented a role within the database. Figure 121 displays how this hook was used as condition in relation to view routes in the `App.tsx` file.



```
export const useRoles = () => {
  const { user } = useAuth();

  const userRoles: RoleName[] = user?.roles ?? [];


  const hasRole = (roles: RoleName[]): boolean => {
    return roles.some(role => userRoles.includes(role));
  };

  // Specific role-checking functions
  const isAdmin = () => hasRole([RoleName.ADMIN]);
  const isCustomer = () => hasRole([RoleName.CUSTOMER]);
  const isEmployer = () => hasRole([RoleName.EMPLOYER]);
  const isEmployee = () => hasRole([RoleName.EMPLOYEE]);

  return { isAdmin, isCustomer, isEmployer, isEmployee, hasRole };
};
```

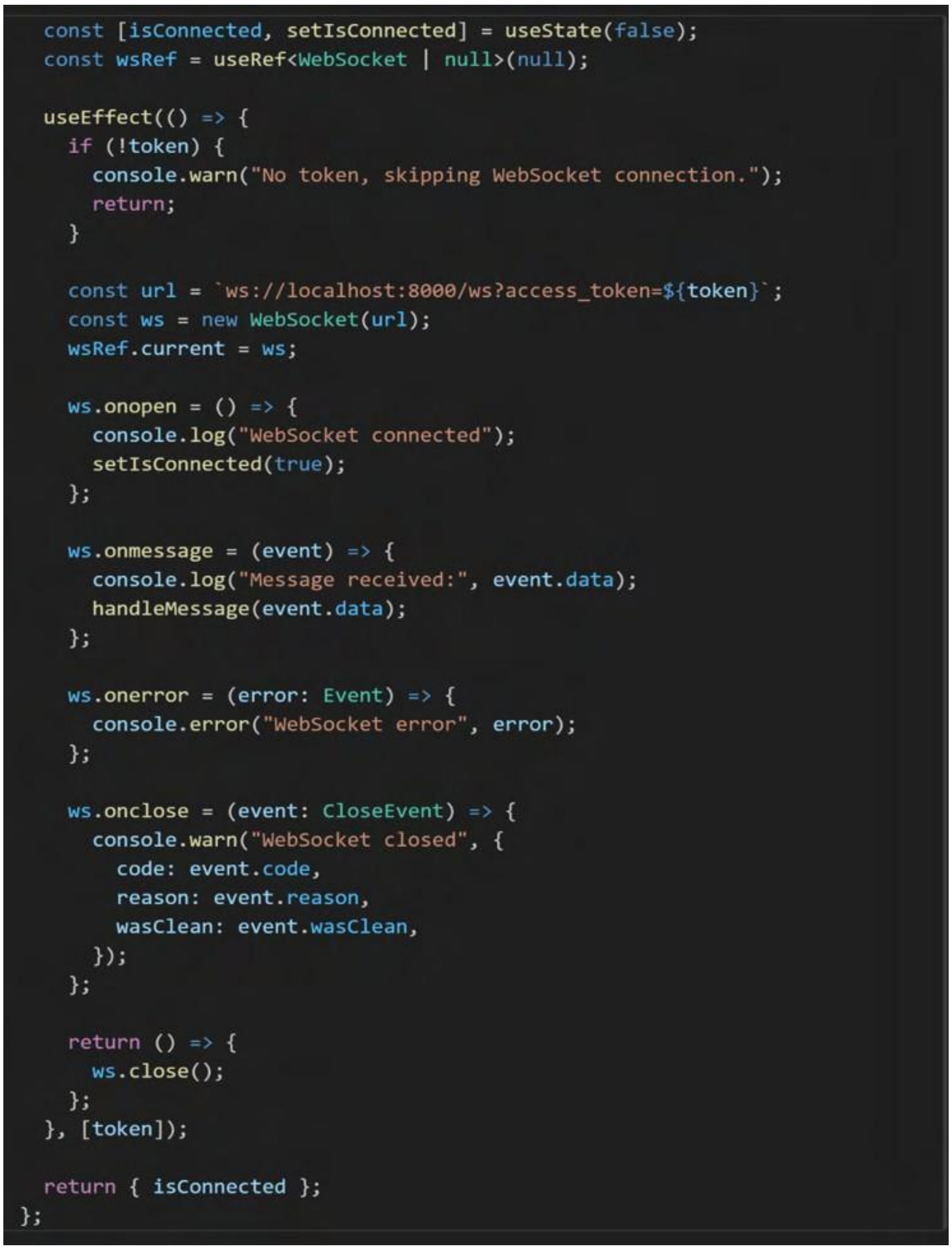
Figure 120, UseRoles hook



```
    {isEmployer() && (
           <>
           <Route path="/employees" element={<ViewUsers />} />
           <Route path="/teamAvailabilities" element={<ViewTeamAvailabilities />} />
```

Figure 121, UseRoles hook being used

The `useWebSocket.tsx` file was responsible for managing the WebSocket connection, including connecting, handling messages, monitoring status and disconnection. This hook utilised `useEffect` hook to initiate the WebSocket connection immediately when the application is mounted and when the login is made. The development URL was initialised in a variable passing the access token with the URL as a query parameter. Using this URL a new WebSocket instance was created using new WebSocket. The hook maintained a connection status state of `isConnected` to pass it to other components, this allowed the user interface to provide real-time feedback to the users regarding their connection status. Figure 122 displays how this code was implemented.

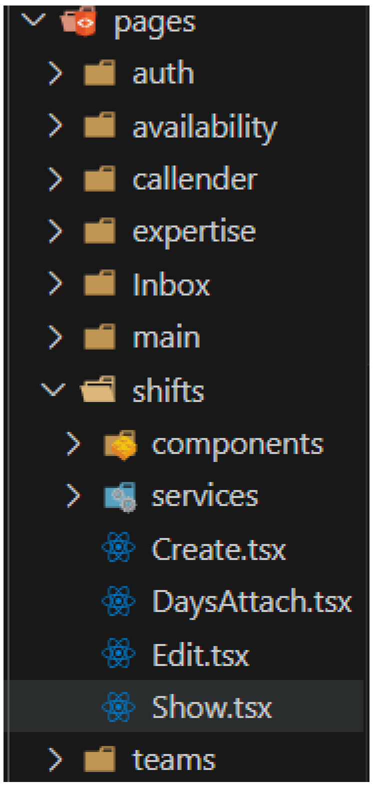


```
  const [isConnected, setIsConnected] = useState(false);
  const wsRef = useRef<WebSocket | null>(null);

  useEffect(() => {
    if (!token) {
      console.warn("No token, skipping WebSocket connection.");
      return;
    }

    const url = `ws://localhost:8000/ws?access_token=${token}`;
    const ws = new WebSocket(url);
    wsRef.current = ws;

    ws.onopen = () => {
      console.log("WebSocket connected");
      setIsConnected(true);
    };

    ws.onmessage = (event) => {
      console.log("Message received:", event.data);
      handleMessage(event.data);
    };

    ws.onerror = (error: Event) => {
      console.error("WebSocket error", error);
    };

    ws.onclose = (event: CloseEvent) => {
      console.warn("WebSocket closed", {
        code: event.code,
        reason: event.reason,
        wasClean: event.wasClean,
      });
    };

    return () => {
      ws.close();
    };
  }, [token]);

  return { isConnected };
};
```

Figure 122, WebSocket Hook

5.6.6 Views and Routing

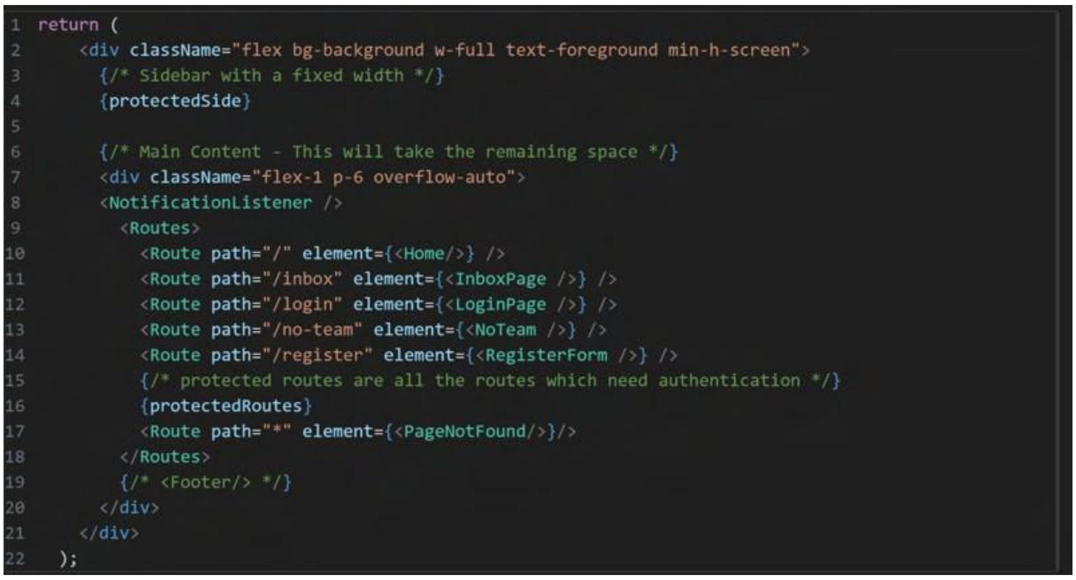
The implementation of each view was modularised into separate folders, following a clear structure that included a components folder for UI components, a services folder for the API interactions and additional files for handling the CRUD operations, this layout can be found in figure 123.



```
pages
  auth
  availability
  callender
  expertise
  Inbox
  main
  shifts
    components
    services
    Create.tsx
    DaysAttach.tsx
    Edit.tsx
    Show.tsx
  teams
```

Figure 123, Structure of pages folder

These view files such as `Show.tsx` from figure 123 was imported and initialised as a route in the `App.tsx` file. In relation to figure 124, each `Route` created was wrapped inside the `Routes` component provided by `react-router-dom`, which acted as a container for each `Route`. A route component needed a path and the page to present on this chosen path.



```
return (
    <div className="flex bg-background w-full text-foreground min-h-screen">
      {/* Sidebar with a fixed width */}
      {protectedSide}

      {/* Main Content - This will take the remaining space */}
      <div className="flex-1 p-6 overflow-auto">
      <NotificationListener />
        <Routes>
          <Route path="/" element={<Home/>} />
          <Route path="/inbox" element={<InboxPage />} />
          <Route path="/login" element={<LoginPage />} />
          <Route path="/no-team" element={<NoTeam />} />
          <Route path="/register" element={<RegisterForm />} />
          {/* protected routes are all the routes which need authentication */}
          {protectedRoutes}
          <Route path="*" element={<PageNotFound/>}/>
        </Routes>
        {/* <Footer/> */}
      </div>
    </div>
  );
```

Figure 124, Routing

Protected routes were all the routes which need authentication, if unauthenticated tried to access it would redirect to the page not found component. This was done separately for the sidebar component `protectedSide`, as the sidebar needed to be outside the route's container. This modular layout ensured that the routing structure to be scalable and easily managed with authentication.

