Proactive Order Management System

Stephen Fox
C13475462

Supervisor: Dr. Basel Magableh
Second Reader: Dr. Bianca Schoen-Phelan

School of Computing
Dublin Institute of Technology

4th April 2017

Abstract

This report details the research, design, implementation, and results of a final year project which aimed to create a proactive order management system (see Definition 1.1). The proposed system's objective is to help businesses handle orders by analysing data on the orders in a system and providing a business with information on when to process these orders. The orders are put into the system remotely by a mobile application that allows customers to make orders.

This report introduces a new method of task scheduling which can advise employees of a business on the sequence to process the orders which have been put into the system via the mobile application. The orders are assumed to contain food items, where the time the order is prepared is important.

The report also highlights how NuPIC (Numenta's Platform for Intelligent Computing) an open source machine intelligence project is used to help a business by predicting the expected max number of employees needed per hour and the expected number of orders per hour.

Acknowledgements

Firstly I would like to thank my supervisor Dr. Basel Magableh for his time, support and advice throughout this project, without your guidance and recommendations the outcome of this project would not have been possible.

I also want to thank my family and friends, especially my parents for always supporting and believing in me throughout this project and the past four years.

I would like to thank Andrew from Urbanity Cafe for offering your time to meet with me to discuss this project and offering feedback.

I would like to thank Astrid for her patience and support during the past few months of this project.

Declaration

Declaration

I **Stephen Fox** hereby declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed:

Stephen Fox 4th April 2017

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

1.1 Motivations of the project

The motivation behind this project is to introduce a new way for customers and businesses to interact. The method of interaction will allow customers to make orders via mobile application and allow businesses to view these orders through a web application. By ordering through a mobile application the ordering process could be automated for the customer and business. Along with order automation, the customer's location will be tracked to provide a business with real-time data about the deadlines for all orders in the system. This real-time data will provide the ability for the system to schedule orders so that employees can see which orders have the highest priority at any given time, calculation of how many employees will be needed at certain times and assigning orders to available employees. This data will be used with the aim to make the interaction between customer and business less time consuming and more efficient for both parties.

1.2 Project Objectives

1.2.1 Overview

The objective of this project is to build a proactive order management system (the term proactive is defined in Definition 1.1 in Section 1.2.5), that is further enhanced with prediction data using NuPIC. The objectives are defined below.

1.2.2 Allow orders to be made remotely via mobile application

This project should allow customers to make orders remotely from a business through a mobile application and have the orders scheduled so they are as close to ready as possible when a customer arrives at a business. Customers should be able to choose their travel method to the business and based on their arrival time their order should be scheduled appropriately.

1.2.3 Product management

A business should be able to manage their products through a web interface. A business should be able to configure different options for each product such as size etc and update and edit products too.

1.2.4 View Orders

A business should be able to view all their orders in the system. A business should also be able to view, for each order, the price, contents, creation time, deadline time and the employee who the order is assigned to for processing.

1.2.5 Proactive

For this report the definition of proactive should be considered as the one defined in Definition 1.1.

Definition 1.1 *Proactive The process of scheduling orders, task allocation, calculation of employees needed to process tasks and generating a utilisation status.*

The exact terms used in the definition are explained below.

Schedule orders

As orders arrive into the system they will be scheduled according to a specific time. This time will be the customer's arrival time at a business to collect the order. The scheduling of the orders will take inspiration from the Earliest Deadline First algorithm.

Orders are allocated to available employees

As orders are scheduled, they should also be allocated to the employees of a business appropriately so that they are achieved on time. Each employee will be given tasks according to their working hours and according to the maximum amount of tasks they can process simultaneously. The term processing of orders with regards to employees means an employee is "making", "preparing" or "fulfilling" an order so that it is ready on time for a customer to collect.

The maximum number of employees needed to process a set of orders can be calculated

When there is a set of orders that need to be processed, the system should be able to advise the business on the number of employees needed to process the orders, this is to help businesses with achieving all tasks by their deadlines.

Generate a utilisation status

From the scheduling algorithm and task allocation described in Section 4.4 a utilisation status for a business should be generated for example if the business is extremely busy it may get a "very busy" status or "busy" status. This should be viewable from the mobile application so customers know if a business is busy or not before they make an order.

1.2.6 Integrate NuPIC

The system should also be able to use prediction data generated from NuPIC[1] which is an open source machine intelligence project. The prediction data should further enhance the proactive aspects of the system. These predictions should include the expected orders for a business during each hour of a day and the max number of employees needed for each hour of a day.

1.3 Project Challenges

The main challenges of this project include development of a scheduling algorithm, allocation of orders to employees, developing a technique to estimate the max number of employees needed during certain periods of the day where orders are scheduled. Integrating NuPIC into the system so the predictions about employee counts and

[1] https://github.com/numenta/nupic

order counts per hour can be made is also another area of challenge due to no previous experience with NuPIC.

Identifying a task scheduling algorithm that can be used to advise employees when orders should be processed and in what sequence will be challenging. Many scheduling algorithms are suited for operating systems and CPUs to schedule tasks with temporal accuracy of milliseconds down to nanoseconds such as the Earliest Deadline First algorithm (EDF) [2]. Most scheduling algorithms and existing literature surrounding scheduling are related to real time systems, for example EDF which is designed for hard real time systems [2], where failure to process a task before its deadline is disastrous [3]. Real time systems are discussed in Section 2.4. This means a scheduling algorithm will have to be developed that can schedule food orders more appropriately for humans which does not have the same temporal constraints that most scheduling algorithms have.

Allocating orders correctly to employees is also another challenging area as different employees work different shifts knowing how to distribute tasks correctly between workers will be essential to having the orders finish by the correct deadlines.

Developing a way to estimate how many employees that will be needed at each period of the day where an order is scheduled is also another challenge.

Lastly, learning how to use NuPIC so it can analyse data from the system to make predictions is another area of challenge. NuPIC is an extremely powerful machine intelligence piece of software with many capabilities such as pattern recognition and predictions. Correctly parsing and modifying data so it can be used by NuPIC will be a challenging area as well as learning and knowing how to correctly use NuPIC to extract the data needed for this project.

1.4 Document Structure

This section outlines the content in the upcoming chapters.

1.4.1 Research - Chapter 2

In Chapter 2, research will be conducted into similar systems, technologies, scheduling algorithms, real time systems and the interval tree data structure.

1.4.2 Design & Architecture- Chapter 3

In Chapter 3, the design of the system will be discussed including the overall architecture, use cases and database design.

1.4.3 Implementation - Chapter 4

In Chapter 4, the implementation of the system will be presented. The implementation will include how each of the objectives in Section 1.2 are implemented.

1.4.4 Testing - Chapter 5

In Chapter 5, the testing that is performed on the system is discussed.

1.4.5 Evaluation - Chapter 6

In Chapter 6, the evaluation methods and results are presented.

1.4.6 Conclusion - Chapter 7

Chapter 7, provides a conclusion to the project along with the learning outcomes and future work.

Chapter 2

Research

2.1 Background

Order Management Systems (OMS) are a necessary part to any business that sells products, online or in store. Most OMS provide several functionalities for businesses including handling clients, managing orders, monitoring inventory and tracking orders. Each OMS provides solutions for different businesses from small to large with different needs and budgets, Software Suggest [4] lists several different OMS that currently exist for different industries.

In the food and drink industry technology in recent years has become as Rosenheim Advisors [5] describe 'piping hot with $6.8 billion of capital flowing into private companies' in the food tech industry. In 2015 there was over just over $1billion raised, up from $760 million [5] in 2014 in the U.S. for the food tech industry. In Ireland there are many food tech companies such as JustEat[1] and Deliveroo[2] that are competing in the online food ordering business. Companies like JustEat and Deliveroo provide the ability to order food and drink online through mobile and web applications and have these orders delivered directly to a customer. While JustEat does allow for orders to be collected, orders must be a minimum of €10 in cost. It is possible there is a niche for order collections of any price from businesses where Deliveroo and JustEat don't service.

[1] http://www.justeat.ie
[2] http://www.deliveroo.com

2.2 Similar Systems

2.2.1 Starbucks Order and Pay

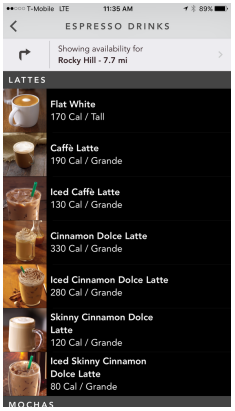

Figure 2.1: Starbucks Order and Pay product page

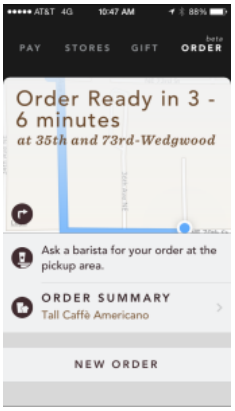

Figure 2.2: Starbucks Order and Pay order page

One software solution that has some similar traits to this project is Starbucks Coffee Order and Pay[3] mobile application. Starbucks Coffee Order and Pay mobile application is available on both iOS and Android, however, this application currently is not available in Ireland. It allows customers to order and pay for a coffee and collect it when they arrive in store. Once a customer makes the order, they are told how long until the order is ready and the mobile application will guide the customer via map to a Starbucks store. It provides the ability for Starbucks customers to order their coffee remotely and collect. The application allows customers to view the many coffees on offer from Starbucks in a menu. For each coffee, the customer can select different options for each product, such as size or toppings. Once the order has been received by the store, a time is given until the order will be ready, which can be seen in Figure 2.2 along with a map and direction to the store.

[3]https://www.starbucks.com/coffeehouse/mobile-order

One possible issue with the Starbucks Coffee Order and Pay application would be not knowing when the customers will arrive to collect their order. An article by Fortune [6] mentions problems that arose for its barista's because of this, saying 'cafes had difficulty keeping up with mobile orders in the latest quarter, creating bottlenecks at drink delivery stations and leading some walk-in customers to walk out'. By scheduling orders based on when a customer will arrive to a business and allocating them to employees, a business should be able to foresee when bottlenecks may occur.

2.2.2 ChowNow

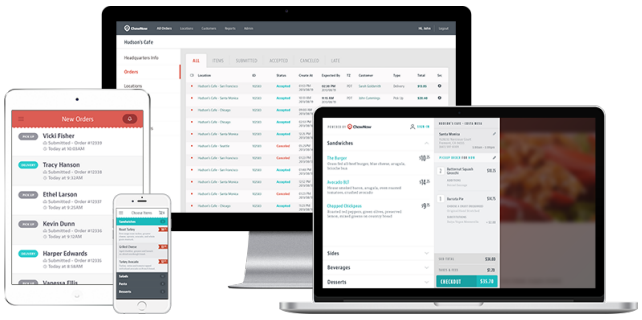

Figure 2.3: ChowNow

ChowNow[4] provides an order management solution for restaurants as well as iOS and Android applications for customers to make orders from. They provide an order management system that can be used behind a business's website to take orders as well as take orders through Facebook, Google and other platforms. They also provide each business with a tablet for Android or iOS to manage these orders in store through the website. They build mobile applications specifically for each restaurant to allow customers to order from.

The proactive order management system discussed in this report should allow customers to find businesses through one central mobile application and order, instead of customers needing many mobile applications for each business.

2.2.3 Case Study

Case Study: Urbanity Cafe Date Conducted: 27 November 2016

Urbanity Cafe (Urbanity) is a cafe store situated in Smithfield, Dublin 7. It is a relatively new start up cafe which opened for business in February 2016. It serves

[4]https://www.chownow.com

a large selection of both coffee and food. After speaking with a barista in Urbanity Cafe named Andrew, some more information was gathered about order managements systems in the real world.

Current System

Urbanity uses AIB Clover[5] for their order management system. AIB Clover is an electronic point of sale (EPOS) system. Some of its features include taking payments via credit/ debit card, view and edit information about products, as well as view sales and ordering data. All this data can be accessed through a tablet application and is connected to their AIB business account. Focusing on the order management side, it allows Urbanity to view all historic ordering data over certain time periods as well as some analytics on this data.

Current Problems

One notable problem Urbanity faces is customers ringing in and ordering food and drink, which is problematic as when customers estimate their time of arrival it is usually wrong. The order that is prepared becomes stale the longer it is not consumed. This is a problem where this project will address; by analysing the users location and distance when they order their time of arrival can be estimated using Google's Maps Distance Matrix API, which is discussed in later sections. This will allow for orders to be scheduled and allocated to employees to process. This method could possibly replace ring-in orders for Urbanity and other businesses alike. While one of the requirements for scheduling orders would be knowing how long each item of the order would take to process, Andrew advised that only experienced members of staff would know the amount time it would take to process each item they sold.

The ring-in method of ordering is also troublesome for employees of Urbanity as they have to remember and correctly write down the order, and then not forget to process it. The ordering page for this project should remove the ambiguity of misinterpreted phone orders by displaying the orders through a user interface. Phone orders are also time consuming as it usually requires an employee to explain the menu as well as prices and then take the order. Using this system employee interaction with customers will be reduced as employees who used to take phone orders can focus their energy on

[5] http://www.aibms.com/products-and-services/clover/

only needing to process the orders at the time scheduled by the system, as customers can view all menu information and prices through the mobile application.

2.3 Technologies Researched

2.3.1 Backend Web Framework (Node.js vs Django vs Flask)

The initial technological research began with the backend of the system, specifically a web framework which could handle HTTP requests as each client, from the mobile application to the web application need to communicate with the back end. As the back end web API is the central point of communication for all clients, a uniform interface is needed in which they can interact with.

Nowadays there are many server side technologies one could use for developing web apps, from Google's Node.js to other open source technologies such as Django[6] from the Django Software Foundation or Flask[7] which is a micro framework developed by Armin Ronacher. Each one with their own mantra and methodologies, providing developers with different tool sets and architectures to work with. Both Django and Flask are written in and use Python, whereas Node.js uses JavaScript which runs on Google Chromes V8 JavaScript Engine [7].

All three technologies allow for the use of Representational State Transfer (REST), which makes it ideal for creating the web application programming interface (API), as each client can use HTTP methods GET, PUT, POST, DELETE, PATCH etc to interact with resources. Each technologies have their own framework for using REST. For Node.js one can use Express[8] which is described as 'a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications' [8]. Django REST Framework[9] is available for Django and has a rich set of features from authentication policies to object relation mapping and extensive documentation. Lastly, Flask has Flask-RESTful[10] which is an extension for Flask that adds support for quickly building REST APIs.

While investigating each technology, Node.js with Express and Django seem to have the best documentation and support for what is needed, along with numerous tutorials

[6] https://www.djangoproject.com/
[7] http://flask.pocoo.org/
[8] https://expressjs.com/
[9] http://www.django-rest-framework.org/
[10] https://flask-restful.readthedocs.io/en/0.3.5/

and guides. Similarly Flask, has very detailed documentation, but a much smaller community than Node.js or Django. This trend is also visible online at a website called Hot Frameworks [9] which tracks popularity of each web framework. Table 2.1, shows each web frameworks popularity across GitHub and StackOverflow for the month of November 2016.

Framework	Github Score	Stack Overflow Score	Overall Score
Express	93	80	86
Django	90	93	91
Flask	91	75	83

Table 2.1: Web Framework popularity from http://hotframeworks.com

The decision came down to wanting to learn more about JavaScript and finding Node's package manager NPM[11] much easier to use than PIP with Django, therefore Node.js was chosen as the main backend web framework. (Note in Section 2.3.5 Flask is chosen to provide the proactive module with a HTTP interface for communication)

2.3.2 Frontend Web Application (AngularJS vs React)

For the front end web application there are two technologies that stand, Google's AngularJS[12] and Facebook's React.js[13]. Both allow for front end web development using JavaScript and HTML, however AngularJS also allows the use of TypeScript and Dart. AngularJS, however, is a framework and React is a library, so it may seem unfair to compare them, but they both do offer a way to develop front end web applications which is the desired goal for this part of the system. Cory House mentions [10] that because AngularJS is a framework it 'offers more opinions out of the box, which helps you get started more quickly without feeling intimidated by decisions' whereas React does not offer as many opinions about your code architecture. AngularJS seems to guide developers on the decisions they can make more than React. While this has both pros and cons, the AngularJS framework aligns better with this project.

AngularJS is also part of the Mongo, Express, AngularJS and Node (MEAN) stack, which has become a very common technology stack in recent years for full stack web development [11]. As Node.js and Express will be used for the backend web API, it integrates nicely with an AngularJS frontend. AngularJS is a more suitable technology for the web front end of the system for these reasons.

[11] https://www.npmjs.com/
[12] https://angularjs.org/
[13] https://facebook.github.io/react/

2.3.3 iOS vs Android (Swift vs Java)

Both iOS and Android offer rich SDKs for developing mobile applications. Being familiar with both iOS development and Android, the decision was based on experience. iOS will be used as the mobile application due to experience.