5.6.7 Calendar

The calendar was the core page within the application, serving as the main component for user interaction. It utilised React-Big-Calendar to provide basic structure and functionality. Keeping the calendar page modular was important to separate each detail which was displayed on the calendar. The calendar page was broken into two folders `components` and `services` folder and a central file, `CalenderPage.tsx`. Figure 125 illustrates this calendar page.

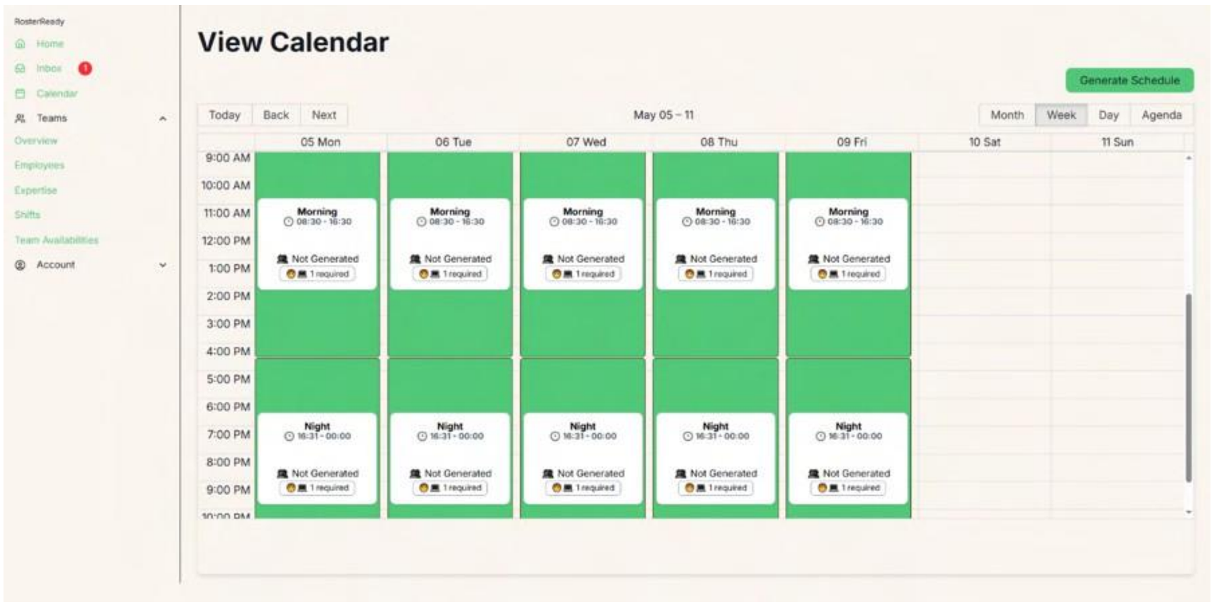


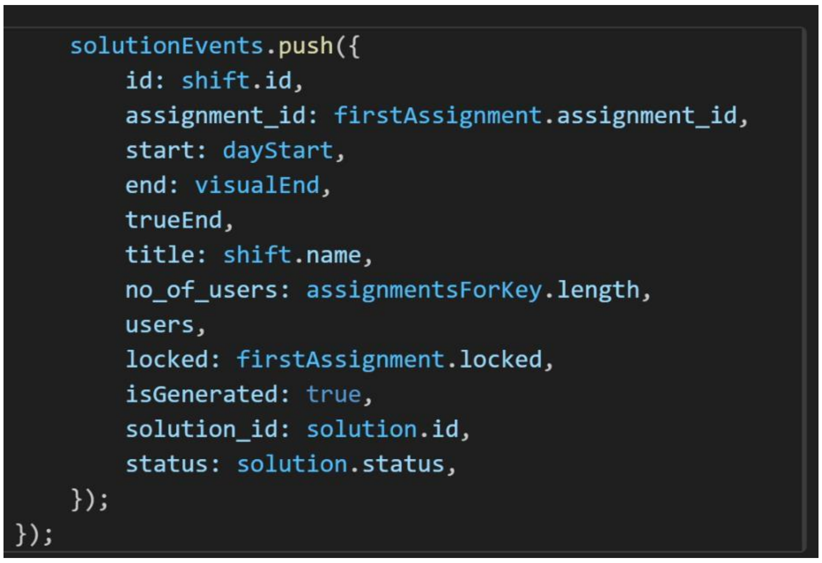
Figure 125, Callender Page Display

5.6.7.1 Calendar main page

The `calendarPage` component serves as the primary interface for managing and displaying the correct shifts or solutions on the correct dates. An asynchronous function called `loadEvents` is placed inside a `useEffect` hook. This function is composed of three separate requests, to retrieve the solutions available, the assignments of each solution and shifts, where solutions are not found.

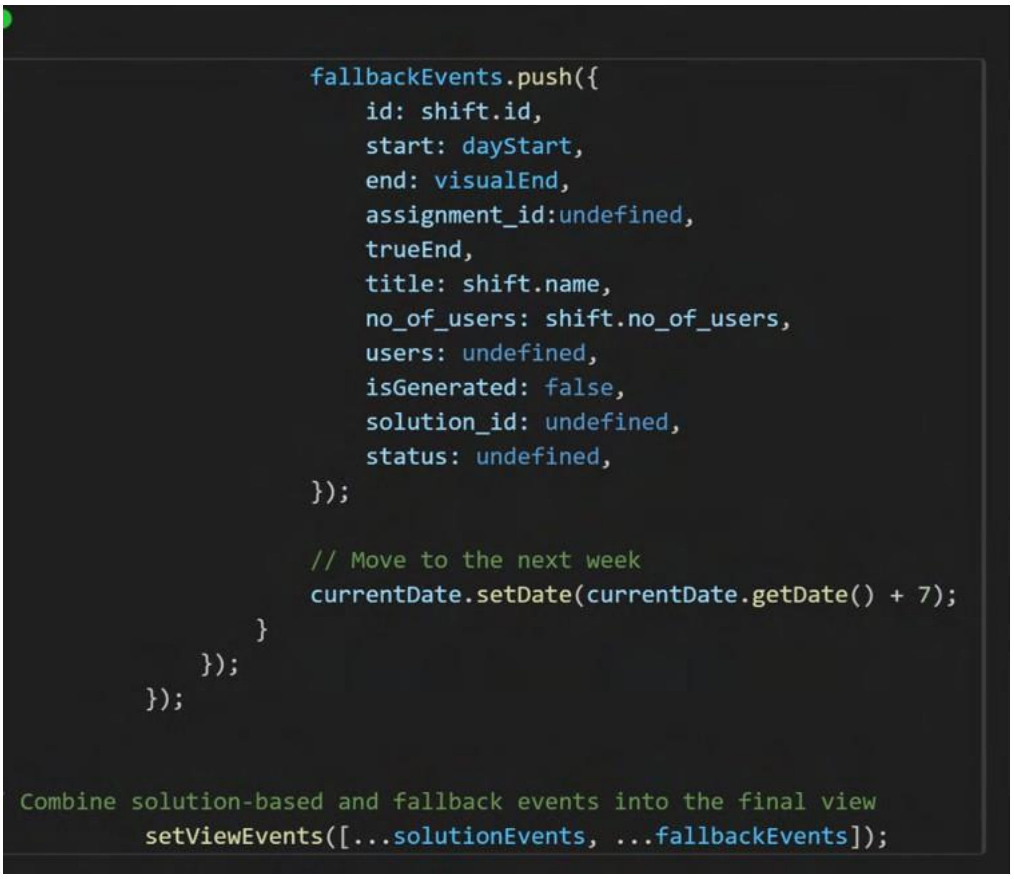
The function begins with retrieving all the solutions and storing them in variable, a loop is placed to extract the start date and end date of each solution from the `week_id` provided. The assignments of each solution are stored in an `assignments` variable, which is then looped to group each assignment by shift and day. Once the assignments are sorted correctly, extraction of the keys is to take relevant data from the assignments and create events for each.

One the solutions are finished; the shift request is called. This function loops through each shift and takes all necessary data to populate the event with the shift data. This function creates shifts for the next six months, this would mean that users can create schedules for the next six months. The solution and shift events are then combined using the spread operator and stored in a state called `viewEvents`. The events are then passed as a prop to the calendar component.



```
    solutionEvents.push({
        id: shift.id,
        assignment_id: firstAssignment.assignment_id,
        start: dayStart,
        end: visualEnd,
        trueEnd,
        title: shift.name,
        no_of_users: assignmentsForKey.length,
        users,
        locked: firstAssignment.locked,
        isGenerated: true,
        solution_id: solution.id,
        status: solution.status,
    });
});
```

Figure 126, Solutions being pushed to events

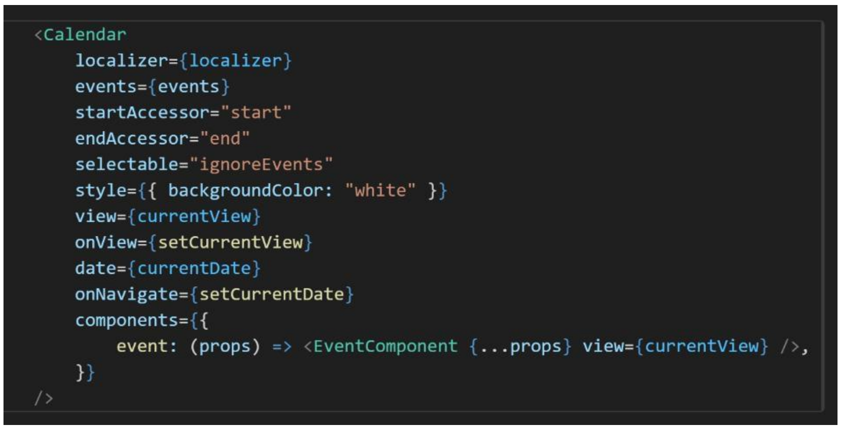


```
                        fallbackEvents.push({
                            id: shift.id,
                            start: dayStart,
                            end: visualEnd,
                            assignment_id:undefined,
                            trueEnd,
                            title: shift.name,
                            no_of_users: shift.no_of_users,
                            users: undefined,
                            isGenerated: false,
                            solution_id: undefined,
                            status: undefined,
                        });

                        // Move to the next week
                        currentDate.setDate(currentDate.getDate() + 7);
                    }
                });
            });

 Combine solution-based and fallback events into the final view
          setViewEvents([...solutionEvents, ...fallbackEvents]);
```

Figure 127, Layout of shifts being pushed to events

5.6.7.2 Calendar Component

Once the events are passed to the calendar component, the component formats these events in relation to the date and the start and end time of the event. Additional features were added which rendered conditionally. If a shift was being displayed on a week view, the calendar would display a generate button. However, if a solution is present for that current week, it would provide three buttons, regenerating the solution, accept and decline. Additionally, the events were also passed the event component. The figure below displays how these events and other variables were passed to react-big-calendar component.



```
<Calendar
    localizer={localizer}
    events={events}
    startAccessor="start"
    endAccessor="end"
    selectable="ignoreEvents"
    style={{ backgroundColor: "white" }}
    view={currentView}
    onView={setCurrentView}
    date={currentDate}
    onNavigate={setCurrentDate}
    components={{
        event: (props) => <EventComponent {...props} view={currentView} />,
    }}
/>
```

Figure 128, Calendar Component

5.6.7.3 Event Component

The event component took these events and formatted them to make them more user friendly. These events displayed the name of the shift, the employees on the shift, the time of the shift and a button which is rendered conditionally. This button only appeared on generated solutions which have a status of "Draft", the button would be used for locking a specific assignment. Once the button was pressed it would send a request to the backend locking that specific assignment and changing the event card visual display. Once these assignments are locked the user can press regenerate solution to build a new schedule around these locked assignments. Figure 128 illustrates how these solutions looked once generated with locked assignments and figure 129 displays how this lock assignment code was implemented.

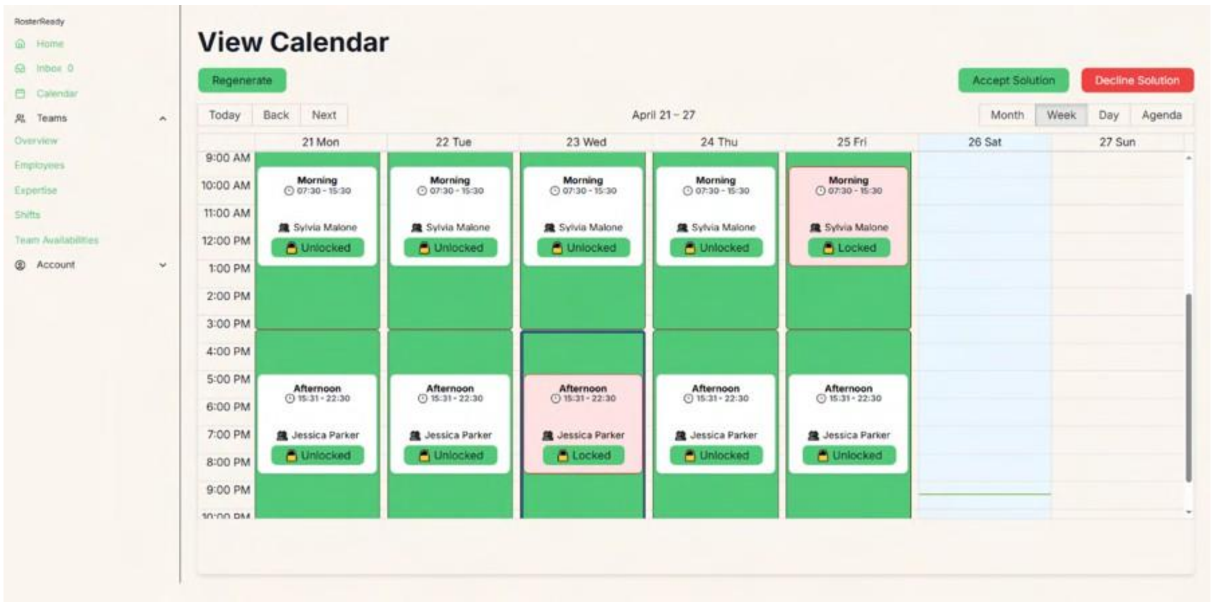
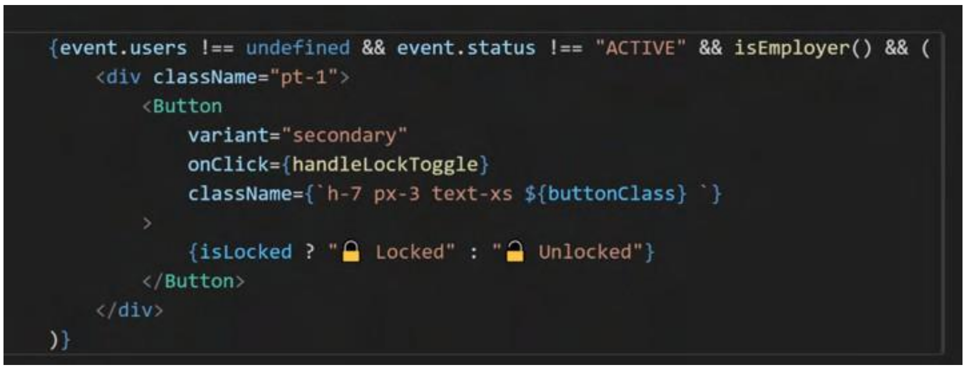


Figure 129, View of locked assignments

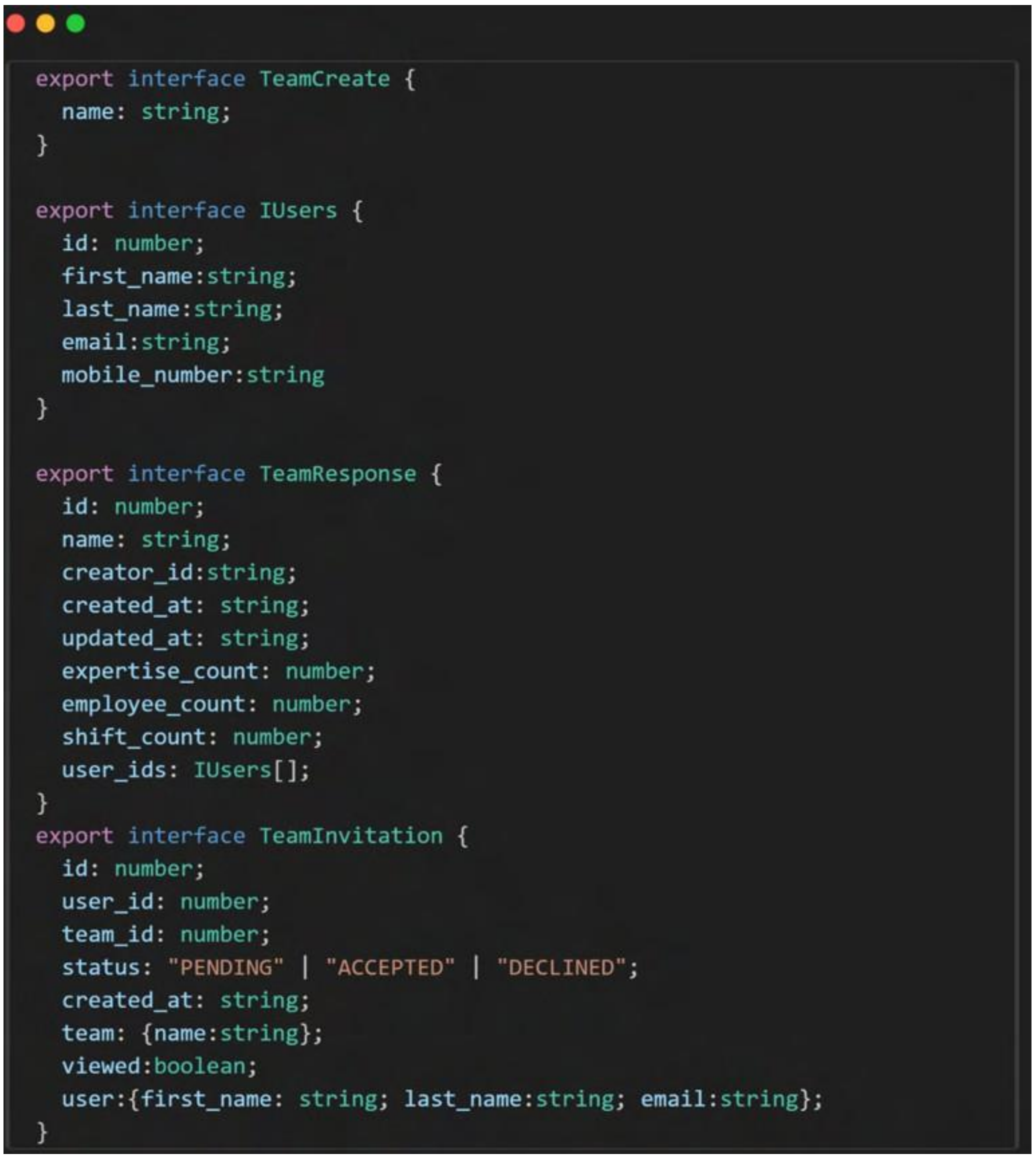


```
{event.users !== undefined && event.status !== "ACTIVE" && isEmployer() && (
    <div className="pt-1">
        <Button
            variant="secondary"
            onClick={handleLockToggle}
            className={`h-7 px-3 text-xs ${buttonClass} `}
        >
            {isLocked ? "🔒 Locked" : "🔓 Unlocked"}
        </Button>
    </div>
)}
```

Figure 130, Locked Code Implementation

5.6.8 Types

Throughout the development of the application, TypeScript was implemented to provide clarity and maintainability of the code. All the types and interfaces were stored within the “types” folder, each page had their own dedicated file for types and interfaces. This structure allowed for types to be reused and updated easily. Figure 131 illustrates a part of the team file which included all relevant interfaces.