2.3.4 Numenta Platform for Intelligent Computing (NuPIC)

NuPIC is an open source project based on a theory of the neocortex called Hierarchical Temporal Memory [12]. Hierarchical Temporal Memory is a machine learning technology that aims to capture the structural and algorithmic properties of the neocortex [12]. NuPIC provides software for analysing streams of data which can learn time based patterns, detect anomalies and generate predictions. NuPIC is needed in this project for its prediction capabilities, which will enhance the proactive objectives of the system by predicting the number of orders expected for the current hour and the maximum number of employees needed per hour.

Hierarchical Temporal Memory (HTM)

As mentioned before HTM is a machine learning technology that aims to capture the structural and algorithmic properties of the neocortex, HTM works best with streams of data; from these streams of data NuPIC can recognize patterns. These streams of data must be time based as described by Numenta [13] 'time plays a crucial role in learning, inference, and prediction' as static data provides no context for NuPIC when inferencing, therefore it is needed to make predictions on data sets. The data used in this project for NuPIC will all be time based. NuPIC will be used in this project to predict the number of orders expected for the current hour and max number of employees needed for the current hour for a businesses.

Tutorial - One Hot Gym

NuPIC provides several tutorials online, one of them being the One Hot Gym tutorial [14]. The data used in this tutorial is very similar to the data that will be used in this project. The data is composed of hourly energy consumption records (kilowatt hour) for a gym over a certain period of time.

For NuPIC to generate the best prediction models a process called swarming must be executed. Swarming is a process that automatically determines the best prediction model for a given dataset [15] . Swarming will generate the best prediction models by trying multiple different models on a dataset and outputting the parameters for the one model that performed the best (generated the lowest error score) [15]. Once the data has been swarmed NuPIC is able to predict the energy usage one hour in advance for the gym. The data in this tutorial is almost identical to that of this project, however this project will not be measuring energy consumption, but order amounts and max number of employees.

2.3.5 Proactive Module - Flask

Although Flask was not chosen for the main backend web framework of the system in Section 2.3.1, its simplicity and ease to setup makes it a great candidate for the proactive module which only needs a very basic HTTP interface. The proactive module should be a standalone process that can take HTTP requests and can be initiated by a business to analyse orders for scheduling and task allocation as well as monitor the number of employees needed for time periods during operating hours. This will fulfill the proactive aspects of this project. The proactive module needs to be separated from the Node.js application as it will have its own complex code logic and architecture which will make it easier to test. It also will take complexity away from the Node.js application and allow the code base to be more modular.

2.3.6 Googles Map Distance Matrix API

When customers are ordering from a business, it will be their location and distance from the business that will be used to determine when they will arrive to collect their order. Google's Map Distance Matrix API[14] allows developers to calculate the arrival time. This API allows users to specify their mode of travel from walking, cycling and driving and have the arrival time calculated.

[14]https://developers.google.com/maps/documentation/distance-matrix/intro

2.4 Scheduling Algorithms and Real Time Systems

2.4.1 Overview

This section describes research into scheduling theory and real time systems, a description of the necessary notations typically used throughout scheduling literature, some of which is used throughout this report. Also included is an overview of the Earliest Deadline First scheduling algorithm.

2.4.2 Real time systems

As described by Stankovic et al. [3, p.1]:

'Real-time systems are defined as those systems in which the correctness of the system depends not only on the logical result of the computation, but also on the time at which the results are produced'

There are typically three different categories of real time systems, they include: hard real time system, firm real time systems and soft real time systems. In terms of scheduling within real time systems it involves the allocation of resources and time intervals to tasks in such a way that certain timeliness performance requirements are met [3].

Hard deadlines: Hard real time system have hard deadlines which mean missing a deadline would be disastrous, such as safety critical applications.

Firm deadlines: Firm real time system have firm deadlines which as described by Stankovic et al. [3, p.15] 'a task should complete by the deadline, or not execute at all. There is no value to completing the task after its deadline'.

Soft deadlines: Soft real time systems have soft deadlines which permits missing deadlines. Stankovic et al. [3, p.15] claim 'it is possible for soft real time system to have no deadlines and the only requirement is to have task completed as soon as possible'. An example of a soft real time system would be an video controller where missing deadlines may cause some degradation of video quality.

2.4.3 Notations

The notations used in this section are the common notations used to explain scheduling across the scheduling literature, they are also the notations that are used throughout this report, with more notations introduced in Section 4.4.2.

Task (t) In the scheduling framework a single unit of work is referred to as a task and can be denoted as t. The i^{th} task in the system is denoted by t_i, all tasks in the system can be represented by the set $\{t_1, t_2, t_3, ..t_n\}$ and can be denoted as n.

Worst case execution (C_i) The worst case execution/ processing time of task t_i.

Release (r_i) The release is the time at which the task t_i is ready for processing.

Deadline (d_i) The deadline of task t_i, the time the task is promised to be completed. This deadline can be hard, firm or soft.

Period (T_i) For the periodic task types described in Section 2.4.4 a period indicates that a task must execute once per period T. The most common case is when the deadline equals the period [3] ($d_i = T_i$), which means, for example if a periodic task must execute every 8 time points and the period is equal to the deadline, then the current execution of that task must be done by the next time period which would be at time point 16. A time point is some point in the schedule, it could be seconds or minutes etc, although it is typically not specified when explaining scheduling algorithms.

2.4.4 Task types

In the scheduling literature there are three types of real-time tasks: periodic, aperiodic, and sporadic. [3].

Periodic

Synchronous periodic tasks are a set of periodic tasks which are all released at the same time, usually considered time zero.

Asynchronous periodic tasks are a set of periodic tasks where they can be released at different times.

Aperiodic tasks are real-time tasks which need to be scheduled at some irregular and unknown rate.

Sporadic tasks are real-time tasks which need to be scheduled irregularly at a known rate.

2.4.5 Types of scheduling algorithms

Static Scheduling Algorithms For static scheduling, the scheduling algorithm has complete knowledge regarding the task set and its constraints, such as deadlines and computation times.

Dynamic Scheduling Algorithms For dynamic scheduling, the scheduling algorithm has complete knowledge of only the tasks that a currently active in the system, these tasks are scheduled without prior knowledge of tasks that may arrive in the future.

2.4.6 Earliest Deadline First Algorithm

Overview

This section explains the Earliest Deadline First Algorithm. The Earliest Deadline First (EDF) algorithm was introduced by Liu and Layland in 1973 [2]. This algorithm was researched for this project to gain an understanding of the types of solutions scheduling algorithms provide and how they could help with scheduling and prioritising orders based on their time constraints. This algorithm heavily influenced the method of scheduling as well as utilisation status generation used in this project in Section 4.4

Scheduling

In EDF tasks are scheduled according to their deadline, the task with the earliest deadline is the task that is scheduled first [2]. The utilisation for a task under EDF is $U_i = \frac{C_i}{T_i}$ [2], where C_i is the worst case processing time of a task and T_i is the task‘s period, which for EDF is assumed to be equal to the task‘s deadline as mentioned in Section 2.4.3.

The theorem [2] states that given a set of periodic tasks with deadlines, the task set can be scheduled by EDF if and only if $U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$. This theorem states that if

the utilisation needed for the sum of all tasks is less than or equal to 1 then EDF can schedule all tasks within a task set, otherwise EDF cannot.

t_i	C_i	d_i	T_i
t_1	1	2	2
t_2	1	5	5

Table 2.2: Task set

By applying this formula to the set of periodic tasks in Table 2.2, it can be known if they can be scheduled by EDF.

$U = \frac{1}{2} + \frac{1}{5} = 0.70 - 70\%$ as $U < 1$ this task set can be scheduled by EDF.

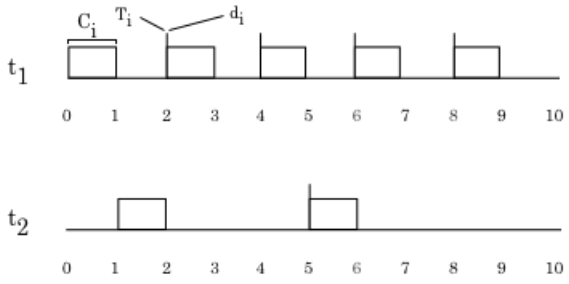

Figure 2.4: Periodic tasks scheduled using EDF

Figure 2.4 illustrates how these tasks would be scheduled using EDF. The numbers 0,1,2,3.. etc represent each time point in the schedule, and the vertical lines represent the deadline and period for the respective tasks. The tasks are periodically scheduled at their respective periods (T_i). For example t_1 has a period and deadline of 2, which means a new instance of task is t_1 is activated every 2 time points for processing and it must be complete processing before its next period. At each time point the task with the earliest deadline is scheduled. The pseudo-code for EDF would look something like the following:

```
for each time-point:
     for each available task:
          schedule the task with closest deadline
```

2.4.7 Discussion

It is worth mentioning this project will not be a real time system, as the tasks are to be process and completed by employees and not a computer system. Real time system's were explored in this section to gain an understanding of scheduling. However the term soft deadline may be used throughout this report specifically to conclude this

section in Section 2.4.8 and in Section 4.4 in which the term is used to describe that missing deadlines is permissible, but this is no longer in reference to a real time system.

2.4.8 Conclusion

Real time systems and scheduling algorithms specifically the Earliest Deadline First algorithm were explored and researched in this section to gain an understanding of the existing literature on scheduling and real time systems and what solutions they could provide for this project. While real time systems and EDF were researched, one thing that has become evident is the time metrics used by EDF and typical real time systems is not applicable to this project. As the time granularity for real time systems is typically considered in terms of CPU cycles or very small time quantum's, the time granularity needed for this project is not nearly as fine as the schedule of tasks is expected to be completed by humans and not computers. For this reason the deadlines in this project must be soft as guarantees about the exact finish times of tasks are opaque and cannot be guaranteed when regarding humans due to factors such as experience level, mood or motivation. As the consequences for missing deadlines in this project cannot be calculated i.e. missing a deadline actually may have no consequences for a business or conversely may mean loss in revenue for a business, the only assumption that can be made is that the task must be completed as soon as possible if it misses its deadline.

In regards to EDF, the research conducted was to investigate how its schedule was generated based on the time constraints from a set of tasks such as deadlines and processing times, as well as to learn the possibilities and limits of building these schedules based on these time constraints. While EDF schedules according to task deadlines, this is not sufficient for this project which is discussed in Section 4.4.7 as scheduling a food order (task) too early may spoil it, therefore the release time will be used to dictate the schedule of tasks. EDF inspires many of the formulas which are used in Section 4.4.2, such as generating a utilisation status for a business based on how busy their workers are

2.5 Interval Tree

2.5.1 Overview

The interval tree data structure is used extensively in this project for detecting conflicts between tasks. This section contains the research into understanding the interval tree data structure and how the search operation is useful for finding conflicts in a schedule. The term conflict is used throughout this paper and is defined below.

Definition 2.1 *Conflict A conflict is when two or more tasks processing times overlap in a schedule.*

The theory of finding conflicts is explained further in Section 4.4.3. In section 4.5.4 the implementation of the interval tree data is presented.

2.5.2 Data structure

Interval trees are used for querying time intervals to find out what events occur during a given interval [1], for example this querying is useful for finding different time intervals of a schedule produced by a scheduling algorithm to find out which tasks need to be processed within those time intervals and also detecting conflicts. Intervals are a pair of numbers that represent a start and end point in time. Cormen et al. [1, p.348] mentions that 'Intervals are convenient for representing events that each occupy a continuous period of time' which in this project represents the processing time of each task in a schedule. The interval tree can be based on a self balancing binary search tree, for example the interval tree described by Cormen et al. [1, p.349] uses a red-black tree as the underlying data structure to store the intervals, however the interval tree used in the implementation in Section 4.5.4 uses an AVL tree as the underlying data structure [16].

The interval tree supports a search operation which given an interval i finds if it overlaps with any other intervals. To explain this operation the following notations are used [1]:

Interval - An interval $[t_1, t_2]$ can be represented as an object i, where $i.low = t_1$ (starting point) and $i.high = t_2$ (end point).

Tree - The interval tree can be represented as T (T was represented in Section 2.4 as the period of a task, for all other sections throughout this report T will represent the period of a task except for this section)

Node - x is a node in the tree. Each node in the tree stores an interval i as $i.int$ as well as the maximum endpoint ($i.high$) of any interval stored in the sub tree rooted at x, and is denoted as $x.max$. The left child of x is denoted as $x.left$ and the right child is denoted as $x.right$

Overlap - Two intervals i_1 and i_2 overlap only if $i_1.low \leq i_2.high$ and $i_2.low \leq i_1.high$ [1]. A visualisation of an overlap can be seen in Figure 2.5

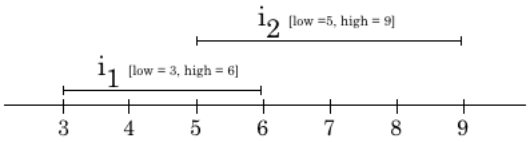

Figure 2.5: Overlapping intervals

The interval tree in Figure 2.7 is a representation of how intervals from Figure 2.6 are stored within an interval tree data stucture.

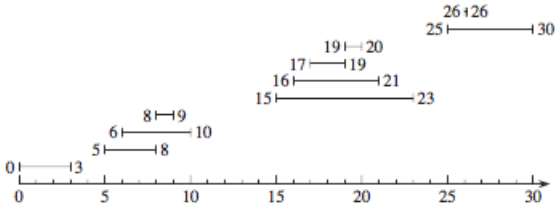

Figure 2.6: Interval tree data structure [1, p.350]

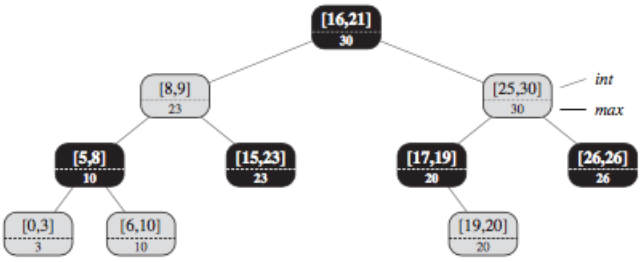

Figure 2.7: Interval tree data structure [1, p.350]

To query a point in the interval tree to see if it overlaps the following pseudo code described by Cormen et al.. [1, p.351] can be used.

```
IntervalTree-Search(T, i)
      x = T.root
      while x != T.nil and i does not overlap x.int
            if x.left != T.nil and x.left.max ≥ i.low
                  x = x.left
            else x = x.right
      return x
```

The parameters for the procedure are T the tree that contains the intervals and i the interval to find overlaps with.

2.5.3 Conclusion

In this section the interval tree data structure was researched and investigated, it allows for searching overlaps in a set of intervals. This will be needed for searching conflicts between tasks in a schedule produced by a scheduling algorithm in Section 4.4.3.

Chapter 3

Design & Architecture

3.1 Software Methodology

The Prototyping Model will used to develop software for this project. Dinesh Thakur mentions [17] 'this model assumes all the requirements for the software may not be known at the start of the development. It is usually used when a system does not exist or in case of a large and complex system where there is no manual process to determine the requirements'. Using this methodology, prototypes can be built, evaluated and expanded. The aim for this project is to develop a loosely coupled system of different modules. Using the Prototyping Model each prototype for each module can be developed with the most important requirements first and then evaluated. Once evaluated more requirements are developed into the prototype until the prototype is ready for production.

3.1.1 Requirements and analysis

In this stage the system requirements are gathered in detail. While requirements are gathered in this stage, this methodology assumes not all requirements will be known on the first iteration. So requirements stage may need to be revisited on following iterations.

3.1.2 Quick Design

Once the requirements have been gathered a quick design for the system is created. The quick design is composed of the important requirements.

3.1.3 Build Prototype

A prototype is built from the design and requirements. This prototype should be composed of all important requirements.

3.1.4 User Evaluation

Evaluate the prototype by through user interaction and get feedback and comments.

3.1.5 Refining Prototype

Once feedback and comments have been received from users, the prototype is refined.

3.1.6 Engineer Product

Once all the final prototype meets the requirements and is accepted, it is evaluated on a regular basis.

3.1.7 Conclusion

As the Prototyping Model emphasises quick prototyping development by concentrating on the most important requirements, the core functionality requirements of the system are the highest priority. This model is most suitable for this project as it offers a way to quickly build prototypes, evaluate and refine them iteratively. As all requirements for this project may not be known in the first iterations, each phase can be revisited and a new prototype can be developed with the new requirements.

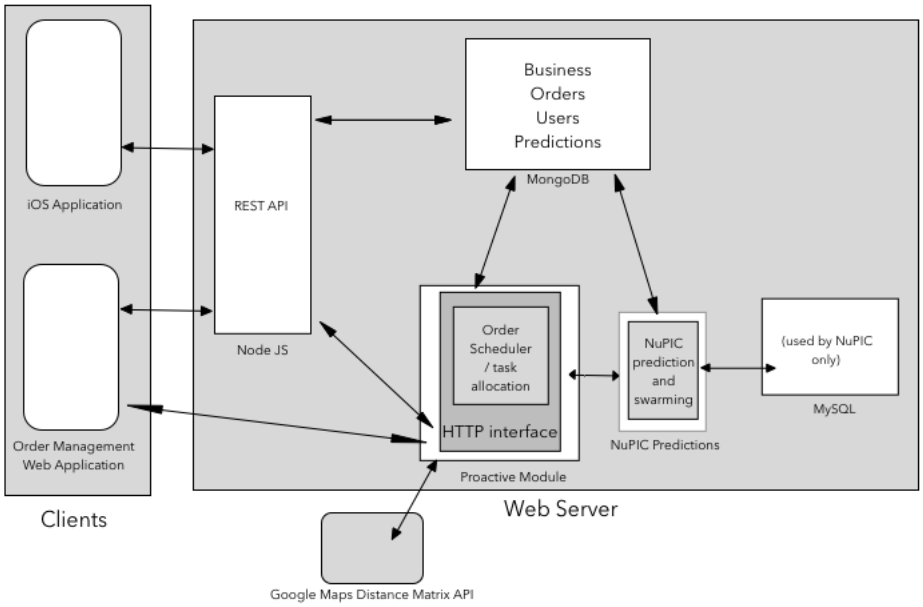

Figure 3.1: Overall system design

3.2 Overall Design & Architecture

The overall system design and architecture can bee seen in Figure 3.1. The system is designed to have a client server architecture, with the iOS application and web application as the clients and Node.js application, proactive module as server systems. NuPIC and the MongoDB instance can be accessed via the Node.js REST API or the proactive module. The arrows in Figure 3.1 represent the direction of communication between each of the components.

3.3 Use Cases

The initial use case for this project can be seen below in Figure 3.2. However, Figure 3.3, more accurately represents the use cases of the final system. In Figure 3.3 the use case has been expanded to include some extra features, all use cases for a customer and business are explained in the lists below.

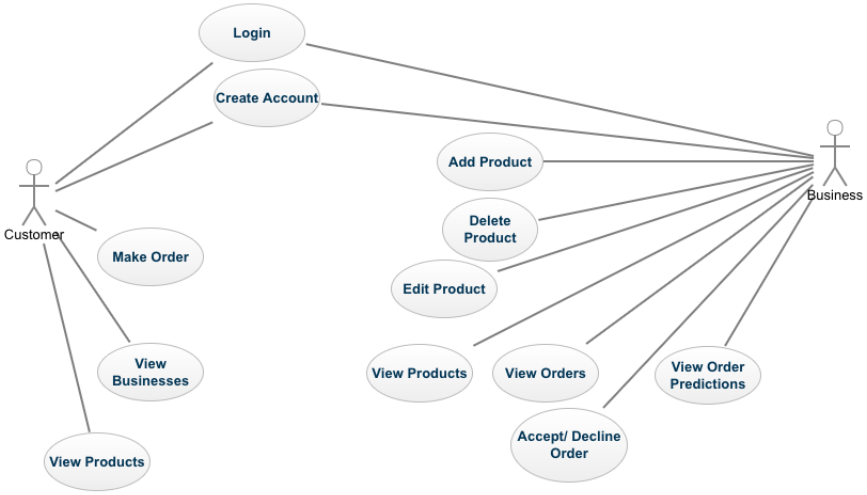

Figure 3.2: Initial use case of the system

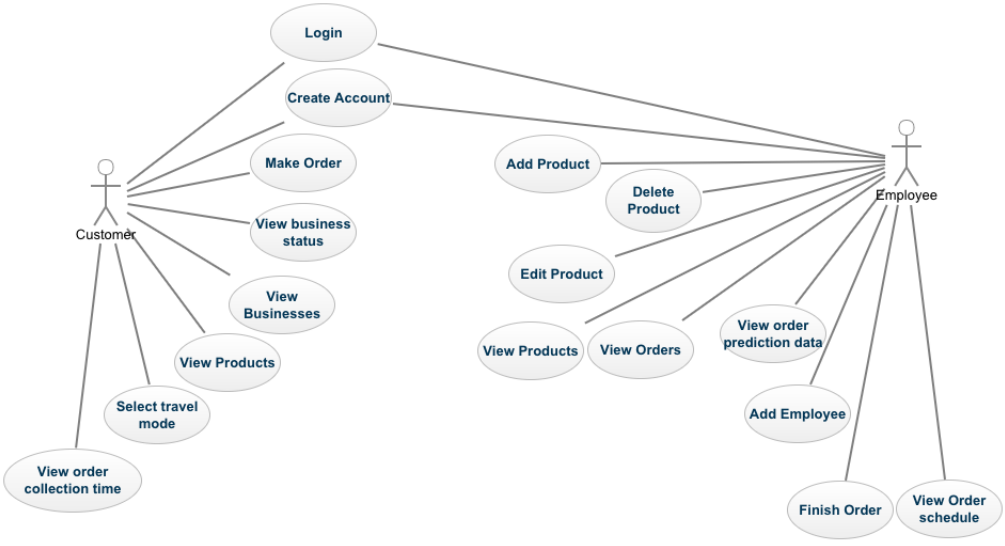

Figure 3.3: Final use case of the system

User

- Create account - Create an account for user
- Login - User login or business login
- Make order - Make order from a business
- View business status - View the business status, if the business is busy or not.
- View businesses - View all businesses in the system
- View products - User: View products available to order from a business.
- Select travel mode - When making an order select the mode of travel, so their arrival time can be calculated and the order scheduled
- View order collection time - View the time the order can be collected at.

Business

- Create account - Create an account for a business
- Login account - Login in a business
- Add product - Add a product to a business
- Edit product - Edit a product e.g. change price or description
- Delete product - Delete a product from a business
- View products - Each business can view all their products
- View orders - View current orders in the system
- View orders schedule - View the schedule of the orders, so the order in which each order in the system should be serviced.
- Finish Order - Once an order has been served, the it can be removed from the order schedule and current orders list
- Add employee - Add an employee to the system, so they can be assigned orders.
- View order predictions - View the predictions for the expected orders for the current hour and the max number of employees, as well as being able to see this data in a chart

3.4 Database Design

MongoDB (version 3.2.10) is used to persistently store information for the system. The database consists of the following collections of documents, users (customers), businesses, orders and predictions. Both the Node.js application and the proactive module will read and write from the database. In Table 3.1 there is a description of each document that will be used to make up each collection in the MongoDB database. In MongoDB, a document is composed of field-and-value pairs and have the following structure they are similar to rows in a traditional relation database management systems (RDBMS) [18]. Collections are composed of many documents and are analogous to tables in RDBMS [18].

Document	Field Descriptions
Business document	_id: of a business email: of a business password: for a business address: of a business contact number: for a business period: the hours of business coordinates: The coordinates of a business, used during ordering, to compare with customer location.
User/ Customer document	_id: The id of a user email: User email firstname: User firstname lastname: User lastname password: User password
Order document	_id: Order id businessID: The id of the business the order belongs to userID: The id of the user that made it processing: The time it takes to process the order status: The status of the order - processing or finished cost: The cost of the order travel Mode: The mode of travel the customer is using to get to a business release: The release time of the order (the an employee should begin servicing the order) deadline: The time the order should be done by finish: The time the order was actually finished or ready coordinates: The coordinates of the customer when they made the order products: The products of the order.
Product document	_id: Product id businessID: The id of the business the product belongs to name: business name description: Description of the product processing: The amount of time the product takes to service
Prediction document	_id: The id of the prediction data data: The data generated prediction data swarmType: The type of swarm that was run e.g swarm type for prediction max employees per hour or orders per hour businessID: id of the business for which the prediction data belongs to

Table 3.1: MongoDB design

Chapter 4

Implementation

4.1 Overview

This section describes the implementation of each component of the system in detail and solutions provided. This section builds on the research and design from the previous sections.

4.2 iOS mobile application - Customer Application

4.2.1 Overview

The iOS mobile application is used by customers to order from any business of their choosing. It is developed using Apple's programming language Swift[1], version 3.0. The mobile application has a Model-View-Controller architecture. The application communicates with the Node.js application via HTTP where its content is loaded. The iOS application allows customers to create accounts, view businesses, view products and create orders. The application is only used by customers.

4.2.2 Business Page

The business page allows customers to view information about a business, such as utilisation status (discussed further in Section and Section) and view products.

[1] https://swift.org/

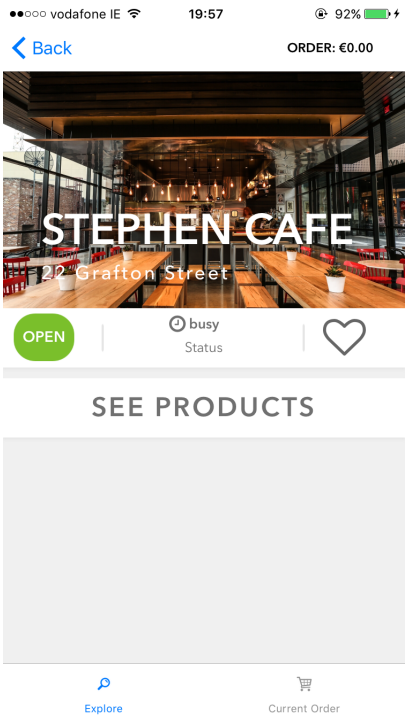

Figure 4.1: Business page mobile application

4.2.3 Products Page

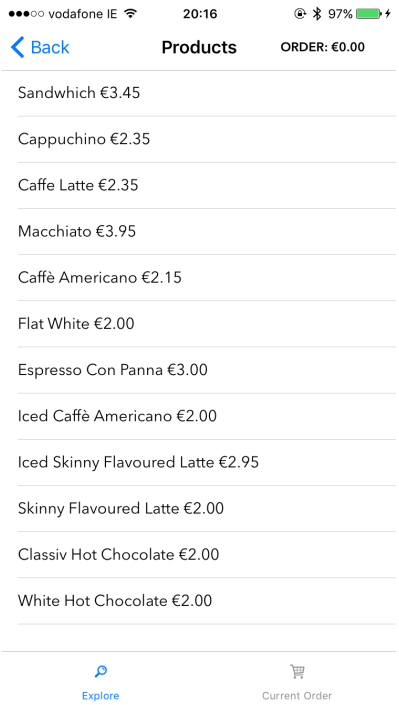

Figure 4.2: Products page

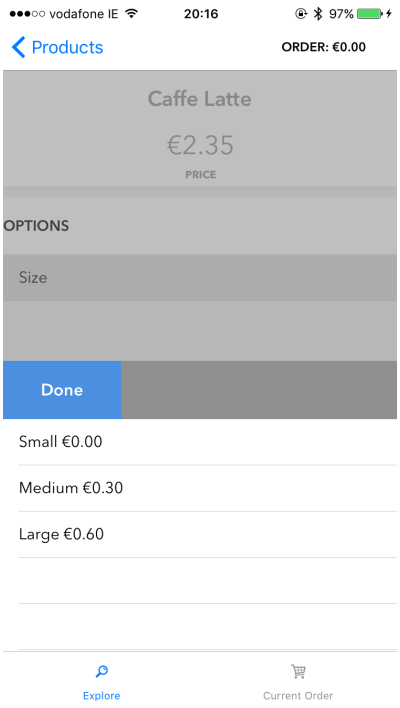

Figure 4.3: Selection options for a product

A customer can view the products by clicking the “See Products” button on the business page. This will navigate a customer to the products page of a business. The

products page displays all of the products that have been added through the web application by a business. Customers can select each product from this page and add it to their order. For each product that has options associated with it, they can also be selected. For example in Figure 4.3 the options “Small”, “Medium”, “Large” can be selected for the product. Once a customer has added all their products to the order. Once all the products have been added to the order by a customer they can be ordered. Figure 5.3a shows the ordering page.

4.2.4 Orders Page

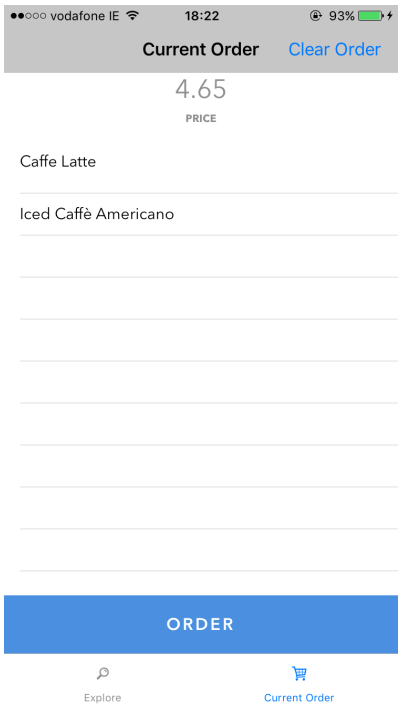

Figure 4.4: Current Order

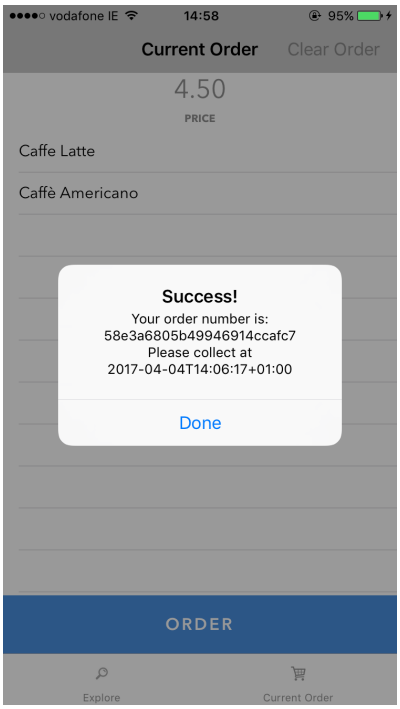

Figure 4.5: Order success

Before a customer can order they must select their travel method. The travel method is used in calculating the customer arrival time through Google Maps Distance Matrix. Once the order has been received by the proactive module it is scheduled and the time the customer should collect the order is displayed in the iOS application.

4.3 Node.js Application (REST API)

4.3.1 Overview

The Node.js application exposes a Representational State Transfer (REST) Application Programming Interface (API) that can be used by clients from the iOS application to the web application to interact with the database, proactive module and NuPIC predictions.

4.3.2 Implementation

The implementation of the Node.js application began with development of the REST API. Express[2], which is a REST framework for Node.js was used to build the REST API. Express allows for the development of endpoints for the REST API, which are URIs that can accept requests, specifically HTTP requests. There are three main layers to the Node.js application, they are the routing layer, business logic layer and database layers, these can be seen in Figure 4.6 where the arrows represent the direction of communication between the layers.

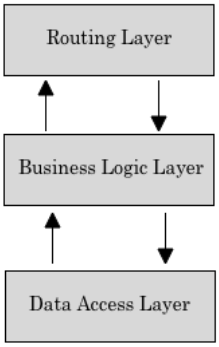

Figure 4.6: Node.js layers

Routing Layer

The routing layer consists of endpoints, which are URLs (in this case) that can handle requests. Each endpoint provides a way for clients to interact with resources available by the system. Each request is parsed and routed to the correct resource. The routing layer forwards requests to the business logic layer where it can further handle

[2]https://expressjs.com/

the request. Listing 4.1 shows how some of the common endpoints for products that are available through the routing layer.

```
// Endpoint: product/ (HTTP Request: POST) Adds a new product
router.post('/', vr.validRequest, function (req, res) ...

// Endpoint: product/ (HTTP Request: GET) Gets a product
router.get('/', vr.validRequest, function (req, res) ...

// Endpoint: product/ (HTTP Request: PATCH) Updates a product
router.patch('/', vr.validRequest, function (req, res) ...

// Endpoint: product/ (HTTP Request: DELETE) Deletes a product
router.delete('/:productID', vr.validRequest, function (req, res) ...
```

Listing 4.1: Product endpoints in the routing layer

Business Logic Layer

The business logic layer provides the core business logic for the system. This layer parses and reads the requests which have been forwarded from the routing layer and decides what actions should be taken based on the request. The business logic layer is the middle layer that can talk to both the data access layer and the routing layer. Most of the business logic layer code is encapsulated in JavaScript functions, which are analogous to classes from the standard object oriented approach. For example the Order function holds all business logic needed for ordering, such as making a new order and finishing the order. Listing 4.2 shows the function definitions for some of the operations that are available in the business logic layer for Orders.

```
// Finishes an order
Order.prototype.finish = function finishOrder(id, cb) ...

// Gets an order from the database
Order.prototype.get = function get(client, cb) ...
```

Listing 4.2: Order function in business logic layer

Data Access Layer

The data access layer provides an API to interact with the persistent storage where business, user, order, prediction data can be accessed from the MongoDB instance.

Through the data access layer, records can be created, read, updated and deleted. Mongoose[3] - an object document mapper for MongoDB is used in this layer. Mongoose is analogous to an object-relational mapper with traditional relational database management systems, it provides object translation from MongoDB to the Node.js application. The documents for MongoDB e.g. order document, product document etc from Table 3.1 are specified in the Node.js application, through Mongoose, the *User* document is defined in Listing 4.3 through Mongoose.

```
var User = new Schema({
    id: ObjectId,
    firstname: String,
    lastname: String,
    email: String,
    password: String,
    createdAt: { type: Date, default: Date.now },
});
```

Listing 4.3: User schema in the data access layer

4.4 Scheduling Algorithm

4.4.1 Overview

This section explains the order scheduling algorithm, conflict detection, calculating workers needed, utilisation status generation and task allocation. It should be noted that the term worker and employee used in this section and for the rest of this report are synonymous as well as the term task and order. The scheduling algorithm that is described in this section to schedule tasks took inspiration from the Earliest Deadline First algorithm explained in the Section 2.4.6, however, instead of using the deadline time to schedule tasks, it is the release time that is used. For this project the release time is the time a worker must begin processing a task. By using the release time to schedule the orders, the aim is to have the tasks finish at their deadline or as close to the deadline as possible, to ensure the task is as "fresh" as possible when a customer arrives to a business. Freshness of an order is important as this project will be scheduling food items. The deadlines in this algorithm are also soft, therefore deadline misses are tolerable as discussed in Section 2.4.7 soft deadlines in this section do not refer to real time systems.