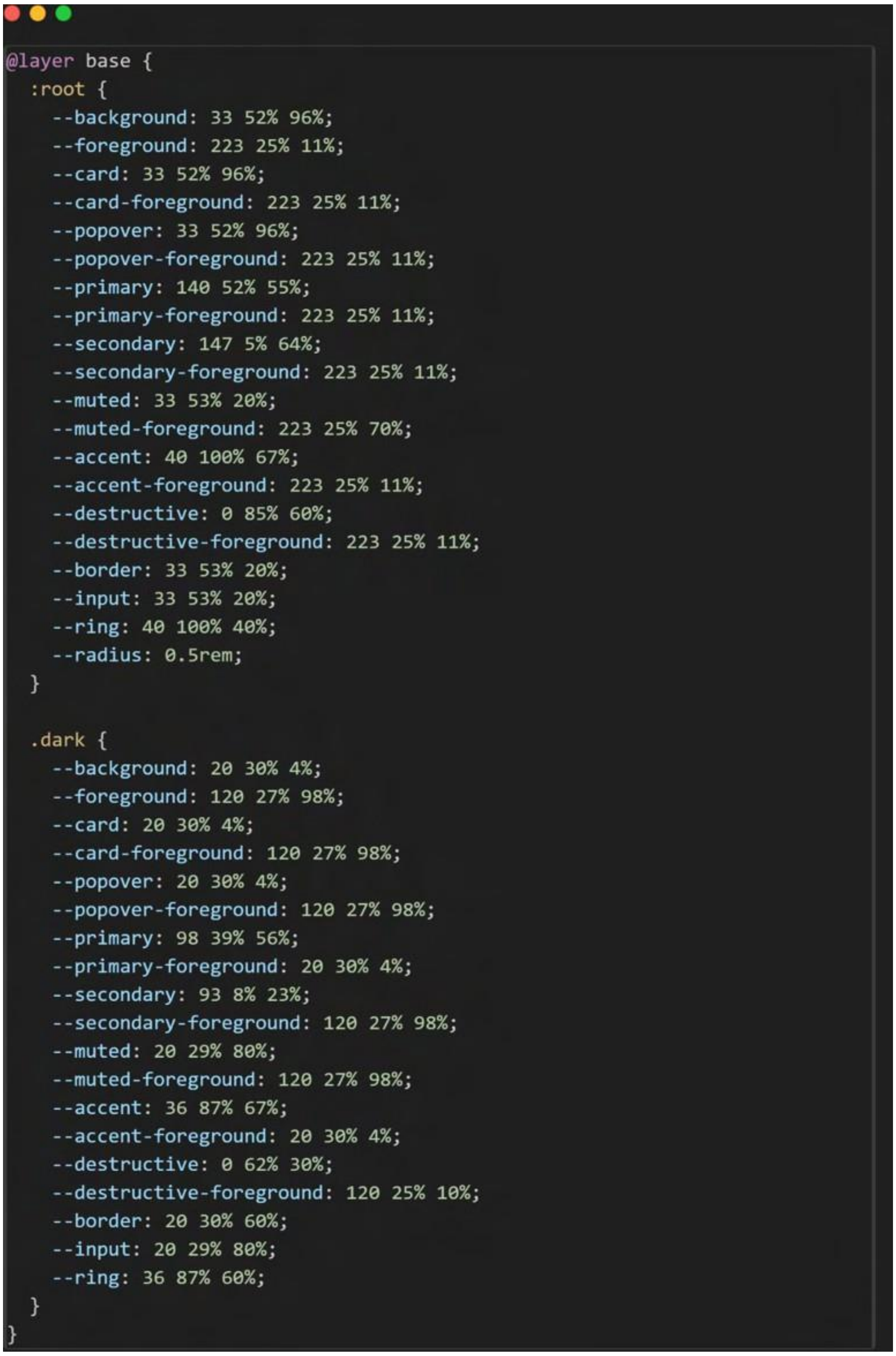


```
export interface TeamCreate {
  name: string;
}

export interface IUsers {
  id: number;
  first_name:string;
  last_name:string;
  email:string;
  mobile_number:string
}

export interface TeamResponse {
  id: number;
  name: string;
  creator_id:string;
  created_at: string;
  updated_at: string;
  expertise_count: number;
  employee_count: number;
  shift_count: number;
  user_ids: IUsers[];
}
export interface TeamInvitation {
  id: number;
  user_id: number;
  team_id: number;
  status: "PENDING" | "ACCEPTED" | "DECLINED";
  created_at: string;
  team: {name:string};
  viewed:boolean;
  user:{first_name: string; last_name:string; email:string};
}
```

Figure 131, Team Type File

5.6.9 Design

The design components were implemented during the implementation of each component and page. The colour palette, typography and animations were implemented once the application had all routes and functional requirements implemented. Shadcn offered a very easy transfer of the colours and typography, by modifying the index.css file to have the appropriate colour palette displaying using HSL colour format and typography was imported from google fonts. Figure 132 illustrates how these base colours were imported to the application using HSL colour format.



```css
@layer base {
  :root {
    --background: 33 52% 96%;
    --foreground: 223 25% 11%;
    --card: 33 52% 96%;
    --card-foreground: 223 25% 11%;
    --popover: 33 52% 96%;
    --popover-foreground: 223 25% 11%;
    --primary: 140 52% 55%;
    --primary-foreground: 223 25% 11%;
    --secondary: 147 5% 64%;
    --secondary-foreground: 223 25% 11%;
    --muted: 33 53% 20%;
    --muted-foreground: 223 25% 70%;
    --accent: 40 100% 67%;
    --accent-foreground: 223 25% 11%;
    --destructive: 0 85% 60%;
    --destructive-foreground: 223 25% 11%;
    --border: 33 53% 20%;
    --input: 33 53% 20%;
    --ring: 40 100% 40%;
    --radius: 0.5rem;
  }

  .dark {
    --background: 20 30% 4%;
    --foreground: 120 27% 98%;
    --card: 20 30% 4%;
    --card-foreground: 120 27% 98%;
    --popover: 20 30% 4%;
    --popover-foreground: 120 27% 98%;
    --primary: 98 39% 56%;
    --primary-foreground: 20 30% 4%;
    --secondary: 93 8% 23%;
    --secondary-foreground: 120 27% 98%;
    --muted: 20 29% 80%;
    --muted-foreground: 120 27% 98%;
    --accent: 36 87% 67%;
    --accent-foreground: 20 30% 4%;
    --destructive: 0 62% 30%;
    --destructive-foreground: 120 25% 10%;
    --border: 20 30% 60%;
    --input: 20 29% 80%;
    --ring: 36 87% 60%;
  }
}
```

Figure 132, Colour imports to CSS file

5.7 Conclusion

The implementation section resulted in a successful development of the full-stack automated scheduling application. The backend was carefully structured with the necessary routes, models and business logic, while an optimal and flexible constraint satisfaction problem (CSP) solver was developed to efficiently process and retrieve relational data from the SQL database created. The optimalisation of the CSP provided an optimal solution within seconds.

The backend routes were tested using Insomnia and Swagger UI which was provided by Fast API, this provided instant feedback on routes and functionality within each route. Once all routes and functionality were implemented successfully the frontend development had commenced.

The implementation of the authentication and regular CRUD operations went smoothly especially when the code structure was modular. Technical challenges did arise in the development of the frontend in relation to the CSP. The use of Scrum methodology with its emphasis on iterative development and flexibility, provided space to accommodate and overcome challenges like these. The implementation phase remained highly manageable, while leaving enough time for evaluation of code and testing the application.

6 Testing

6.1 Introduction

Testing was crucial to ensure the application met the functional requirements (3.3.4) and provided a seamless user experience which matched the non-functional requirements (3.3.5). This section explores the two types of testing methods which were applied, functional testing and user testing. Both testing approaches were essential in validating the application, with functional testing focusing on the technical perspective and user testing focusing on a user's perspective.

Functional testing ensured that the application responds correctly in relation to specification and requirements of the user. This project utilised white-box tests for the backend routes and methods. The use of Insomnia provided a structured environment for simulating requests to various endpoints while analysing the responses.

User testing ensured that each tester was able to achieve the defined goal. The testers' actions were closely monitored by tracking several metrics, including the number of misclicks, time taken to reach the goal and whether the goal was successfully completed. Feedback from each tester was given at the end of the session, which provided insights on what components needed to be refined.

6.2 Functional Testing

Functional testing was applied during the implementation of the backend. Each route was organised into modular folders, containing each method and function which were implemented within the route. This iterative process of testing allowed for instant feedback on the creation of each route and function, which eliminated any bugs or errors immediately.

6.2.1 Validation of Middleware's

The testing process involved simulating various user scenarios to ensure that the backend correctly handled validation of these middleware's with different types of requests. These users were divided into categories based on the middleware functions applied, including authentication, creator identification, user roles and team associations. This ensured that each route was tested under each middleware function, verifying the control logic of the route was as intended.

The application contained 47 implemented methods, of which 43 required testing the validation middleware functionality. Certain methods in specific routes required every middleware validation. To illustrate this testing process, three structured tables were created, each demonstrating the same methods with different types of users. Each table displays the route name, URL, expected output and the actual output, providing the middleware validation outcomes.

6.2.1.1 User 1

User 1 details; {id: 1, team_id:1, Roles: “Employer”}

Route	**URL**	**Expected Output**	**Authentication**	**Team Association**	**Creator**	**Required Roles**	**Actual Output**
Create Shift	. POST /shifts	201 Created	PASS	PASS	PASS	PASS	201 Created
Edit Team	. PUT/teams/2/edit	400 Bad Request	PASS	FAIL	FAIL	PASS	400 Bad Request
Create Availability	. POST /available	201 Created	PASS	SKIP	SKIP	SKIP	201 Created
Generate Schedule	. POST /schedule/assign-shifts/1/1	201 Generated	PASS	PASS	PASS	PASS	201 Generated

6.2.1.2 User 2

User 2 details; {id: 2, team_id:2, Roles: “Employer”}

Route	**URL**	**Expected Output**	**Authentication**	**Team Association**	**Creator**	**Required Roles**	**Actual Output**
Create Shift	. POST /shifts	201 Created	PASS	PASS	PASS	PASS	201 Created
Edit Team	. PUT/teams/2/edit	201 Updated	PASS	PASS	PASS	PASS	201 Updated
Create Availability	. POST /available	201 Created	PASS	SKIP	SKIP	SKIP	201 Created

Generate Schedule	. POST /schedule/assign-shifts/1/1	403 Forbidden	PASS	FAIL	FAIL	PASS	403 Forbidden

6.2.1.3 User 3

User 3 details; {id: 3, team_id:2, Roles: “Employee”}

Route	URL	Expected Output	Authentication	Team Association	Creator	Required Roles	Actual Output
Create Shift	. POST /shifts	403 Forbidden	PASS	PASS	FAIL	FAIL	403 Forbidden
Edit Team	. PUT/teams/2/edit	400 Bad Request	PASS	PASS	FAIL	FAIL	400 Bad Request
Create Availability	. POST /available	201 Created	PASS	SKIP	SKIP	SKIP	201 Created
Generate Schedule	. POST /schedule/assign-shifts/1/1	403 Forbidden	PASS	FAIL	FAIL	FAIL	403 Forbidden

6.2.2 CSP Functional Tests

The constraint satisfaction problem (CSP) solver was vital to the functional testing process, as the application was built around this algorithm. Each part of the CSP had to be tested including, verification of the data imported, data processed, and ensuring a valid solution was returned by this algorithm. This testing approach was critical to evaluate the accuracy and computational performance of the CSP.

6.2.2.1 Data Imported and Processed

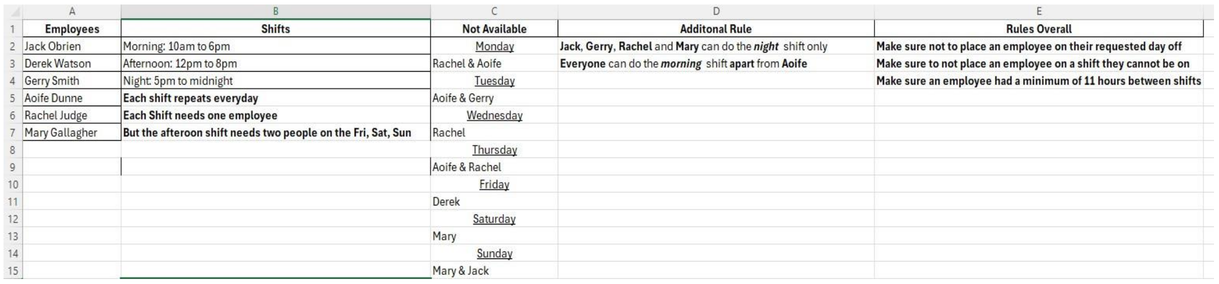
The data that was imported and passed to the CSP was in relation to the `team_id` which was passed in the URL of the route. This was done by filtering within the `schedule_crud.py` file, the data received was then passed to the solver. The data that was being passed to the CSP class was firstly printed in the terminal to review the structure of the data. This was done by running a built-in python function called print, `print(request_data)`.

Once the imported data satisfied the representation of the database records, this data that was being processed within the CSP had to be tested. The testing was done by using the print function within each function in the CSP to identify the steps the algorithm is taking. This was to see what optimisation could be done to further improve accuracy and decrease computational complexity.

6.2.2.2 Valid Solutions

To test the validity of the CSPs generated schedule was accomplished by manual inspection of the database tables. Each individual assignment was reviewed to ensure that none of the enforced constraints were violated. This cross-checking test provided that the CSP was creating solutions successfully.

Additionally, two schedules were created using the same variables, domains and constraints at the same time. The first schedule was created within the application by passing the data to the CSP, the second schedule was created by a tester which regularly creates schedules for their business. Both schedules were measured by the time it took to complete, while comparing the results and checking if no constraints were broken through manual inspection of both schedules. The Information about the business and the constraints can be found in the figure 133 below.

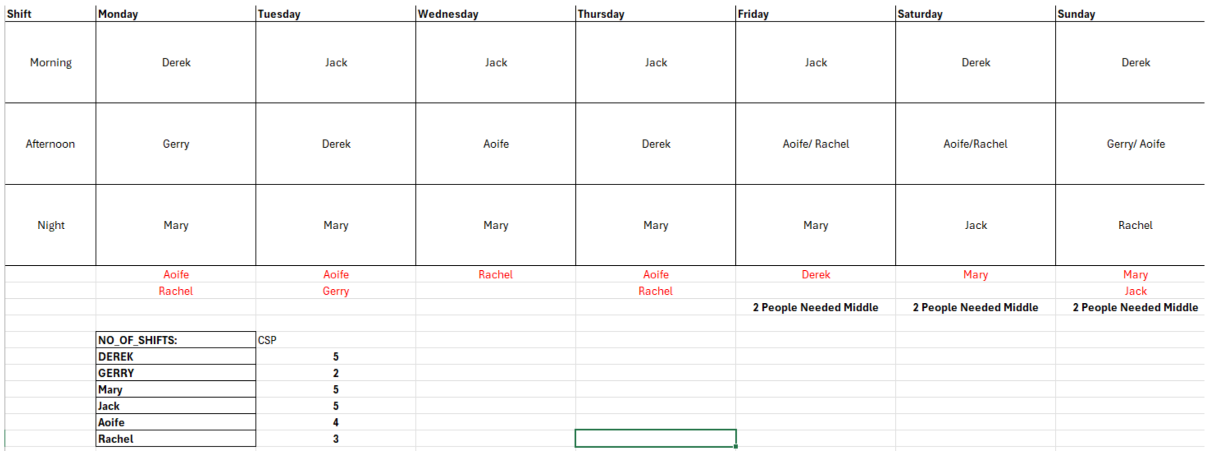


	A	B	C	D	E
1	**Employees**	**Shifts**	**Not Available**	**Additonal Rule**	**Rules Overall**
2	Jack Obrien	Morning: 10am to 6pm	Monday	**Jack**, **Gerry**, **Rachel** and **Mary** can do the ***night*** shift only	**Make sure not to place an employee on their requested day off**
3	Derek Watson	Afternoon: 12pm to 8pm	Rachel & Aoife	**Everyone** can do the ***morning*** shift **apart** from **Aoife**	**Make sure to not place an employee on a shift they cannot be on**
4	Gerry Smith	Night: 5pm to midnight	Tuesday		**Make sure an employee had a minimum of 11 hours between shifts**
5	Aoife Dunne	**Each shift repeats everyday**	Aoife & Gerry		
6	Rachel Judge	**Each Shift needs one employee**	Wednesday		
7	Mary Gallagher	**But the afteroon shift needs two people on the Fri, Sat, Sun**	Rachel		
8			Thursday		
9			Aoife & Rachel		
10			Friday		
11			Derek		
12			Saturday		
13			Mary		
14			Sunday		
15			Mary & Jack		

Figure 133, Information for comparative test

CSP Schedule

The setup of the environment to generate the schedule took 3 minutes and 24 seconds, the generation of a schedule which didn't break constraints took 1.3 seconds. The figure below illustrates the schedule created by the CSP. The CSP schedule was transferred to excel for clearer comparison.

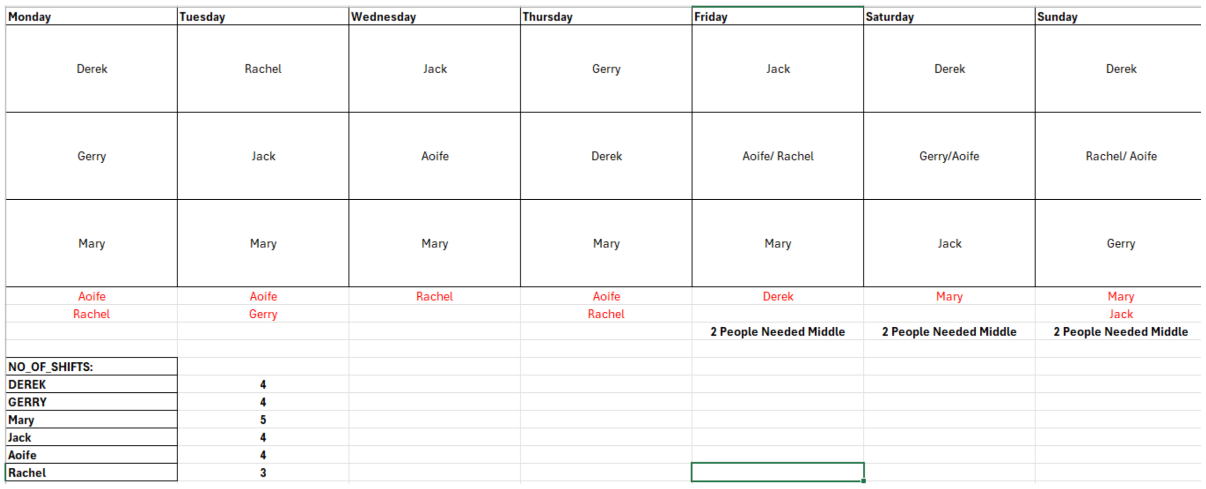


Shift	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
Morning	Derek	Jack	Jack	Jack	Jack	Derek	Derek
Afternoon	Gerry	Derek	Aoife	Derek	Aoife/ Rachel	Aoife/Rachel	Gerry/ Aoife
Night	Mary	Mary	Mary	Mary	Mary	Jack	Rachel
	Aoife	Aoife	Rachel	Aoife	Derek	Mary	Mary
	Rachel	Gerry		Rachel			Jack
					2 People Needed Middle	2 People Needed Middle	2 People Needed Middle
	NO_OF_SHIFTS:	CSP					
	DEREK	5					
	GERRY	2					
	Mary	5					
	Jack	5					
	Aoife	4					
	Rachel	3					

Figure 134, CSP Schedule Test

Excel Spreadsheet Schedule

The setup of the environment was not included within the manual schedule. To create a schedule which didn't break any constraints using excel took 8 minutes and 43 seconds. The user's manual schedule didn't break any constraints. The figure to illustrate the schedule created by the user is seen below.



Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
Derek	Rachel	Jack	Gerry	Jack	Derek	Derek
Gerry	Jack	Aoife	Derek	Aoife/ Rachel	Gerry/Aoife	Rachel/ Aoife
Mary	Mary	Mary	Mary	Mary	Jack	Gerry
Aoife	Aoife	Rachel	Aoife	Derek	Mary	Mary
Rachel	Gerry		Rachel			Jack
				2 People Needed Middle	2 People Needed Middle	2 People Needed Middle
NO_OF_SHIFTS:						
DEREK	4					
GERRY	4					
Mary	5					
Jack	4					
Aoife	4					
Rachel	3					

Figure 135, Manual Schedule Test

6.2.3 Discussion of Results

The results of the functional testing demonstrated that the middleware validation of routes was correctly enforced across each route. The structured tables provided users were either granted or denied access based on their details. The documented core routes provided proof that this consistency was spread across other routes within the application.

Testing the CSP solver further validated the reliability of the core scheduling functionality. Manual inspection of the data being imported and processed provided feedback on preventing unnecessary data being passed and optimise the data being processed to reduce computational power and provide faster results.

The testing on the comparative evaluation of the CSP schedule and manual schedule provided valuable insights on future considerations to make to the CSP. Both schedules looked nearly identical, with a minor change on certain assignments. The generated schedule was superior in terms of speed, however, in relation to optimality it could have provided more of a balanced distribution between users as seen within the manual schedule. Optimality becomes questioned, where additional optimisation techniques could be explored to improve balance without significantly sacrificing computational performance.

6.3 User Testing

User testing was conducted once the implementation of the system architecture was completed. User testing was carried out locally providing in-person feedback collected immediately regarding the usability of the application. Two user types were tested, employers and employees, each with specific goals designed to assess different functionality of the application. Both users were from the same workplace, which meant they could simulate their own schedule. The current schedule is complex with nine employees and various shifts which either have one or two users per shift, dependant on the day of the week.

6.3.1 Employer Role

The user participant was currently working in a management role, dealing with scheduling frequently within their workplace. To provide realistic conditions, the employer provided details on current employees, which enabled seeding the database with mock users before beginning the initial test goal. The user was given two goals to complete. The first goal was to create a team, this involved registering, creating a team and inviting employees. The second goal was to generate a schedule for the next week. The second included sub-goals within such as creating a shift, adding expertise, and accepting any availability within the inbox page, then generating a schedule. A table below is displayed to summarize the outcomes of the users' completed goals.

Goal Number	**Goal Description**	**Steps Taken**	**Time Taken**	**Errors/ Misclicks**	**Outcome**
1	Registration	Filled form, pressed register	32 seconds	0	Success
1	Create a team	Navigated to sidebar, found teams, clicked create	22 seconds	0	Success
1	Invite users x7	Navigated to employees, clicked add	1 minute 47 seconds	2, misspelled email	Success
2	Create Shifts x6	Navigated to shifts, clicked add, filled form	1 minute 53 seconds	Did not add days	Failure
2	Create Expertise x3 and add to required	Navigated to expertise, clicked create, fill form, added users, added shifts	46 seconds	0	Success
2	Accept Availability x8	Navigated to inbox, clicked accept on all	17 seconds	0	Success
2	Generate a schedule for next week	Navigated to team, navigated to calendar, clicked next month, clicked week, clicked	1 minute 3 seconds	2	Success

		generate, clicked accept			

6.3.2 Employee Role

The user was currently working as a part-time employee, working as a cashier in their workplace. This employee user had two goals, which were must basic in relation to the employer goals. The first goal was to accept the invitation received from the employer, for this test the user was presented with login details. The second goal was to create availability requests based on their current situation. A table below is displayed to summarize the outcomes of the users' completed goals.

Goal Number	**Goal Description**	**Steps Taken**	**Time Taken**	**Errors/ Misclicks**	**Outcome**
1	Login	Filled form, clicked enter, clicked submit	20 seconds	1, clicked enter but that does not trigger the submit	Success
1	Accept Invitation	Navigated to Inbox, clicked Accept	15 seconds	0	Success
2	Create Availability	Navigated my availability, clicked on create availability	30 seconds	0	Success

6.3.3 Discussion of Results

The results of the user testing demonstrated that the applications core functionalities are successful by both user types, employers, and employees, with minimal errors. The employer didn't complete a goal fully, which indicates that the design of how the days are added is poor and will need to be solved. The employee did provide a misclick in the login, indicating that an additional function to check if “Enter” key is pressed, to submit the form. Overall, both users remained calm while navigating through the application providing positive feedback on the structure of the sidebar.

6.4 Conclusion

The testing phase was essential in validating both the functionalities and users’ experiences with the application. Functional testing confirmed that the backend routes were correctly implemented with different types of middleware and the CSP solver operated reliably while avoiding passing redundant data to the solver and decreasing the computational complexity during data processing. The comparative evaluation test between the generated schedule and manual schedule supported the solvers efficiency through speed and correctness, while also identifying opportunities for improving optimality with balancing distributions between users.

User testing successfully demonstrated that both user flows were natural with minimal guidance. Positive feedback was received upon the navigation and structure of the UI design. However, minor usability issues did arise during the testing, such as the process of adding days to shifts and login form adjustments. This provided clear areas for future improvements. Overall, the results from both, functional and user testing builds confidence in the deployment of the application within the near future.

7 Project Management

7.1 Introduction

This section outlines the development cycle of the automated scheduling application, detailing each major project phase and the project management tools, and methodology utilised. It begins with exploring the core phases which include, the initial proposal, requirements, design, implementation, and testing. Each phase highlights the deliverables and objectives of each. Scrum methodology provides how it was applied throughout the project. The section concluded with an overview of the project management tools used, outlining GitHub, Miro and Figma and their roles within the project.

7.2 Project Phases

7.2.1 Proposal

The projects proposal phase outlined the problem statement, proposed solution, target audience, core functionalities and potential technical stack. It identified a common issue faced by organisations; the complexity and inefficiency of manually managing employee schedules. A proposed solution was addressed to develop a full-stack application which generates schedules for business, which would be powered by a constraint satisfaction problem solver (CSP) algorithm.

The proposal defined a clear set of objectives, detailing functionalities within the backend and frontend that had to be met to fulfil the proposed solution. Initialising with the proposal phase structured the application of the aims to accomplish, and offering an initial strategy of how it could be successfully implemented.

7.2.2 Requirements

The requirements phase determined the essential needs of the application through a combination of methods. This included analysing similar applications and their functionalities, conducting a survey to understand users' needs and expectations, and carrying out informal interviews with potential users to gain deeper insight into how they manage their current scheduling operations and their biggest challenges. The gathered information provided detailed modeling for user, technical, functional and non-functional requirements.

This phase was crucial in establishing a detailed structure for the capabilities expected from the finished application. It presented a clearer understanding of how the application should be designed in terms of architecture and user experience.

7.2.3 Design

The design phase was dependant on the requirements phase. The design section focussed on two key areas, the program design, and user interface design. The program design included a detailed description of the technologies, including diagrams such as system architecture, database schemas, and process flows. The diagrams provided visual aids of how each technology works independently and as an integrated application. The user interface design concentrated on the visual representation of the application. This included wireframes, style guides, and user flow diagrams to visually describe the navigation each user would have for specific functionality.

The design phase translated the requirements phase to a more detailed and visual understanding. It ensured that both functionality and user experience were addressed before implementation began.

7.2.4 Implementation

The design and requirements phased acted as a blueprint for the implementation phase. This phase included a structured process of implementing certain technologies before others. The implantation began with translating the program design and functional requirements within the FastAPI backend. Once the implementation of the backend was complete the solving algorithm was created within the backend applying optimisation techniques from the research phase. The frontend was implemented once all functionality within the backend was tested and correctly working. This ensured that no back tracking was done when creating the frontend, which allowed for applying the user interface design to the frontend.

The implementation phase couldn’t have been executed without the foundation of design and requirements phase. This phase demonstrated clear success in translating previous phases into a full-stack application which handled generating schedules.

7.2.5 Testing

The testing phase was carried out concurrently with the implementation phase. This allowed for immediate feedback and action with any issues. Two types of tests were conducted, functional and user tests. Functional testing involved white box testing of backend routes, middleware and authentication. Additionally, a comparative evaluation was performed to assess the speed and accuracy of the CSP and a schedule created manually by a user. User testing was conducted after the frontend was implemented. The two testers had a unique roles and goals to complete. This provided insight into the user flow and any feedback on each goal.

This phase was crucial for ensuring the user requirements and functional requirements were met from the requirements phase. This provided additional insights for future implementation within the backend and frontend.

7.3 Scrum Methodology

Scrum is an Agile project management framework that is used to develop software through iterative process. (Al-Saqqa, 2020). Scrum method is based up of events which include sprint planning, sprint reviews and a weekly scrum meeting. The implementation of the Scrum method for this application is crucial as it follows a structured process for the next nine sprints.