[3]http://mongoosejs.com/

The scheduling algorithm described in this section is a dynamic scheduling algorithm as described in Section 2.4.5 means the algorithm only has knowledge of the tasks that are currently in the system. The algorithm schedules aperiodic tasks, which as mentioned in Section 2.4.4 means tasks need to be scheduled at some irregular and unknown rate. This is the case as customers can make orders at any time and will need to be scheduled. Through researching and developing this algorithm, it was discovered that the number of workers needed for processing all tasks in the system could be known as well as generating a utilisation status for a business based on the current tasks in the system. The utilisation status took inspiration from the utilisation formula used by EDF $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$ in Section 2.4.6.

Throughout the next sections there is a heavy focus on the number of tasks in the system that conflict at a given time and the maximum amount of tasks that a worker can service simultaneously which is referred to as the multitask value. As per definition 2.1 a conflict can be described as two or more tasks whose processing time overlap at some point in the schedule. In Figure 4.7 tasks t_1 and t_2 conflict and in Figure 4.8 the time at which the two tasks conflict is inside the two red vertical lines.

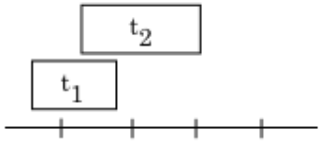

Figure 4.7: Two conflicting tasks

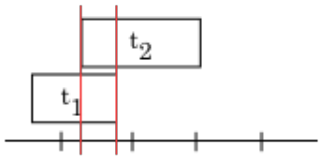

Figure 4.8: Area of conflict highlighted

4.4.2 Scheduling

To explain the scheduling algorithm, much of the notations that are used to explain scheduling theory in the research section are used again. However, there are some new notations which are outlined below. (One thing to note in this section the period which was denoted as T_i in Section 2.4.3 and was used in EDF is no longer used for this algorithm as once a task is scheduled it will never need to be scheduled again at any period).

- r_i - release time (calculated by deadline (d_i) - processing (C_i)) of task t_i
- k - highest number of conflicts of a set of tasks
- w - the number of workers available in the system. As already mentioned the term employee and worker are synonymous
- a - the number of tasks assigned to a worker
- m - the multitask value for each worker, the highest number of task a worker can process simultaneously
- $M = w \times m$ - the maximum number of tasks that can be completed simultaneously by all workers available in the system
- $W = \left\lceil \frac{k}{M} \right\rceil$ - the maximum number of workers needed to complete a set of tasks that conflict.

To schedule tasks with this algorithm, consider the following task set in Table 4.1, along with the deadline, processing and release time for each.

Task	d_i	C_i	$r_i = d_i - C_i$
t_1	9.30	2.00	9.28
t_2	9.35	6.00	9.29
t_3	9.45	4.00	9.41
t_4	9.47	3.00	9.44
t_5	9.47	4.00	9.43
t_6	9.49	5.00	9.44
t_7	10.00	3.00	9.57
t_8	10.05	4.00	10.01
t_9	10.08	4.00	10.04
t_{10}	10.14	4.00	10.10
t_{11}	10.15	8.00	10.07

Table 4.1: Sample tasks

In this algorithm, each task is scheduled according to its release time, the task with the earliest release is always scheduled first, therefore the schedule of the task set for this algorithm will always order tasks $\{r_1 < r_2 < r_3 < ...r_n\}$. This is visualised in Figure 4.9 using the tasks from Table 4.1. The horizontal black line indicates time.

The pseudo code for scheduling by release time would be:

```
for each time-point:
     for each available task:
          schedule the task with earliest release time
```

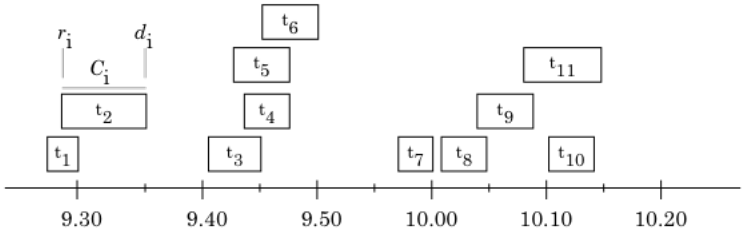

Figure 4.9: Schedule produced using release time

4.4.3 Conflict detection

There are three subsets of tasks that conflict from the task set in Table 4.1. A conflict as defined in Definition 2.1 is two or more tasks whose processing time overlaps, the reader is referred back to Figure 4.7 for a visualisation of a conflict. A conflict set is the longest amount of time that a set of tasks continuously conflict. To find a conflict set, first find a task that conflicts with another, and keep following any subsequent conflicting tasks until there are no more conflicts or tasks. The interval tree data structure which was discussed in Section 2.5 is used to detect conflicts, it is implemented in 4.5.3 to query a set of points in the schedule to find conflicts.

The highest number of conflicts between tasks is denoted as k, also any time in the schedule where a task exists that does not conflict $k = 1$ always. The sets of conflicts in the task set presented in Table 4.1 are:

- Subset 1: $\{t_1, t_2\}$, with $k = 2$
- Subset 2: $\{t_3, t_4, t_5, t_6\}$, with $k = 4$
- Subset 3: $\{t_8, t_9, t_{10}, t_{11}\}$, with $k = 2$. For Subset 3 the highest number of conflicts is still only 2, as from t_8 to t_{11} there a never more than two tasks that conflict at any one time.

These conflicts are shown in Figure 4.10, the areas between the vertical red lines are times that tasks conflict.

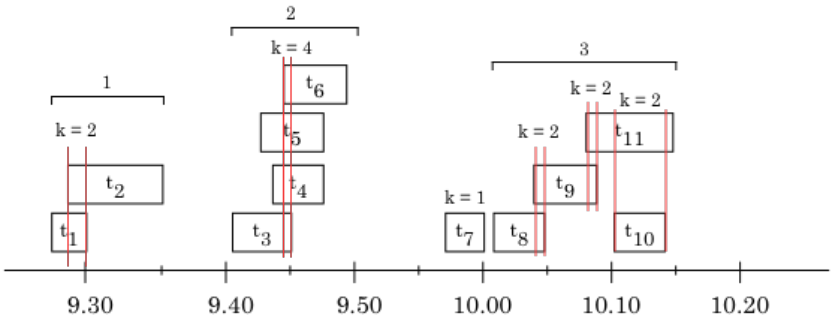

Figure 4.10: Schedule with conflicts highlighted

Now assume another task was added into the system t_{12}, with $d_{12} = 10.15$ and $C_{12} = 5.00$, therefore $r_{12} = 10.10$. Figure 4.11 visualises the schedule with t_{12} added.

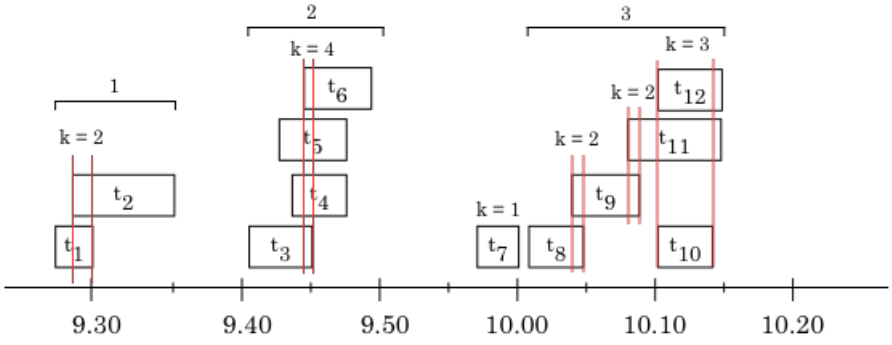

Figure 4.11: Schedule with t_{12} added

Subset 3 becomes: $\{t_8, t_9, t_{10}, t_{11}, t_{12}\}$, with $k = 3$. The highest number of conflict in Subset 3 is now 3.

4.4.4 Calculating workers needed

To calculate the number of workers needed to process these tasks, a multitask value is first needed, as described before, a multitask value is the highest number of tasks that each worker can process simultaneously and is the same for each worker, it is denoted as m, in this case assume $m = 2$ (The maximum amount of tasks that can be completed simultaneously by a set of workers is denoted as M). That means each worker can process up two tasks simultaneously. To calculate the maximum number of workers needed for a set of tasks that conflict, the following formula can be used: $W = \left\lceil \frac{k}{M} \right\rceil$ For the task set presented in Table 4.1 assume there are two workers

available to process tasks, therefore $w = 2$. As $M = w \times m$, $M = 4$ in this case. To calculate the number of workers needed we can apply this formula to the conflicting subsets of tasks, which is shown in Table 4.2 (The tasks from Table 4.1 are used here again.)

Subset	Tasks	k	m	$W = \left\lceil \frac{k}{M} \right\rceil$ (Workers needed)
1	t_1, t_2	2	2	1
2	t_3, t_4, t_5, t_6	4	2	2
N/A	t_7	1	2	1
3	t_8, t_9, t_{10}, t_{11}	2	2	1

Table 4.2: Workers needed calculations

There are two ways of knowing if a worker set cannot process every task in a set they are:

- if $k > M$ then there are more tasks which must be simultaneously processed, than which can be processed by all workers
- if $W > w$ then there are more workers needed for task processing than there are currently available in the system.

Assume a task t_{12}, with $d_{12} = 10.15$ and $C_{12} = 5.00$, therefore $r_{12} = 10.00$ is added into the schedule, it now conflicts in Subset 3, with $k = 3$ for Subset 3, as shown in Figure 4.11. If the formula $W = \left\lceil \frac{k}{M} \right\rceil$ is used on Subset 3 now that t_{12} has been added $\{t_8, t_9, t_{10}, t_{11}, t_{12}\}$, the schedule can no longer be completed with $w = 2$ as $W = 3$. As $W > w$, the number of workers needed is greater than the number available, therefore a task deadline may be missed. If the number of workers needed is more than are available, then it us up to the business to decide how the conflict should be resolved, i.e by swapping somebody in to handle the extra tasks or by allowing the possibility of a task deadline to be missed, as mentioned in Section 2.4.8 missing a deadline may have little to no consequences for a business based on how late the task is actually finished.

4.4.5 Utilisation status

The utilisation status is used to calculate how busy a business will be based on the tasks in the system and the workers available to process these tasks during a time period. The status can be calculated based on the utilisation of the workers. In this

project there are three different statuses, they are “ok”, “busy” and “very busy”. Each status is calculated based on measuring the employees needed and the available employees. Table 4.3 shows how each status is calculated.

Status	Formula	Description
ok	$W < w$	The number of workers needed is less than the workers available, meaning not all workers will be utilized.
busy	$W = w$	The number of workers needed is the same as the number of workers available, meaning each worker will be utilized.
very busy	$W > w$	The number of workers needed is more than are available. Utilisation is higher than can be achieved by the set of workers.

Table 4.3: Calculations for utilisation statuses

4.4.6 Task allocation

Task allocation is process of assigning workers tasks from a schedule that was produced by the scheduling algorithm. Each worker in the system has a shift, which is the start time and end time they are available to be assigned tasks. Each worker can be assigned up to m tasks, the number of tasks assigned to worker is denoted as a. A worker can only be assigned a task if:

- $a < m$
- if $a = m$ and the task to be assigned has an earlier release time than one of the tasks already assigned to the worker
- deadline of the task to be assigned is not later than the end of the workers shift
- release time is after the the start time of the workers shift

4.4.7 Conclusion

The scheduling algorithm, conflict detection, calculating workers needed, utilisation status generation and task allocation described in this section were created to reach the objectives set out in Section 1.2.5. The scheduling algorithm can advise workers in a business how orders should be processed based on time constraints such as deadline and release times. As the deadlines for this algorithm are considered soft some tasks may miss there deadlines i.e a worker failed to process a task fully before its deadline.

There is currently no mechanism for the algorithm to suggest what should be done for a business if a worker processing a task misses its deadline, the algorithm leaves that decision to the business. However, the components discussed such as calculating workers needed, conflict detection, utilisation status should alleviate some of these issues by suggesting how many workers would be needed during different parts of the schedule to help reduce deadline misses. These components were researched and developed to allow a business to foresee the different parts of the schedule were task deadlines might be missed. For example the utilisation status tells the business how busy they will be at certain periods during a day, which should be an indicator for them to prepare for a busy period so to minimise deadline misses. Overall the different components discussed in this section, from the scheduling algorithm to the task allocation are to help a business deal with orders that are made by customers in a proactive way as per Definition 1.1

4.5 Proactive Module

4.5.1 Overview

This section discusses the implementation of the theory developed in Section 4.4. The proactive module implements scheduling, conflict detection, calculation of workers needed, utilisation status generation and task allocation.

4.5.2 Architectural overview

The proactive module is implemented using Python 2.7. The requirements for the proactive module are the following:

- expose a HTTP interface. This requirement is needed as the task schedule needs to be accessed by other components of the system
- implement task scheduling algorithm
- calculate workers needed
- generate utilisation status
- task allocation

The source code layout for this component of the system is located in the quick-proactive/ proactive Python module and can be seen in Figure 4.12.

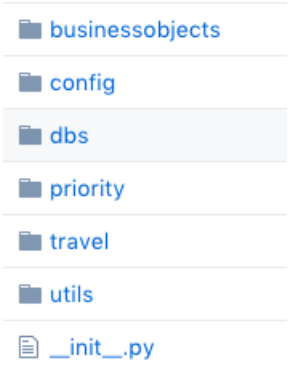

Figure 4.12: Code Structure of proactive module

The class structure for this code can be seen in the class diagram in Figure 4.13.

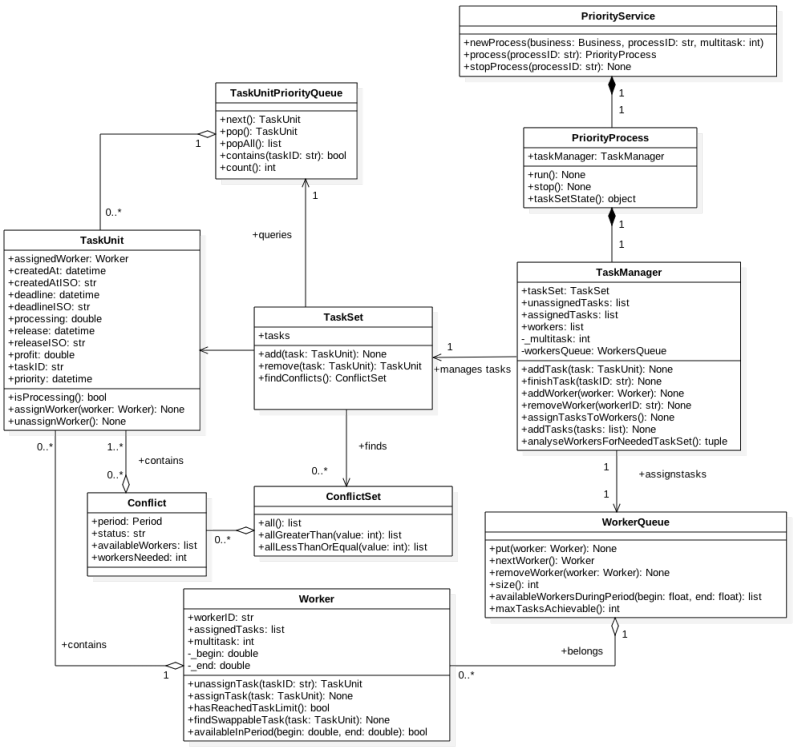

Figure 4.13: Class diagram for proactive module

Each business that is logged into web application can interact with this module through the control panel interface in Figure 4.14. From the control panel on the orders page a business can run a process within the proactive module by clicking the “begin” button which will monitor unprocessed orders, schedule orders, allocate orders to workers, generate a utilisation status and calculate workers needed. Before the “begin” button is clicked a multitask value needs to be entered. As explained in

Section 4.4.4 the multitask value is needed for scheduling and task allocation, as it states the number of tasks that can be assigned to a worker which need simultaneous processing.