This structured process is organised into nine sprints, with each sprint following a consistent and structured approach. The entire Scrum process is visually planned and documented using Miro, a collaborative whiteboard tool. Miro will serve as a central hub to outline sprint goals, sub-goals, and tasks, while also providing a Kanban board to monitor sprint backlog items and the progression of the sprint.

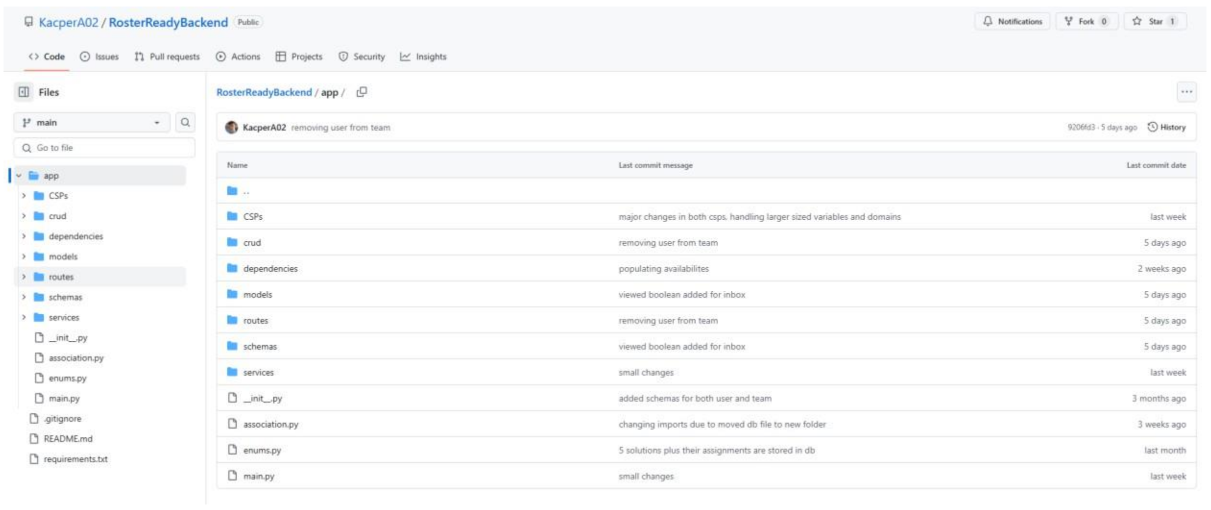
At the end of every sprint, a sprint review is conducted to review the progress and backlog. Any goal within the backlogs which are incomplete for that sprint, are passed to the following sprint, ensuring that nothing is missed. In addition, weekly meetups between myself and the project supervisor are enforced to maintain communication, review the progression and highlight any potential concerns within the development of the automated scheduling application.

Backlogs did occur within the implementation section due to the optimisation techniques which were applied to the CSP. Using a high volume of variables, domains and constraints seemed to be computational for the CSP. This issue was revealed in the testing of the CSP, which pushed the goals behind within that sprint.

7.4 Project Management Tools

7.4.1 GitHub

GitHub was used as the version control tool to manage and track changes across both the frontend and backend codebases throughout the implementation phase. It provided code storage within the cloud, allowing to pull and push the repository when working on another device. Additionally, each repository provides history of commits, messages and errors, this provides proof of iterative process. Figure 136 displays the Backend codebase within GitHub.



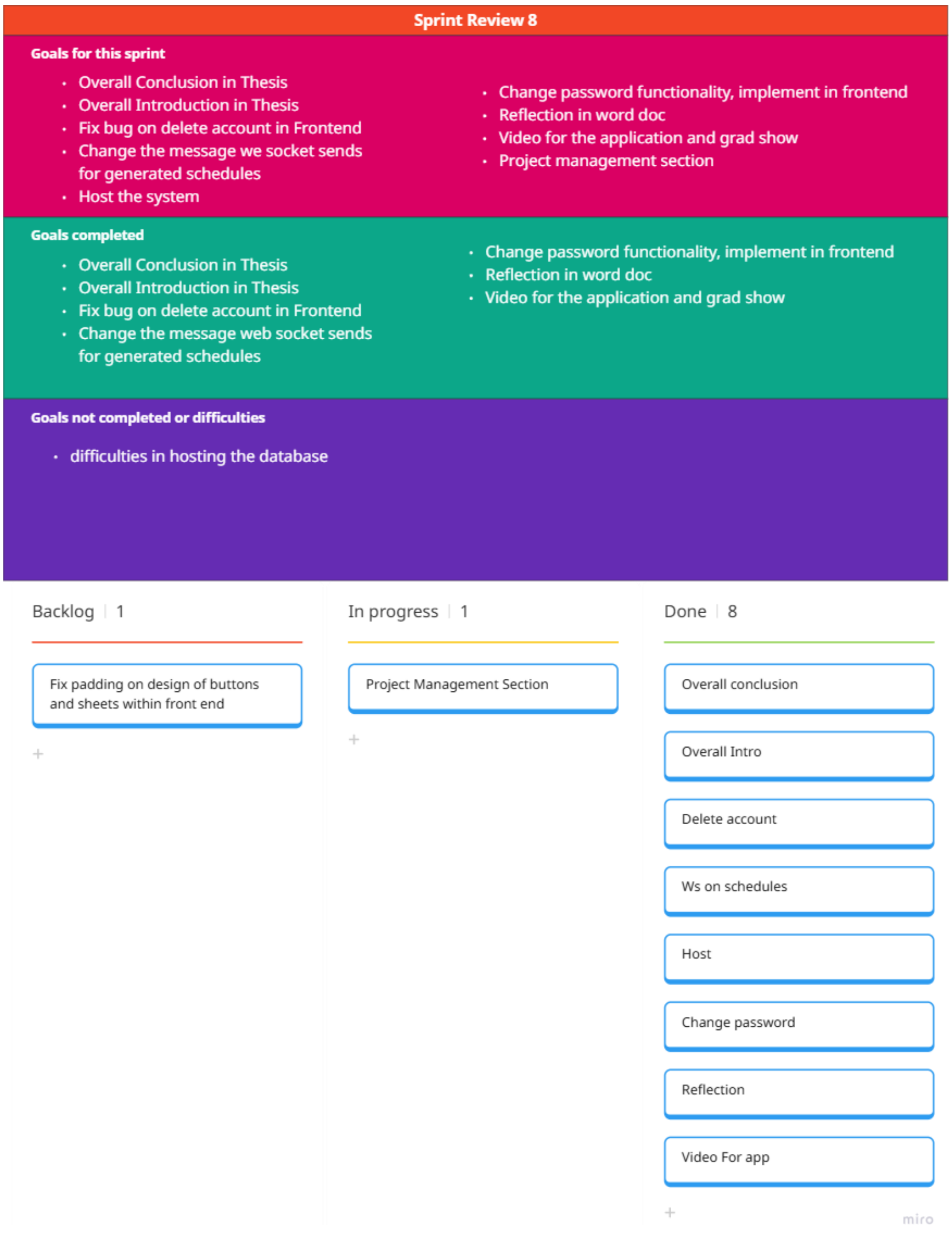
KacperA02 removing user from team 9206fd3 · 5 days ago

Name	Last commit message	Last commit date
..		
CSPs	major changes in both csps. handling larger sized variables and domains	last week
crud	removing user from team	5 days ago
dependencies	populating availabilites	2 weeks ago
models	viewed boolean added for inbox	5 days ago
routes	removing user from team	5 days ago
schemas	viewed boolean added for inbox	5 days ago
services	small changes	last week
init.py	added schemas for both user and team	3 months ago
association.py	changing imports due to moved db file to new folder	3 weeks ago
enums.py	5 solutions plus their assignments are stored in db	last month
main.py	small changes	last week

Figure 136, GitHub Backend Codebase

7.4.2 Miro

Miro is an online collaborative whiteboard platform designed for visual planning and project management. It provides a variety of pre-built templates and components, which were used for simplifying project management. These components and components were Scrum methodology frames, kanban boards and architecture elements. The scrum methodology within included goals to complete, goals completed and goals which were not completed. Each sprint came with a kanban board to track the backlog of each of these goals, which simplified project management. Additionally, Miro was used for creating a variety of architectural designs with its prebuilt components, this includes the system architectural diagram. Figure 137 illustrates the view of sprint nine.



Sprint Review 8

Goals for this sprint

- Overall Conclusion in Thesis
- Overall Introduction in Thesis
- Fix bug on delete account in Frontend
- Change the message we socket sends for generated schedules
- Host the system
- Change password functionality, implement in frontend
- Reflection in word doc
- Video for the application and grad show
- Project management section

Goals completed

- Overall Conclusion in Thesis
- Overall Introduction in Thesis
- Fix bug on delete account in Frontend
- Change the message web socket sends for generated schedules
- Change password functionality, implement in frontend
- Reflection in word doc
- Video for the application and grad show

Goals not completed or difficulties

- difficulties in hosting the database

Backlog 1	In progress 1	Done 8
Fix padding on design of buttons and sheets within front end	Project Management Section	Overall conclusion
		Overall Intro
		Delete account
		Ws on schedules
		Host
		Change password
		Reflection
		Video For app

Figure 137, Miro SS of Sprint 9

7.4.3 Figma

Figma is a tool which used for user interface and user experience designing, including wireframes, prototypes and design iterations. Figma was used throughout the design phase and played a crucial role in managing this iterative design process. Typography, colour palettes and component layouts were continuously refined based on feedback. The platform also offers prebuilt templates within their community, this made it possible to import pre-made UI components to match Shadcn components. Using these prebuilt components accelerated the interface design process. Figure 138 provides an overview of the whole figma file.