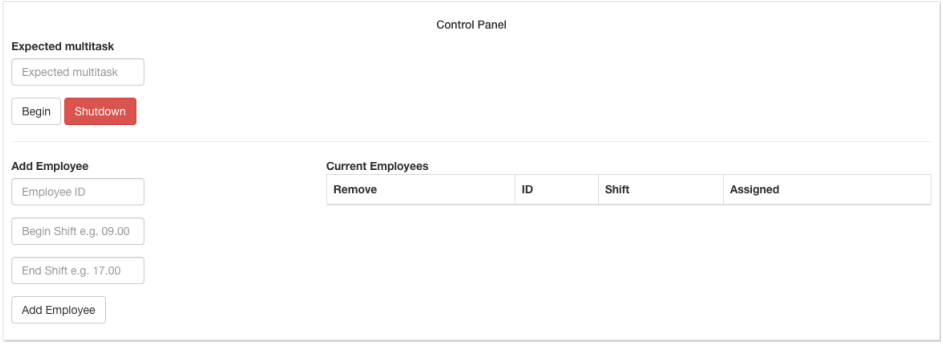

Figure 4.14: Control Panel for business

The proactive module runs on a Flask HTTP server. In this system the clients (businesses) can send the HTTP requests to the proactive module from the front end web application. Table 4.4 lists the HTTP requests the proactive module can accept through Flask.

HTTP Method	URL	Description
POST	localhost:6566/beginservice	begins a service for a business
POST	localhost:6566/addworkers	adds workers for a business
POST	localhost:6566/removetask	removes a task from the task manager for a business
GET	localhost:6566/tasks?id=businesID	retrieves all tasks for a business
GET	localhost:6566/stopservice?id=businessID	stops a service for a business
GET	localhost:6566/workers?id=businessID	retrieves workers for a business

Table 4.4: HTTP interface for proactive module

Businesses can message the *PriorityService* instance via a HTTP request. This *PriorityService* instance only ever exists as a singleton within the proactive module at runtime and is responsible for spawning, managing and forwarding messages to instances of the *PriorityProcess* class. Once a client clicks the begin button from Figure 4.14 a HTTP message is sent to the *PriorityService* class via Flask which will spawn a new *PriorityProcess* instance. The *PriorityProcess* instance is managed by the *PriorityService* singleton instance and runs in its own thread. The sequence diagram in Figure 4.15 shows the sequence of actions that will occur when a business clicks the begin button from the front end web application, up to the creation of the *Priori-*

tyProcess instance.

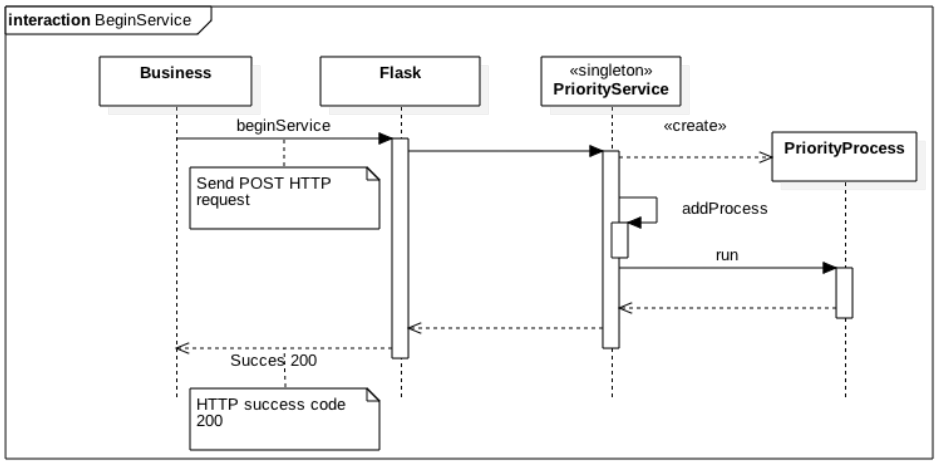

Figure 4.15: Sequence Diagram for starting priority process

PriorityProcess class

Each business that is logged into the system has their own *PriorityProcess* instance within the proactive module. This *PriorityProcess* instance is responsible for monitoring the following:

- retrieving unprocessed orders in the database for each business,
- calculating the customer arrival time for each order (this time becomes the deadline for each order)
- assigning tasks to workers, through its *TaskManager* instance.

The *__monitor* method on the *PriorityProcess* class is called every few seconds to carry out these tasks.

```
1 def __monitor(self):
2     """
3       Monitors the unprocessed orders, calcualates customer arrival time
4       and messages \textit{TaskManager} instance to assign workers to
      the tasks.
5     """
6     orders = self.__readUnprocessedOrders(self.__orderStore)
7     for order in orders:
8       orderObj = Order(
9         orderID=str(order["id"]),
```

```
10          status=order["status"],
11          processing=order["processing"],
12          customerCoordinates=order["coordinates"],
13          travelMode=order["travelMode"],
14          createdAt=order["createdAt"],
15          cost=order["cost"],
16          products=order["products"]
17        )
18        self.__orderStore.append(orderObj) # add to internal storage.
19        deadline = self._customerArrivalTime(orderObj.customerCoordinates,
      orderObj.travelMode)
20        task = TaskUnit(
21          createdAt=orderObj.createdAt,
22          deadline=deadline,
23          profit=orderObj.cost,
24          processing=orderObj.processing,
25          taskID=orderObj.orderID,
26          data=orderObj
27        )
28        self._taskManager.addTask(task)
29      self._taskManager.assignTasksToWorkers()
```

Listing 4.4: monitor method

TaskManager class

Each *PriorityProcess* instance has a *TaskManager* instance, this *TaskManager* instance is responsible for scheduling orders and task allocation to workers. The *TaskManager* instance implements all the functionalities of the scheduling algorithm described in the Section 4.4. The public methods exposed by the *TaskManager* class can be seen in Figure 4.16.

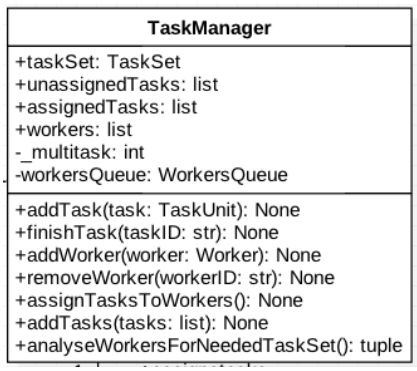

Figure 4.16: Public methods for *TaskManager*

Worker class

Employees are represented by the *Worker* class. When an employee is added from the front end web application it is the *TaskManager* instance that will assign the worker tasks. The *Worker* class in Figure 4.17 list the methods and properties of the worker class.

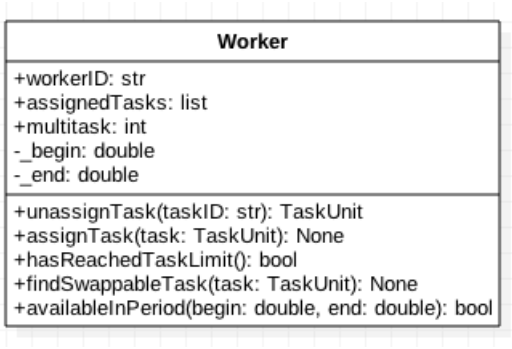

Figure 4.17: *Worker* class

In Figure 4.18 an employee is added from the web application to the proactive module. The employee is added with an id, shift start time, shift end time. The sequence diagram in Figure 4.19 shows the control flow when a business adds an employee from the web application.

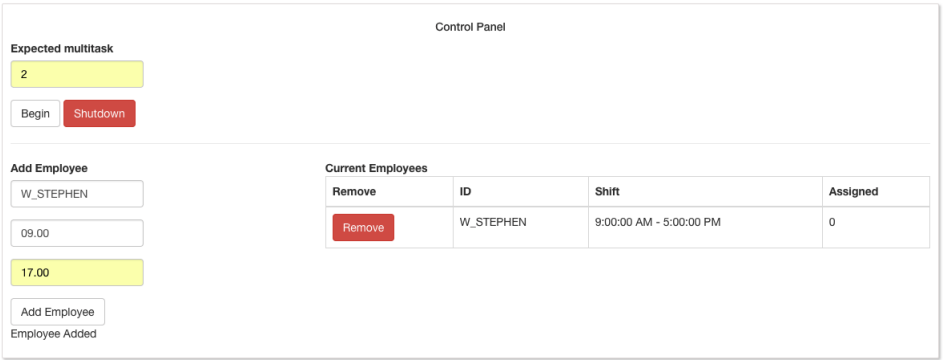

Figure 4.18: Adding employees from web application

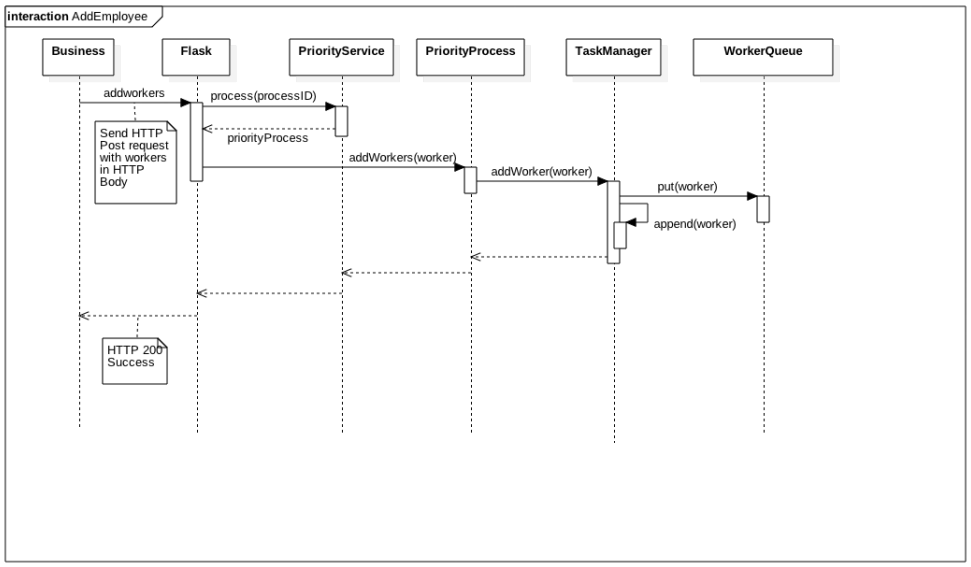

Figure 4.19: Add employees sequence

Tasks

Tasks are represented by the *TaskUnit* class. Tasks represent orders that are made by customers. The *TaskUnit* class contains timing information from release time and processing time as described in Section 2.4, which make it possible for the *TaskManager* to implement the scheduling algorithm.

4.5.3 Implementation of Task Scheduling Algorithm

This section describes the implementation of the scheduling algorithm described in Section 4.4.2. The *TaskManager* class is responsible for scheduling tasks. The *PriorityProcess* class is responsible for fetching all the orders from the database, calculating the customer's arrival time and invoking the *TaskManager* to schedule tasks and then assign these tasks to workers, see code in Listing 5.3. The *PriorityProcess* checks the database every few seconds to find any new unprocessed orders, the flowchart in Figure 4.20 shows the conditions in which new tasks are given to the *TaskManager* from the *__monitor* method in *PriorityProcess*. The *__monitor* method is invoked using the APScheduler[4] Python package every few seconds. The APScheduler package is a timer that allows for invocation of methods periodically.

The orders in the database have been created from the iOS application. The cus-

[4]https://apscheduler.readthedocs.io/en/latest/

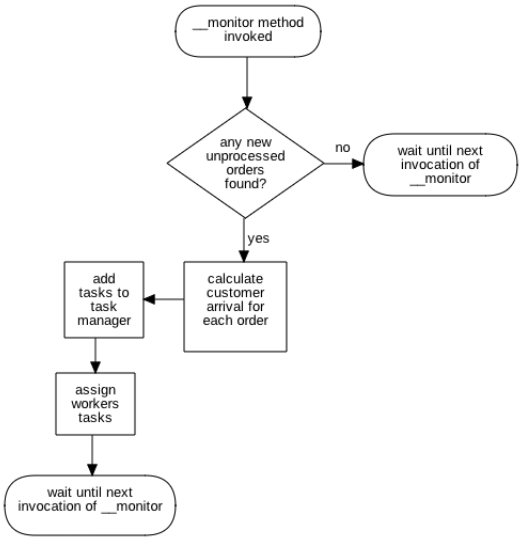

Figure 4.20: Flowchart when fetching new orders

tomer's coordinates are attached to each order that is made. Once the *PriorityProcess* fetches the unprocessed orders from the datavase the coordinates are then passed to the Google Distance Matrix API where the arrival time of the customer is calculated by comparing the location of the business to the location of the customer, this forms the deadline time of the order. Once this data has been accumulated, orders are then encapsulated in the *TaskUnit* class and given to the *TaskManager* class for scheduling.

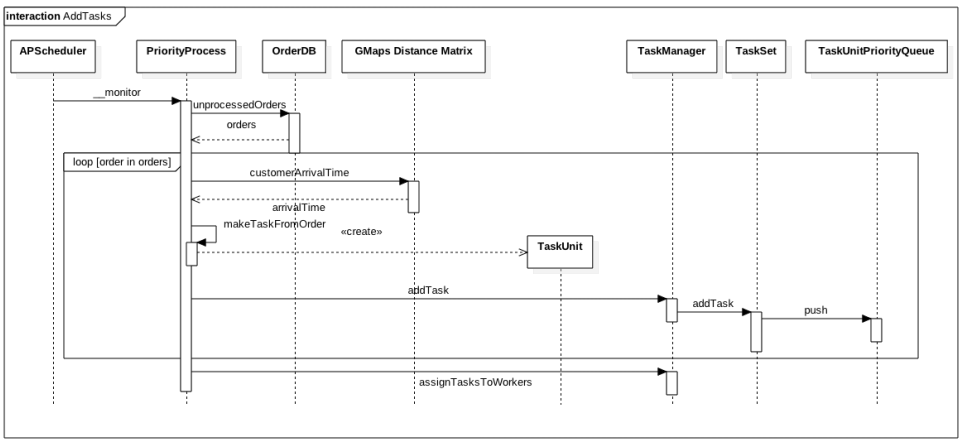

Figure 4.21: Seqeunce diagram to add new tasks

Once the tasks are given to the *TaskManager*, they are scheduled according to the scheduling algorithm described in Section 4.4.2, which means tasks are scheduled according to their release time. The release time for each Task is calculated inside the *TaskUnit* class during instantiation. As described in Section 4.4.2 the release time is calculated by the following formula: release = deadline - processing.

```
def releaseAt(deadline, processing):
  """
  The release time is calculate in the following way:
  r = deadline - processing
  where:
    deadline:   is the time the order should be processed by.
    processing: the time it will take to process the order.
  @param deadline:(datetime.datetime) The deadline.
  @param processing:(int) The amount of time in seconds the item will
    take to process.
  """
  if isinstance(deadline, datetime):
    return deadline - timedelta(seconds=processing)
  else:
    raise TypeError("Deadline must be of type <type 'datetime'>, not %s"
     % type(deadline))
```

Listing 4.5: releaseAt function

The *TaskManager* class schedules each task by this release time property. The *TaskSet* class holds the task schedule for a *TaskManager* instance. The *TaskSet* schedules each order using the *TaskUnitPriorityQueue* class. The *TaskUnitPriorityQueue* class keeps each tasks within a priority queue according to its release time. The priority queue that was used was the heapq[5] package. This packages uses the heap queue algorithm, also known as the priority queue algorithm to store data [19]. A priority queue is different from a "normal" queue, because instead of being a "first-in-first-out" data structure, values come out in order of their priority [1, p.162]. For this implementation tasks need to come out of the priority queue according to their release time. The earlier (lower) the release time the higher the priority. Listing 4.6 shows the instantiation of a the *TaskUnitPriorityQueue* in the *TaskSet* class. The *TaskUnitPriorityQueue* uses the heapq package internally.

```
class TaskSet(object):
  """
    Holds a set of tasks in a priority queue and interval tree.
  """
  def __init__(self):
    self._tasksQueue = TaskUnitPriorityQueue() # keep r1 < r2 < r3 order
    self._intervalTree = IntervalTree()
```

Listing 4.6: TaskUnitPriorityQueue construction

[5]https://docs.python.org/2/library/heapq.html

A priority queue uses a heap data structure to order each node [1, p.163], the heap data structure in Figure 4.22 is a representation of the tasks from Table 4.1 and how they would be stored using a priority queue inside the heapq package. The priority queue allows the *TaskManager* to schedule each task according to release $\{r_1 < r_2 < r_3, ... < r_n\}$ as set out by the scheduling algorithm in Section 4.4.2 and assigned task to workers in the correct order.

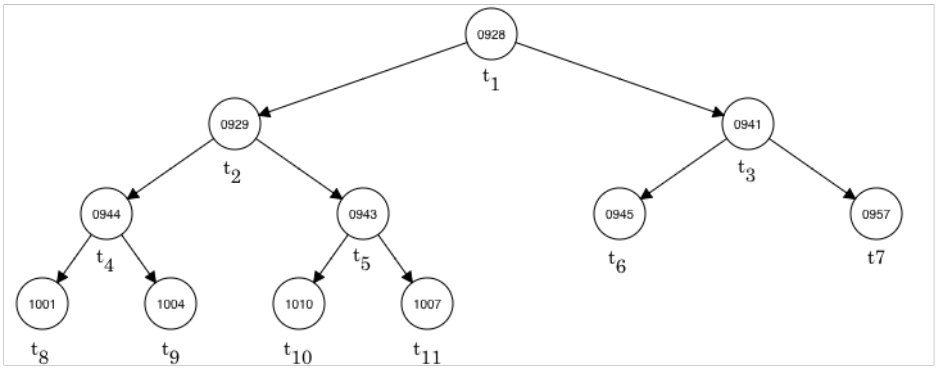

Figure 4.22: Tasks stored in a priority queue using a heap data structure

In Figure 4.22, t_1 has the highest priority, therefore it will be first task to be scheduled and assigned to a worker.

The *TaskUnit* class implements the __lt__ method which is called by the heapq package when the task is being pushed into the priority queue to ensure it is placed in the correct position within the queue.

```
1  def __lt__(self, other):
2    return self.priority() < other.priority()
3
4  def priority(self):
5    return self.release
```

Listing 4.7: __lt__ method

The *__lt__* method calls the priority method which compares the release time of one task to another to check which task has the earliest release, ensuring the priorities in the priority queue are correct.

Once the orders have been retrieved by the proactive module and scheduled they can be viewed from the web application. Various details about the order is shown to the business which can be seen in Figure 4.23. A timeline for the orders can also be viewed alongside the information about each order in 4.24

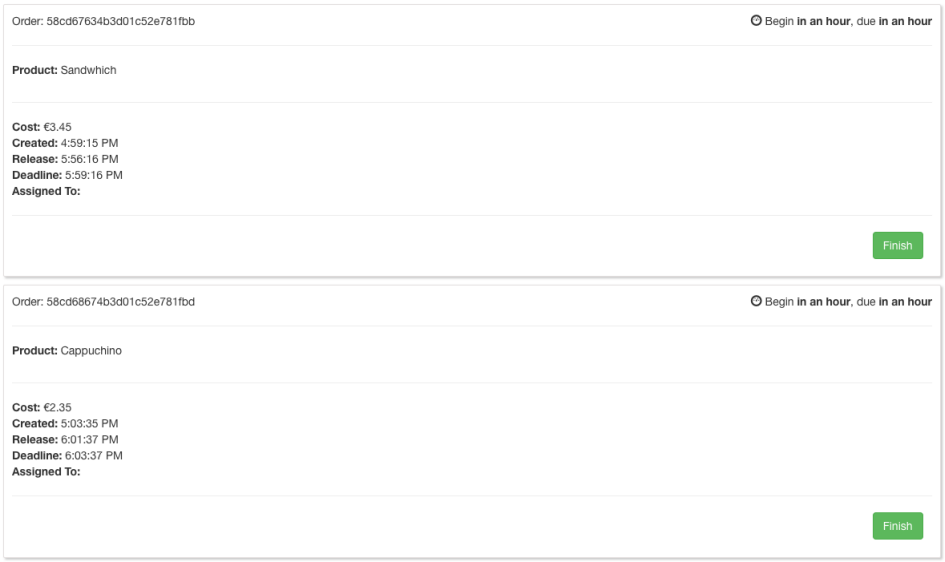

Figure 4.23: Orders displayed in web application

Figure 4.24: Orders displayed in a timeline

4.5.4 Task Conflict Detection - Interval Tree

One problem that arose after the scheduling algorithm was developed, was finding a way to detect conflicts. After some research, the interval tree data structure allowed a way to look up overlaps (conflicts) within sets of intervals (tasks). The interval tree is outlined in research in Section 2.5 along with its search operation to find interval overlaps. The Python package intervaltree[6] is used for the implementation of the interval tree. This package allows for interval tree queries of point, overlap and envelopment to made. This project only uses the point and envelope query for detecting conflicts (overlaps) of tasks.

Along with tasks being stored in a priority queue when they are added to the system for scheduling, they are also added to an interval tree for conflict detection as shown in Listing 4.8. The *intervalTree* is an instance of the *IntervalTree* class from the intervaltree package.

```
1 def _addTaskToTree(self, task):
2     """
3       Adds task to interval tree.
4     """
5     self._intervalTree.addi(
```

[6]https://pypi.python.org/pypi/intervaltree

```
6         begin=task.release,
7         end=task.deadline,
8         data=task.taskID
9     )
```

Listing 4.8: _addTaskToTree method

Once tasks have been added to the interval tree, the tree can be queried to find conflicting tasks. The code for finding conflicts between two points using the intervaltree package can be seen in Listing 4.9, where the interval tree is queried to find intervals between point 0 and point 3.

```
1 tasks = intervalTree[0:3]
```

Listing 4.9: Search for interval between two points method

The *findConflicts* method on the *TaskSet* object will find all conflicts within a set of tasks held by the *TaskSet* class. The *findConflict* method will return a *ConflictSet* instance which is a set of conflicts. The *Conflict* class and the *ConflictSet* class are used to abstract the concept of conflicts. The *Conflict* class is used to hold all tasks which conflict at some point in the schedule. The *ConflictSet* holds a set of conflicts, which means the *ConflictSet* class holds instances of the *Conflict* class. The public attributes which belong to the *Conflict* class can be seen in Figure 4.25. The implementation of the *availableWorkers* and *workersNeeded* attributes are explained in detail in Section 4.5.5 and the implementation of the *status* attribute is explained in Section 4.5.6. The *period* attribute specifies the start and end time of the conflict. The *ConflictSet* class allows for querying of conflicts of different sizes, for example the *allGreaterThan* method will find all conflicts where the highest number of tasks (k) that conflict is greater than a specified value.

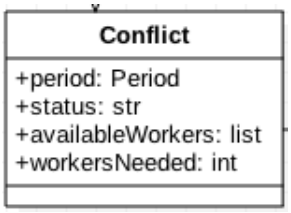

Figure 4.25: Conflict class

The code in Listing 4.10 demonstrates the main code logic for finding sets of conflicts. The main conflict detection is split into two methods *_conflictPath* and *findConflicts* The *findConflicts* method is publicly exposed and handles finding all the intervals in

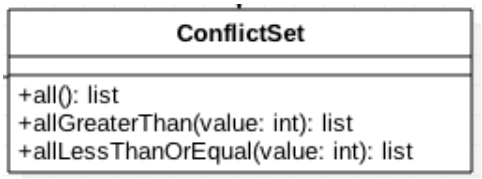

Figure 4.26: ConflictSet class

the intervaltree and then passing each interval to the *_conflictPath* which tries to find all other intervals that conflict with the interval passed.

```
def _conflictPath(self, interval, intervalTree):
    """
      @param interval The interval to find conflicts with.
      @param intervalTree The intervalTree that contains all intervals
      Finds the longest number of intervals that are all overlapping (conflicting).
        For example:
          if A and B conflict and B and C conflict and A is the
          interval we're looking for conflicts with, the returned
          intervals will be A, B, C.
        Another example:
          if D and E conflict and F and G conflict, and we're looking
          for all conflicts with D, only D and E will be returned as
          F and G are not overlapping with either D and E.
    """
    intervals = list(intervalTree.search(interval))
    # if only one interval, check if its the one we're
    # trying to find conflicts with.
    if len(intervals) == 1 and intervals[0] == interval:
      return []
    # now find the latest of all the intervals and get all conflicts
    # with and keep going until there are no more conflicts.
    latestInterval = self._findLatestInterval(intervals)
    # remove all the conflicts, we dont need to check them again.
    intervalTree.remove_overlap(interval)
    # put the latest conflict back into the tree and find its conflicts
    intervalTree.add(latestInterval)
    # now go find all conflicts with the latest interval until there are none.
    return self._orIntervals(intervals, self._conflictPath(latestInterval, intervalTree))

def findConflicts(self):
    """
      Finds all conflicts within the task set.
```

```
33      """
34      begin = self._intervalTree.begin()
35      end = self._intervalTree.end()
36      conflicts = []
37      conflictObjs = []
38      nonConflictsObjs = []
39      intervals = sorted(self._intervalTree[begin:end])
40      for interval in intervals:
41        # check if this interval was already detected to conflict
42        if self._intervalConflictAlreadyDetected(interval, conflicts):
43          continue
44        conflictIntervals = self._conflictPath(interval, self.
    _intervalTree.copy())
45        if len(conflictIntervals) > 0: # there was a  conflict
46          conflicts.append(conflictIntervals)
47          conflictObjs.append(Conflict(conflictIntervals))
48        else:
49          nonConflictsObjs.append(Conflict(interval))
50      return ConflictSet(conflictObjs), ConflictSet(nonConflictsObjs)
```

Listing 4.10: Conflict detection logic

4.5.5 Calculating Workers needed

Once conflicts have been found, the *TaskManager* class can calculate the number of workers needed throughout the schedule. The *analyseWorkersForNeededTaskSet* method on the *TaskManager* class calls the *findConflicts* method in Listing 4.10 and once the *findConflicts* method returns the *TaskManager* can analyses the number of workers that are available throughout the schedule were orders need to be processed.

```
1  def analyseWorkersForNeededTaskSet(self):
2      """
3        Analyses all the conflicts and non conflicts within the task set.
4        Set the appropriate properties for each conflict.
5      """
6      conflicts, nonConflicts = self._taskSet.findConflicts()
7      conflicts = conflicts.all()
8      nonConflicts = nonConflicts.all()
9      for conflict in conflicts:
10       begin = conflict.period.begin
11       end = conflict.period.end
12       workersNeeded = self.workersNeeded(len(conflict), self._multitask)
```

```
13        workersAvailable = len(self._workersQ.availableWorkersDuringPeriod
    (begin, end))
14        conflict.workersNeeded = int(workersNeeded)
15        conflict.availableWorkers = workersAvailable
16      for nonConflict in nonConflicts:
17        begin = nonConflict.period.begin
18        end = nonConflict.period.end
19        workersAvailable = len(self._workersQ.availableWorkersDuringPeriod
    (begin, end))
20        nonConflict.workersNeeded = 1
21        nonConflict.availableWorkers = workersAvailable
22      return conflicts, nonConflicts
```

Listing 4.11: analyseWorkersForNeededTaskSet method

Once the conflicts have been retrieved by the *TaskManager*, they are separated into two sets. The *TaskManager* also checks to see what workers are available to process the tasks. The sets of tasks that a separated are the tasks that conflict and tasks that do not conflict. For the tasks that do not conflict the number of employees needed at that point in the schedule will always be one as mentioned in Section 4.4.3. For the tasks that do conflict, the *workersNeeded* method is called to check the highest number of workers needed during that point in the schedule. The workersNeeded method uses the formula from Section 4.4.2, which is $W = \left\lceil \frac{k}{M} \right\rceil$. Listing 4.13 shows the code used in the *workersNeeded* method to implement the formula $W = \left\lceil \frac{k}{M} \right\rceil$. Once the number of workers needed to handle a conflict has been calculated, the *workersNeeded* and *availableWorkers* attributes on *Conflict* class is set.

```
1   def workersNeeded(self, k, m):
2     """
3       Calculates the number of employees needed to deal with set of
    conflicts.
4       @param k:() The number of conflicts
5       @param m:() The highest number of tasks employees can service
    simultaneously.
6     """
7     # formula: k/m
8     from math import ceil
9     return ceil(float(k)/float(m))
```

Listing 4.12: workersNeeded method

4.5.6 Utilisation Status

Once the *workersNeeded* and *availableWorkers* properties have been set on the *Conflict* object in the *analyseWorkersForNeededTaskSet* method from Listing 4.11, the *Conflict* object will then generate the utilisation status for the periods in the schedule where the conflict begins to where it ends. The formulas for utilisation status are given in Section 4.4.4, they are the following, where W is the max workers needed and w is the available workers.

- Status "ok": $W < w$
- Status "busy": $W = w$
- Status "very busy": $W > w$

```
1  def _setStatus(self):
2    if self._workersNeeded and self._availableWorkers:
3      if self._workersNeeded > self._availableWorkers:
4        self._status = "very busy" # W > w
5      elif self._workersNeeded == self._availableWorkers:
6        self._status = "busy" # W = w
7      else:
8        self._status = "ok" # W < w
```

Listing 4.13: workersNeeded method

These statuses can be viewed from the web application underneath the control panel. Figure 4.27 illustrates how this data is represented on the web application for businesses. The period, status, employees needed and available employees are shown. Each of the three statuses have there own colour coding. As well as giving a status for each period of the schedule where a tasks exists, a status for the current time is given also. This status is used by the web application to show each business how busy they currently are from the perspective of the system and it is also the status that will be shown to customers who are ordering from the mobile application.

Period	Status	Employees Needed	Available Employees
4:22:39 PM - 4:25:41 PM	ok	1	2
4:26:10 PM - 4:30:42 PM	busy	2	2
5:10:50 PM - 5:14:11 PM	very busy	3	2

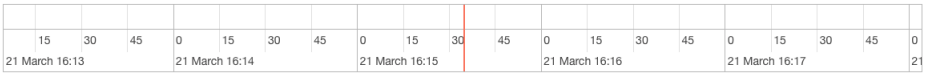

Figure 4.27: Workers needed and utilisation status from web application

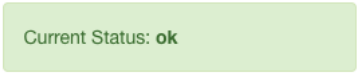

Figure 4.28: Current status shown in web application

4.5.7 Task Allocation

Task allocation is the next requirement to implement. Task allocation is important to the success of tasks being achieved before their deadline, tasks need to be assigned to workers in the correct order as dictated by the scheduling algorithm, to ensure they are done processing before or at their deadline. The number of tasks that can be assigned to workers is based on the multitask value that is specified by the business. This value could be specified by a manager whose expectation of each worker would be to process m orders simultaneously. The multitask value is a vital component when assigning workers tasks.

Once an worker is added to the system (Figure 4.19) they are placed into a circular queue to ensure that tasks are assigned to workers in the correct order. Workers should be assigned tasks in a "first in first out" (FIFO) order. By using a circular queue, once an worker is "dequeued" they will be assigned task and then enqueued back into the queue.

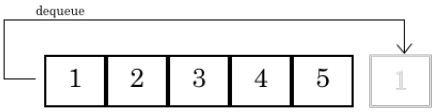

Figure 4.29: Circular queue; dequeue operation

The *TaskManager* instance keeps reference to a WorkerQueue object. The *WorkerQueue* object ensures workers are assigned tasks in a FIFO order, this WorkerQueue class behaves like a circular queue, once an employee is dequeued to be assigned tasks they are put back into the queue at the end, in Listing 4.14 once a worker is taken from the queue it is put back in, this behaviour simulates a circular queue.

```
1 def nextWorker(self):
2     worker = self._queue.get() # take worker from the queue.
3     if worker in self._workersToRemove:
4       self._workersToRemove.remove(worker)
5       return self.nextWorker()
6     self._queue.put(worker) # put the worker to the back of the queue.
7     return worker
```

Listing 4.14: nextWorker method

The *Worker* class implements all of the requirement for task allocation in Section 4.4.6. The requirements used the notations a = assigned tasks, m = multitask value. Each is shown alongside the code that is responsible for fulfilling the requirement below.

Requirement: $a < m$

The *hasReachedTaskLimit* method ensures that no more tasks can be assigned to a worker when $a = m$

```
def hasReachedTaskLimit(self):
    return len(self._assignedTasks) >= self._multitask
```

Listing 4.15: hasReachedTaskLimit method

Requirement: if $a = m$ and the task to be assigned has an earlier release time than one of the tasks already assigned to the worker

The *findSwappableTask* task method will try and swap any task already assigned to a worker whose release time is later than the task passed to the method, this ensures tasks with the earliest release time are always assigned first.

```
def findSwappableTask(self, task):
    """
      Find any task that can be swapped and returns it
      because there release is later than the task passed to the method.
    """
    for _task in self._assignedTasks:
      if task.release < _task.release and not _task.isProcessing():
        return _task
```

Listing 4.16: findSwappableTask method

Requirement: deadline of the task to be assigned is not later than the end of the workers shift, see Listing 4.17

Requirement: release time is later than the start time of the workers shift

The *availableInPeriod* method ensures that the worker is available to process the order based on their shift start time and end time.

```
  def availableInPeriod(self, begin, end):
    return begin >= self._begin and end <= self._end
```

Listing 4.17: availableInPeriod method

The *assignTasksToWorkers* method on the *TaskManager* class assign tasks to workers. It is invoked by the *PriorityProcess* class. The flowchart in Figure 4.30 shows the decision that occur when assigning tasks to workers.