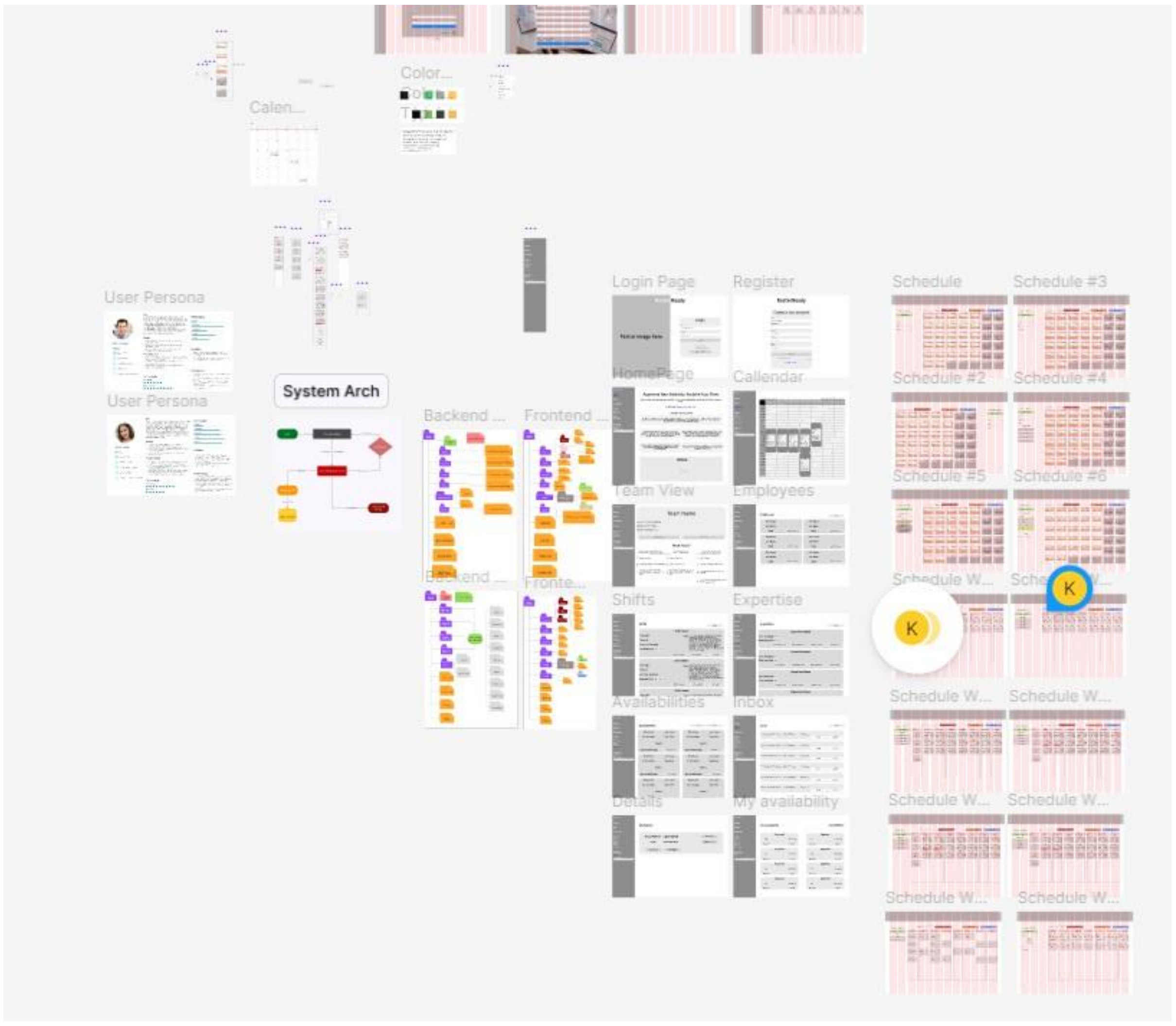


Figure 138, Figma File

7.5 Reflection

7.5.1 My views of the project

Overall, I believe the project was a success. It addressed the complex scheduling problem through a full stack application with its core functionality within the CSP algorithm. However, the scope of the project was quite ambitious, especially considering the challenges of learning an entirely new programming language and framework during the early stages of development. Additionally implementing features such as WebSocket's added to the complexity and required a lot of time to understand and implement the logic.

Developing an entire full stack application independently within a fixed timeframe presented periods of high pressure, particularly during the implementation phase. Despite all the challenges, the project met its core requirements and delivered a solution to the complex scheduling problem. In addition, the project was rewarding as it introduced me to new technical skills and soft skills.

7.5.2 Working with a supervisor

Working with a supervisor brough both challenges and value. At times, finding a time slot was difficult due to my heavy schedule, this presented delays in receiving feedback. However, when our meetings did take place, they were highly productive. Being able to reason with a developer with experience regarding design or technical decisions provided clarity and direction.

7.5.3 Technical skills

From a technical perspective, this project significantly broadened my technical knowledge. I became proficient in Python, despite only beginning to learn the language in September. Using the official FastAPI documentations, I developed a deep understanding of the framework and backend development practices in Python.

I also explored WebSocket's for real-time updates within my application, this required learning new asynchronous programming concepts. The most substantial skill I gained was through research of search algorithms and integrating a constraint satisfaction problem (CSP) solver. Implementing and optimising this algorithm within the backend taught me how to place these theoretical concepts into a functional class.

7.5.4 What was missed

The projects main goals were achieved; however, some features were unable to be fully implemented due to a lack of time. The drag and drop functionality within the frontend calendar component needed to be implemented, alongside breakpoints also weren't fully implemented within each component. Although planned, an issue did arise within the CSP with its optimality, this way I prioritised the core functionality which needed instant attention.

8 Conclusion

The issue around the complexity and time consumption in employee scheduling was successfully solved within a single application, RosterReady. The application provided authentication, role-based access control, real-time updates with WebSocket's, an interactive calendar, and the CSP solver in the backend for scheduling employees to shifts without breaking constraints. The applications logic was handled using the Python framework, FastAPI, with relational data stored in MySQL database, and the frontend was developed using the JavaScript library React.

Managing the project through Scrum methodology allowed for an iterative and structured development process, allowing for feedback at each sprint across all phases. Each phase was essential for developing this full-stack application. The research phase deepened understanding of search algorithms and CSP techniques. The requirements phase gathered valuable input from users and competitors to shape the applications functional and non-functional requirements. The design phase used the requirements to translate this information into a program design and user interface design, resulting with detailed plan. This plan was then transformed into code within the implementation phase, which provided a clear understanding of the code and decision making at each element of the stack. Near the end of the implementation phase, tests were conducted on functionalities and users which provided deficiencies in the application.

The process of each phase and the successful implementation provided many different skills and knowledge. The main skills which I acquired were the technical skills, which include developing algorithmic techniques, understanding and implementing WebSocket's, and learning a new programming language to build the backend effectively.

The application has potential to grow substantially and has many different areas to develop in the future. The main areas which would be considered to develop further would include, the design of the user interface, additional optimisation to the CSP, more control of the CSP by the employer, and transitioning the application from browser-based to a standalone desktop application, as well as extending the application to be available on iOS and android stores.

References

Al-Saqqa, S., Sawalha, S., & AbdelNabi, H. (2020). Agile software development: Methodologies and trends. *International Journal of Interactive Mobile Technologies*, *14*(11).

Barták, R., Salido, M. A., & Rossi, F. (2010). Constraint satisfaction techniques in planning and scheduling. *Journal of Intelligent Manufacturing*, *21*, 5-15.

Bessiere, C. (2006). Constraint propagation. In *Foundations of Artificial Intelligence* (Vol. 2, pp. 29-83). Elsevier.

Connecteam**.** (n.d.). *Connecteam: The World's #1 Employee App.* Connecteam. https://connecteam.com/

Cheng, B. M. W., Lee, J. H. M., & Wu, J. C. K. (1997). A nurse rostering system using constraint programming and redundant modeling. *IEEE Transactions on information technology in biomedicine*, *1*(1), 44-54.

Cheng, C. C., & Smith, S. F. (1997). Applying constraint satisfaction techniques to job shop scheduling. *Annals of Operations Research*, *70*(0), 327-357.

Fox, M. S., & Sadeh, N. M. (1990, August). Why is Scheduling Difficult? A CSP Perspective. In *ECAI* (pp. 754-767).

GeeksforGeeks. (n.d.). *Search algorithms in AI*. GeeksforGeeks. https://www.geeksforgeeks.org/search-algorithms-in-ai/

Homebase**.** (n.d.). *All-in-one Employee Scheduling, Time Clocks, Payroll.* Homebase. https://www.joinhomebase.com/

Javatpoint. (n.d.). *Search algorithms in AI*. Javatpoint. https://www.javatpoint.com/search-algorithms-in-ai

Larksuite. (n.d.). *NP-hard: Definition of NP-hardness*. Retrieved December 27, 2024, from https://www.larksuite.com/en_us/topics/ai-glossary/np-hard-definition-of-np-hardness

Pathak, M. J., Patel, R. L., & Rami, S. P. (2018). Comparative analysis of search algorithms. *International Journal of Computer Applications*, *179*(50), 40-43

Renke, L., Piplani, R., & Toro, C. (2021). A review of dynamic scheduling: context, techniques and prospects. *Implementing Industry 4.0: The Model Factory as the Key Enabler for the Future of Manufacturing*, 229-258.

Rossi, F., Van Beek, P., & Walsh, T. (2008). Constraint programming. *Foundations of Artificial Intelligence*, *3*, 181-211.

Van Beek, P. (2006). Backtracking search algorithms. In *Foundations of artificial intelligence* (Vol. 2, pp. 85-134). Elsevier.

Webcast Departmental []. (28/08/2018). *COMPSCI 188 - 2018-08-28 - Uninformed Search* YouTube. https://www.youtube.com/watch?v=-Xx0QSFYfIQ

Webcast Departmental. (30/08/2018). *COMPSCI 188 - 2018-08-30 - A* Search and Heuristics* YouTube. https://www.youtube.com/watch?v=Mlwrx7hbKPs

Webcast Departmental. (04/09/2018). *COMPSCI 188 - 2018-09-04 – Constraint Satisfaction Problems (CSPs) Part 1/2* YouTube. https://www.youtube.com/watch?v=81z2ANjQcH4

Webcast Departmental. (06/09/2018). *COMPSCI 188 - 2018-09-06 – Constraint Satisfaction Problems (CSPs) Part 2/2* YouTube. https://www.youtube.com/watch?v=_DXf6oaknHw