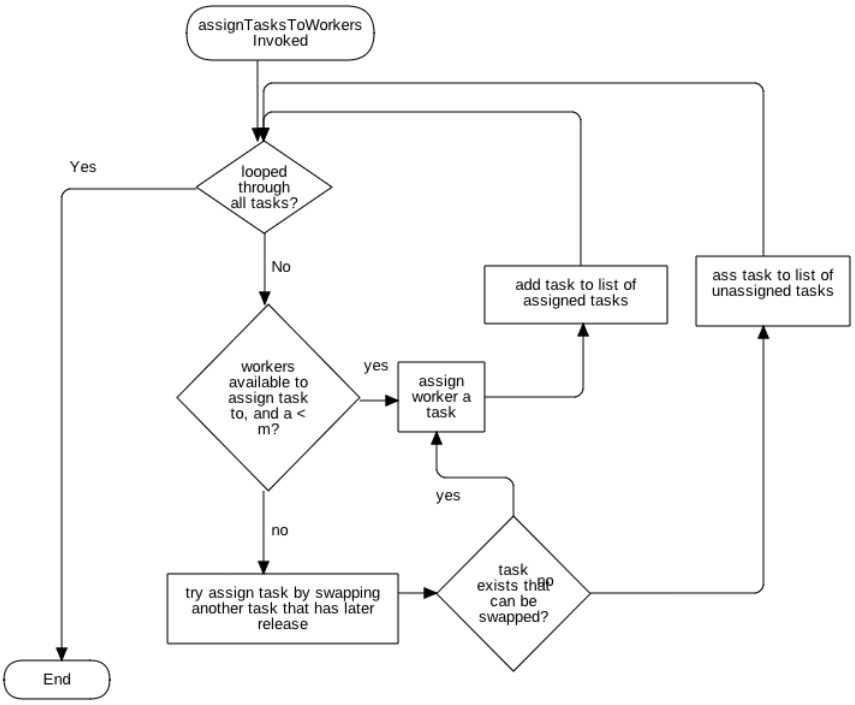

Figure 4.30: Flowchart for task assignment

Once tasks have been assigned to available employees, they can be viewed from the web application on the orders page. In the real time data section employees are displayed with their id, shift times and the number of tasks assigned to them Figure 4.31.

Current Employees

Remove	ID	Shift	Assigned
Remove	W_STEPHEN	4:00:00 PM - 9:00:00 PM	2
Remove	W_JOHN	3:00:00 PM - 8:00:00 PM	2
Remove	W_MARY	9:00:00 AM - 5:00:00 PM	2
Remove	W_PHIL	10:00:00 AM - 5:00:00 PM	1

Figure 4.31: Workers with number of tasks assigned

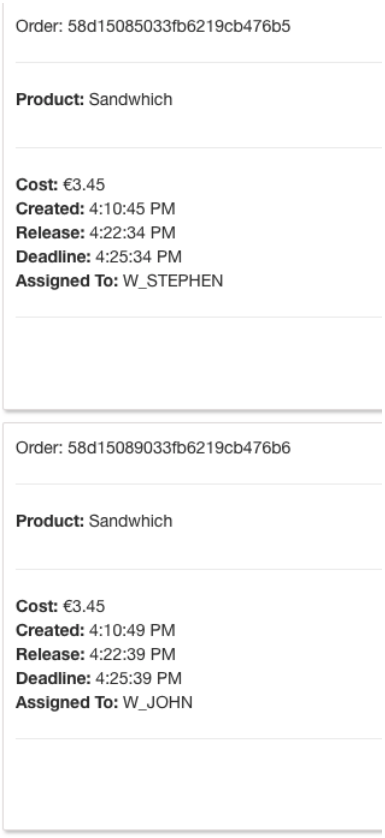

Figure 4.32: Orders with the employees assigned to process them

4.6 NuPIC Predictions

4.6.1 Overview

NuPIC was chosen based on the recommendations of this project supervisor Dr. Basel Magableh. NuPIC is used in this project to enhance the proactive module from the previous section, by predicting the max number of employees needed per hour, and the number of workers needed per hour. The code structure for this component can be seen in Figure 4.33.

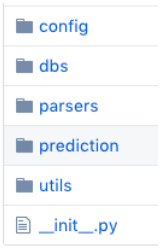

Figure 4.33: Code structure for prediction component

4.6.2 Further Research

To successfully integrate NuPIC into the project the "One Hot Gym" tutorial was undertaken to understand how NuPIC worked, which is described in Section 2.3.4. The "One Hot Gym"(OHG) tutorial formed the basis for the implementation of this component. As well as following the OHG tutorial, the NuPIC package was also explored. NuPIC also has a brilliant open source community along with a YouTube channel with many tutorials including the "Once Hot Gym".

4.6.3 Prediction types

There are two main prediction types that a business can use:

- Expected orders per hour (`ORDERAMOUNT`)
- Maximum number of employees needed per hour (`EXPECTEDEMPLOYEES`)

NuPIC can generate predictions on time based data and predict x amount of steps (hours) ahead. Both `ORDERAMOUNT` and `EXPECTEDEMPLOYEES` predictions were time based. The decision to calculate the number of orders needed per hour for a business meant retrieving historic order data from the database for a business for each hour of every day and putting it through NuPIC. The number of employees needed was more complex, this required using the formula $W = \left\lceil \frac{k}{M} \right\rceil$ from Section 4.4.4 and finding the highest value for W for each hour of every day.

4.6.4 Data Parsing

One of the main challenges of using NuPIC was correctly parsing the order data from the database into a format that could be correctly used by NuPIC. This was the first step to implementing NuPIC into the project. The *timeparsers.py* file is responsible for correctly parsing data from the database into the correct format so it can be used by NuPIC. It parses data for both `ORDERAMOUNT` and `EXPECTEDEMPLOYEES`.

`ORDERAMOUNT` parsing

The *extractHourlyOrders* function parses all hourly orders for a business from a date range.

```
def extractHourlyOrders(orders, fromDate, toDate=datetime.today()):
  """
  Extract the hourly orders for each hour from a given date range.
  @param orders:(list) A list of orders, which contain a timestamp field
    .
  @param fromDate:(datetime) The beginning of the date range.
  @param toDate:(datetime) The ending datetime range.
  @return A list of the number of orders for each hour of each day in
    the date range.
  """
  orderTimeStamps = getTimeStampsFromMongoOrderData(orders)
  toDate = datetime.today() + timedelta(days=1)
  # Every day fromDate to toDate.
  dateRange = getDaysInDateRange(fromDate, toDate)

  orderDetailsForDateRange = []
  for date in dateRange:
    orderDetails = {
      "date": object,
      "orders": []
    }
    orderDetails["date"] = date
    # Get the orders for this date
    ordersForDate = getOrdersForDate(date, orderTimeStamps)
    # If order number is zero just fill all hours with order amount = 0
    if len(ordersForDate) == 0:
      orderDetails["orders"] = zeroFillOrdersForFullDay(date)
      orderDetailsForDateRange.append(orderDetails)
      continue

    for hour in hours:
      ordersAmountForHour = len(getOrdersForHour(hour, ordersForDate))
      # As each hour only contains XX:XX, it doesn't have a date.
      # Combine the current hour iteration with the current date
    iteration
      hour = datetime.combine(date, datetime.time(hour))
      if ordersAmountForHour == 0:
        info = {
          "hour": hour,
          "amount": 0
        }
        orderDetails["orders"].append(info)
      else:
        info = {
```

```
            "hour": hour,
            "amount": ordersAmountForHour
          }
          orderDetails["orders"].append(info)
      orderDetailsForDateRange.append(orderDetails)
    return orderDetailsForDateRange
```

Listing 4.18: extractHourlyOrders function

Once the data has been parse by the *extractHourlyOrders* function it is outputted into a .csv file. The .csv file can be seen in 4.34 it contains a "timestamp" and "orders" field. The "timestamp" field is an hourly timestamp and the orders field is the number of orders made that hour for a business.

```
timestamp,orders
datetime,int
T,
2016-12-05 00:00:00,3
2016-12-05 01:00:00,5
2016-12-05 02:00:00,10
2016-12-05 03:00:00,30
```

Figure 4.34: Output from parsing order data for `ORDERAMOUNT` prediction

`EXPECTEDEMPLOYEES` parsing

The *extractHourlyConflicts* function in Listing 4.19 parses and caclulates the number of employees needed per hour.

```
def extractHourlyConflicts(orders, fromDate, toDate=datetime.today(),
    multitask=2):
  """
    Extracts the conflicts per hour for the date range from the orders.
  """
  def extractReleaseDeadline(order):
    return {
      "id": str(order["_id"]),
      "release": order["release"],
      "deadline": order["deadline"],
    }
  allConflicts = []
  # convert all order to tasks.
  orders = map(extractReleaseDeadline, orders)
  intervalTree = IntervalTree()
  for index, order in enumerate(orders):
    intervalTree.addi(
      begin=order["release"],
```

```
      end=order["deadline"],
      data=order["id"]
    )
  toDate = datetime.today() + timedelta(days=1)
  dateRange = getDaysInDateRange(fromDate, toDate)
  # now get conflicts for each hour
  for date in dateRange:
    conflictsForDate = {
      "date": date,
      "conflicts" : []
    }
    year = date.year
    month = date.month
    day = date.day
    for hour in range(0, 24):
      begin = datetime(year, month, day, hour, 00)
      end = datetime(year, month, day, hour, 59)
      conflicts, nonConflicts = findConflicts(intervalTree, begin, end)
      highest = highestConflictsForHour(conflicts)
      conflictsForHour = {
        "hour": begin,
        "size": len(highest),
        "employeesNeeded": workersNeeded(len(highest), multitask)
      }
      conflictsForDate["conflicts"].append(conflictsForHour)
    allConflicts.append(conflictsForDate)
  return allConflicts
```

Listing 4.19: extractHourlyConflicts function

Once the data has been parsed by the *extractHourlyConflicts* function it is outputted into a .csv file. The .csv file can be seen in Figure 4.35 it contains a "timestamp" and "employeesNeeded" field. The "timestamp" field is an hourly timestamp and the "employeesNeeded" field is the maximum number of employees that were needed that hour for a business. The method of finding employees needed per hour uses the conflict detection method from Section 4.5.4.

```
timestamp,employeesNeeded
datetime,int
T,
2016-12-05 00:00:00,2
2016-12-05 01:00:00,4
2016-12-05 02:00:00,1
2016-12-05 03:00:00,4
```

Figure 4.35: Output from parsing order data for `EXPECTEDEMPLOYEES` prediction

Directory Location

Once the data is parsed it is outputted into directory for the business located under /<businessid>/sources/<predictiontype>/data/<name>.csv. Each business has their own directory that contains all the necessary data for performing predictions. The .csv file used by NuPIC when running swarms and predictions. The directory structure can be seen in Figure 4.36

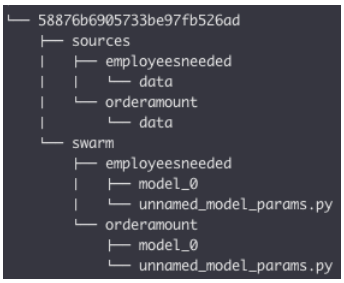

Figure 4.36: Directory structure after data parsing

4.6.5 Running swarms and predictions

The classes for running swarms and predictions can be seen in the class diagram in Figure 4.37

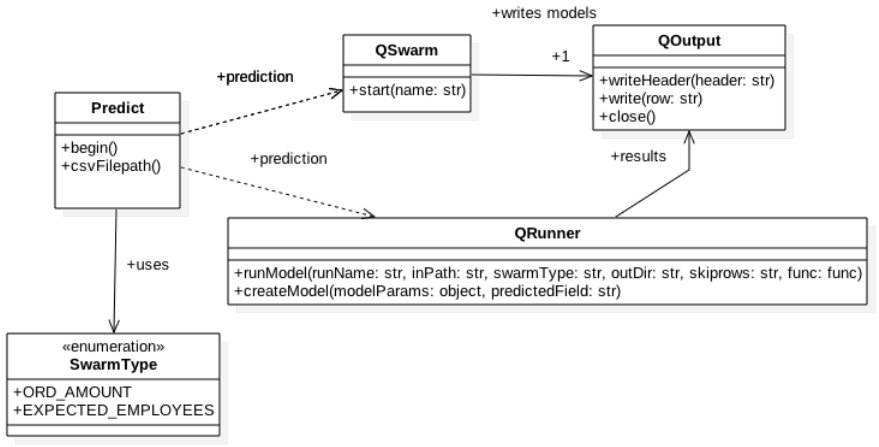

Figure 4.37: Class diagram of swarming and prediction components

Running swarms

Once the data has been parsed from the database into .csv files, NuPIC needs to swarm the data before predictions can be made. Swarming as described by Numenta works by creating and trying multiple different models on a data set and outputting

the parameters for the one model that performed the best (generated the lowest error score) [15].

Swarming is handled by the QSwarm class, the start method on the QSwarm class will run a swarm.

```
def start(self, name="unnamed"):
    """
      Starts a new swarm.
      @param swarmDir: (string) The directory for the swarm data.
      @param name: (string) The name to call the swarm.
      @param (object) The model parameters generated from the swarm.
    """
    self._swarmName = name
    return self.__swarm()
def __swarm(self):
    """
    Starts a swarm and writes a generated files to swarm/ directory for
    the business.
    @return modelParam:(object) Models Parameters object.
    """
    # Create directory for swarm details
    swarmWorkDir = self.__createSwarmWorkDir()
    modelParams = permutations_runner.runWithConfig(
        self._swarmDescriptionObject,
        {"maxWorkers": 2, "overwrite": True},
        outputLabel=self._swarmName,
        outDir=swarmWorkDir,
        permWorkDir=swarmWorkDir
    )
    # Write the model parameters to swarm directory.
    self.__writeModelParams(modelParams)
    return modelParams
```

Listing 4.20: starting a swarm function

The *permutations_runner.runWithConfig* method in Listing 4.20 analyses the parsed data either for `ORDERAMOUNT` or `EXPECTEDEMPLOYEES` predictions and outputs the best model that is found. The *runWithConfig* method takes a swarm description, which tells it which fields to analyse and location of parsed data to analyse. The swarm descriptions for this project are located in the swarm_desc_templates/ directory. There are two files *order_amount_template.json* and *expected_employees_template.json*, both of which are swarm descriptions that can be used for the two types of predictions.

The most important parts of the swarm description for the ORDERAMOUNT prediction are shown in Listing 4.21. Although Listing 4.21 shows parts of swarm description for the ORDERAMOUNT prediction, the swarm description for EXPECTEDEMPLOYEES is almost identical, however the fields to swarm over are different.

- includedFields which tells NuPIC which fields to swarm over, for the ORDERAMOUNT prediction this is the timestamp and the number of orders per hour.
- inferenceArgs which tells NuPIC field should be predicted and the steps ahead to predict. In this case their is only 1 prediction step meaning only one hour ahead will be predicted.

```
"includedFields": [
    {
      "fieldName": "timestamp",
      "fieldType": "datetime"
    },
    {
      "fieldName": "orders",
      "fieldType": "int"
    }
  ]
"inferenceArgs": {
    "predictionSteps": [ 1 ],
    "predictedField": "orders"
  }
```

Listing 4.21: Swarm Description for ORDERAMOUNT

Once the *runWithConfig* method has finished a model is created and is written under the directory location:
/<businessid>/swarm/<predictiontype>/model/ shown in Figure 4.36. Models can be saved on disk in a .py file or they can be used in memory.

Predictions

Once the model is created from swarming it is given to a *QRunner* object which will run the actual predictions. The predictions are run by invoking the *runModel* method on the *QRunner* class. This method will takes the following arguments:

- runName: the name for the run
- inPath: the location of the parsed data from Section 4.6.4
- swarmType: either `ORDERAMOUNT` or `EXPECTEDEMPLOYEES`
- outDir The path to output the prediction data
- skiprows The number of rows to skip in the parsed data .csv file.
- func A callback function that is called for every row in the .csv to make sure that it is in a format acceptable by NuPIC

Once the *runModel* method has finished it will output the predicted data for the number of orders expected per hour or the max number of expected employees needed per hour depending on the prediction that was run. This data is outputted into the same directory that was created for the business during data parsing. Figure 4.38 shows the predicted data generated by NuPIC.

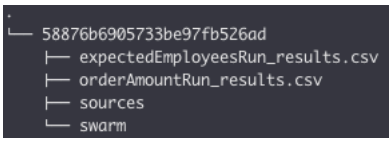

Figure 4.38: Location of prediction results

The *orderAmountRun_results.csv* is the output for the `ORDERAMOUNT` prediction and *expectedEmployeesRun_results.csv* is the output for the `EXPECTEDEMPLOYEES`. These files now contain0 predictions for the one hour in the future. Figure 4.39 shows the format of the *orderAmountRun_results.csv* file and Figure 4.40 The prediction field is the prediction generated by NuPIC for the next hour.

```
timestamp,orders,prediction
2016-12-22 00:00:00,20,14
2016-12-22 01:00:00,14,29
2016-12-22 02:00:00,27,30
2016-12-22 03:00:00,7,5
```

Figure 4.39: Prediction results for `ORDERAMOUNT` run

```
timestamp,employeesNeeded,prediction
2016-12-22 00:00:00,3,2
2016-12-22 01:00:00,2,2
2016-12-22 02:00:00,2,4
2016-12-22 03:00:00,4,2
2016-12-22 04:00:00,5,4
```

Figure 4.40: Prediction results for `EXPECTEDEMPLOYEES` run

Once the prediction data had been successfully generated it was not much use as .csv files. This meant the prediction data needed to be persistently stored in the

MongoDB database that was being used by the other parts of the system. Once the data is prediction data is generated by NuPIC it is written to the database. The collection for storing the generated data is the "predictions" collection. It contains the two types of predictions for each business. Figure 4.41 and Figure 4.42 shows an example of a `EXPECTEDEMPLOYEES` prediction run for a business and how it is stored in the database.

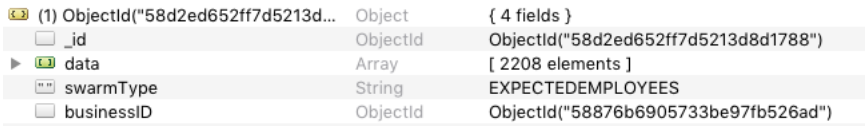

Figure 4.41: Predictions collection

Figure 4.42: Predictions collection records

Once the predictions have been added to the database, they can be viewed from the web application. Figure 4.43 shows how the prediction data is presented via the web application.

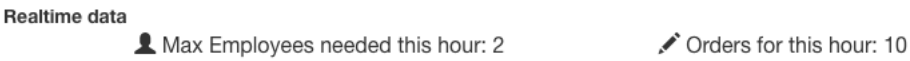

Figure 4.43: NuPIC predictions presented in web application

Running predictions from command line interface

Predictions can be generated from a command line interface, using the "run.py" script. This script allows the two prediction types `ORDERAMOUNT` and `EXPECTEDEMPLOYEES` to be run. This script completely automates the process of generating predictions by fetching the data from the database, parsing the data, running swarms, predictions, outputting to .csv files and inserting into back into the database. To script can take the following arguments:

- -s, –swarmtype [expectedemployees,orderamount] The swarm/ prediction type to perform.

- -b, –businessid The id of the business.
- -m, -monthsprior How many months of data should be swarmed over.
- -d, –dir The base directory to write the files to, if directory does not exists, it will be created.
- –multitask The multitask value (This is used during `EXPECTEDEMPLOYEES` run.)

Listing 4.6.5 show an example of the two types of prediction runs being used.

```
python run.py -s orderamount -b 58876b6905733be97fb526ad
-d ./Desktop/prediction
python run.py -s expectedemployees -b 58876b6905733be97fb526ad
-d ./Desktop/prediction --multitask 2
```

After running the two prediction types, the predictions were graphed alongside the actual data, in Figure 4.44 the `ORDERAMOUNT` prediction run and Figure 4.45 the `EXPECTEDEMPLOYEES` run.

4.6.6 Issues

One of the challenges with integrating NuPIC into the system was the lack of real data to use for running predictions, however this was solved by using a Python script (‘generatedata.py‘). The Python script can generate large amounts of order records for a given time frame. This script was used to generate large data sets of orders throughout development and testing. The script is also during the evaluation discussed in Section 6.3 to analyse the accuracy of NuPIC‘s predictions. The script will randomly generate orders within a start date and end date. The script takes the following arguments:

- -s, –start Start date
- -e, –end End date
- -n, –number The number of records to generate

```
python generatedata.py -s 10102016 -e 31032017 -n 150000
```

Listing 4.22: ‘generatedata.py‘ script example

Listing 4.22 shows an example of the script being used to generate 150,000 order records between 10-10-2016 and 31-03-2017.

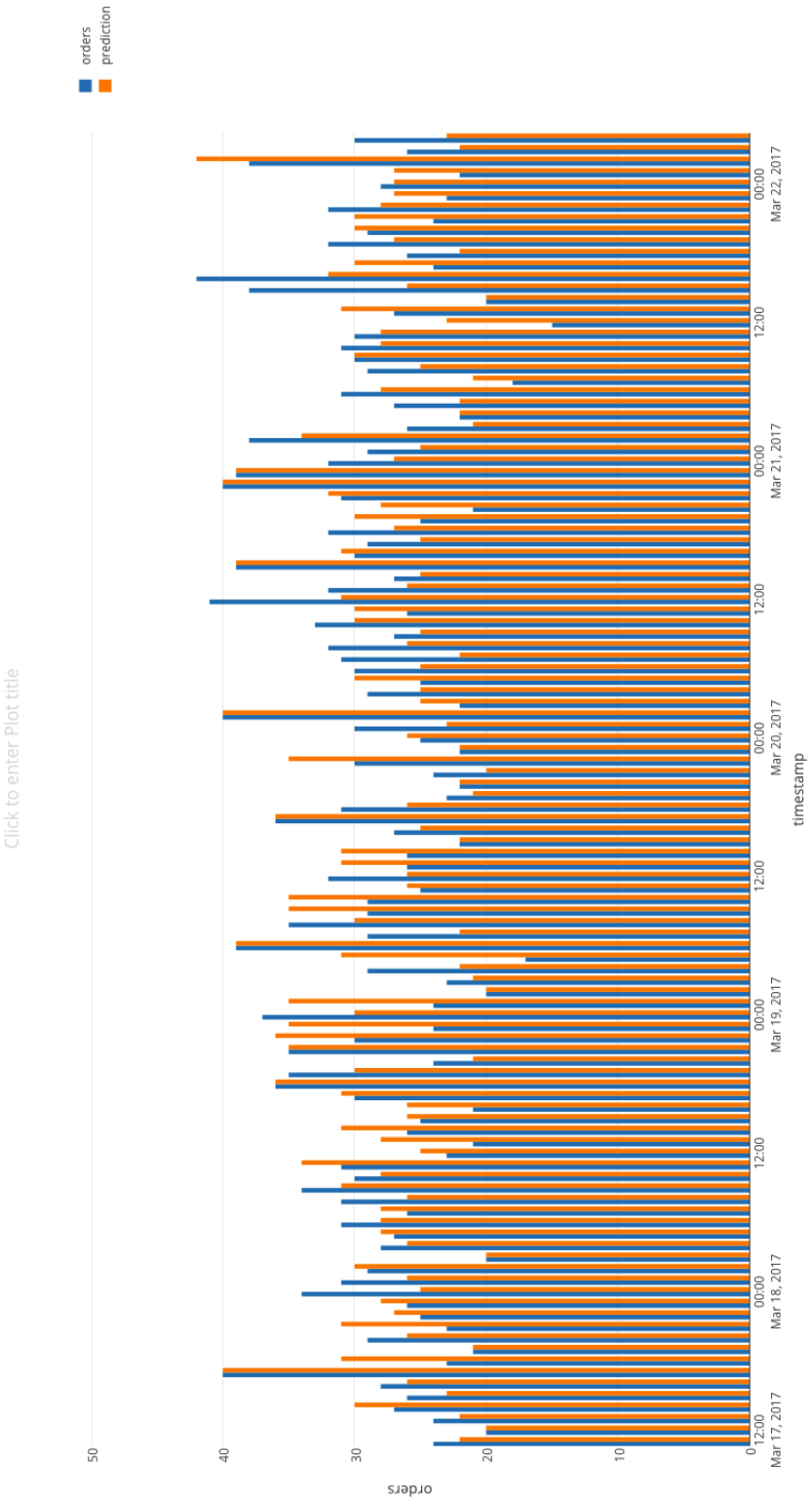

Figure 4.44: Prediction generated by NuPIC for `ORDERAMOUNT` graphed

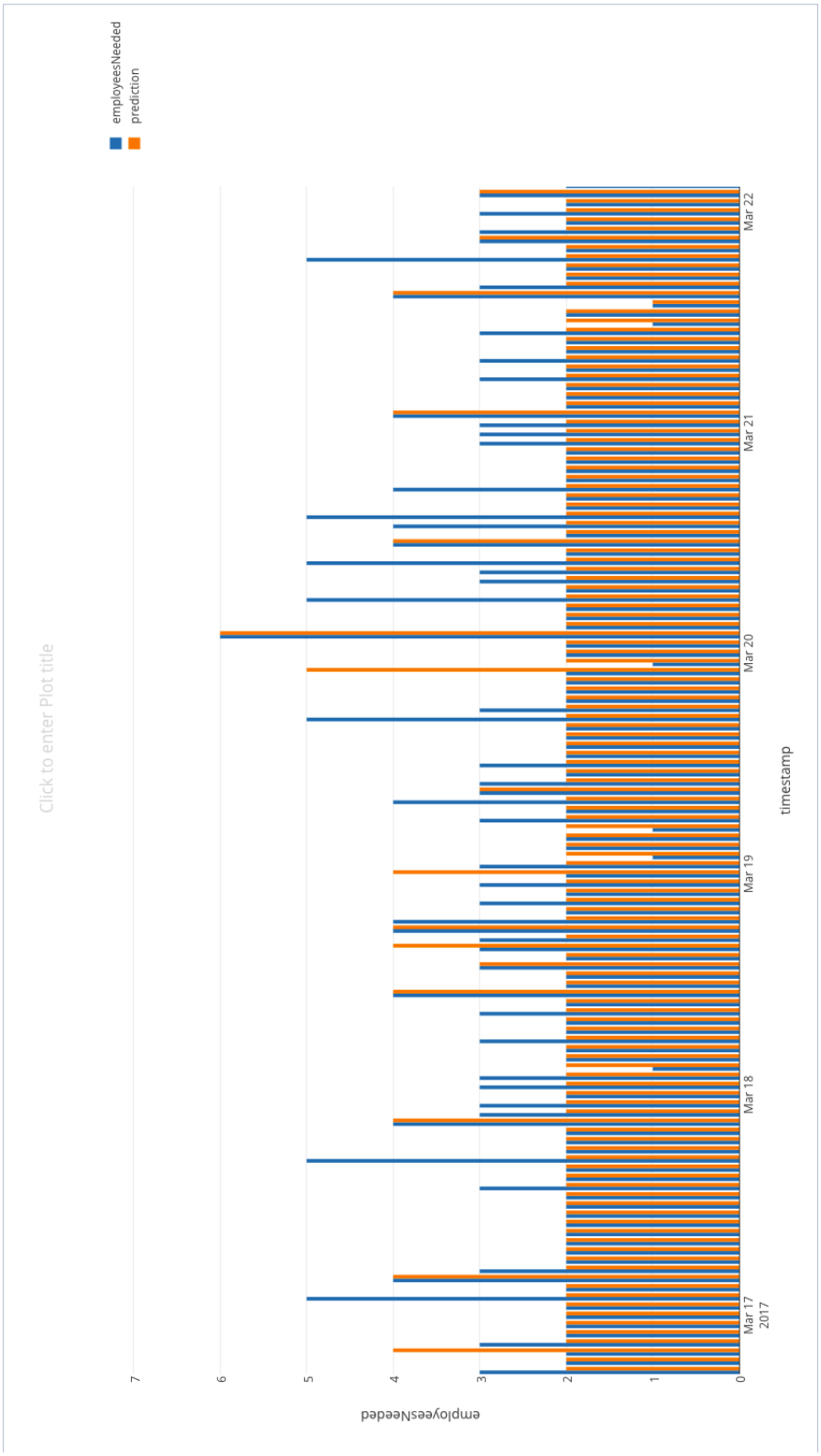

Figure 4.45: Prediction generated by NuPIC for `EXPECTEDEMPLOYEES` graphed

4.7 Web Application for businesses

The web application is built using the AngularJS version 1.5. framework along with CSS and HTML using Bootstrap. Through this web application each business can manage products, view orders and view prediction data generated by NuPIC. The web application communicates with the Node.js application on the web server and the proactive module over HTTP. The web application is composed of three separate components:

- Register and login
- Product management
- Order management

4.7.1 Register and login

Before a business can access the web application, they must first register. When registering they must enter details about the business such as name, number, opening time and close time and their address. The address is an important component for the system as it is geocoded into coordinates before it is stored in the database. The coordinates of the business is used during ordering so the expected arrival time of a customer can be calculated by Google Maps Distance Matrix API. Once business have registered they can be viewed from the iOS application.

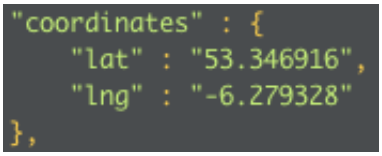

Figure 4.46: Business coordinates stored in the database, once geocoded

4.7.2 Product management

After researching into some common components of OMS, product management became a necessary requirement for the web application. It is needed by businesses so they can sell sell their products through the system and by customers so they can make orders. Each product can have many options such as size. Figure 4.47 shows

how a product is added to the system for a business. When adding a product a business can optional input an id for the product that already exists and optionally add any extra options a product may have. The name, description, price for product are also inputted along with the processing time for the product. The processing time is the time in minutes the product would take to service. During the research phase and discussion with Urbanity Cafe in Section 2.2.3, they mentioned that it would be possible for an experienced employee to know the processing time of a product.

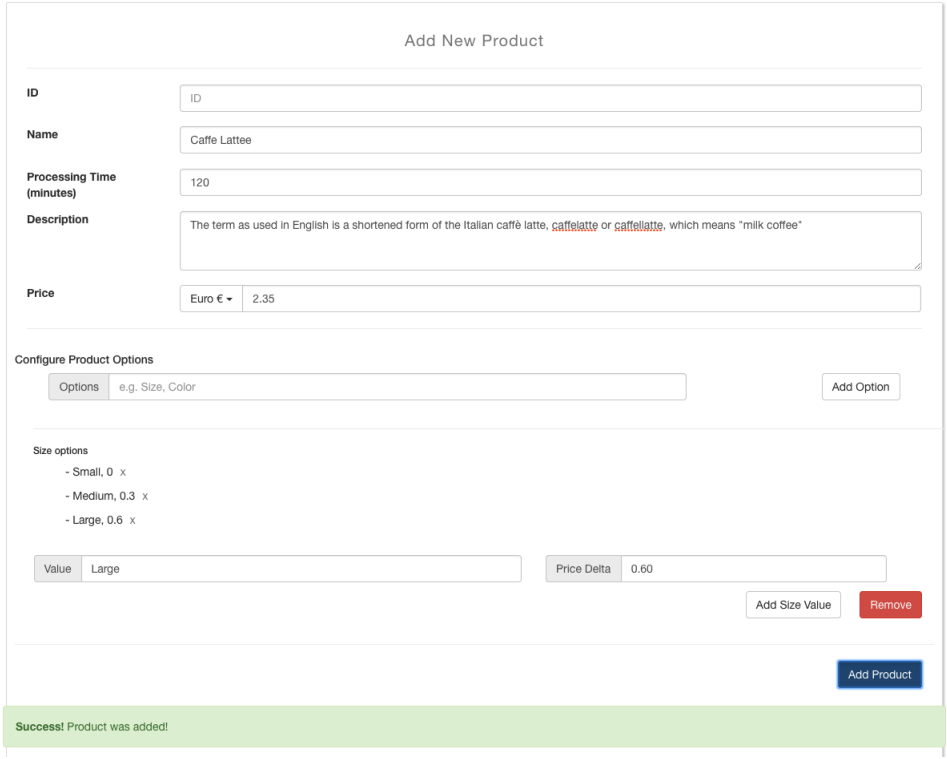

Figure 4.47: Add Product Page

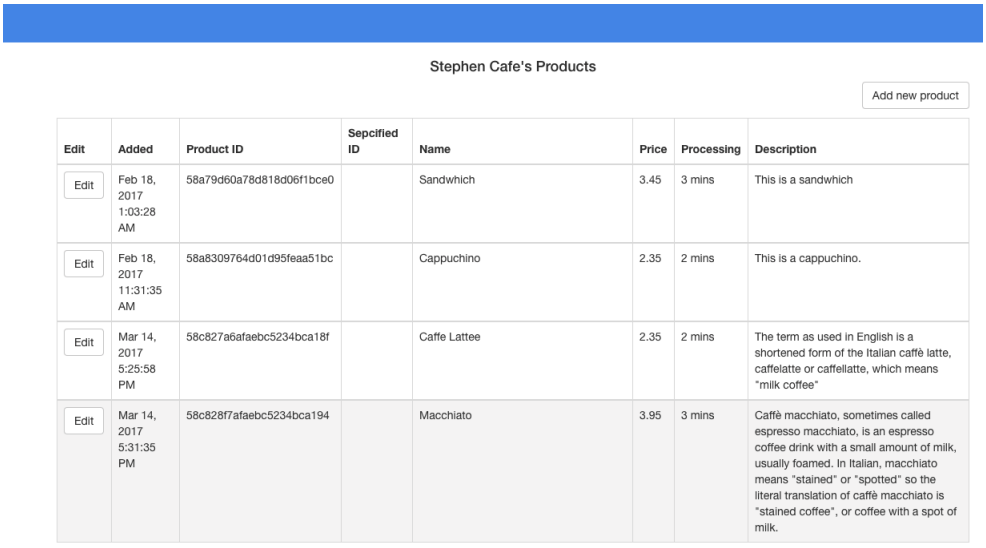

Figure 4.48: All Products Page

Once a product has been added to the system for a business they can view a list of all their products. Figure 4.48 shows how a business can view all their products as well as further manage them through editing.

4.7.3 Order Management

The order management component of the web application allows employees to interact with orders that have been made by customers. Orders are visible through the web application once they are made by a customer through the iOS application. The following processes occur before the orders are available through the web application, they are:

- Customer makes order
- Order is sent to Node.js application
- Node.js application queries Google Maps Distance Matrix API with customer's location to find arrival time
- Order is inserted into the database
- The proactive module reads order from the database
- Proactive module schedules the order appropriately
- Web application queries proactive module for scheduled orders
- Web application displays the order

These steps can be seen in the activity diagram in Figure 4.49 which outlines all activities that occur before the order reaches the web application.

Once the orders are available from the proactive module they can be viewed through the web application. Through the orders page, businesses can view all the information generated from the proactive module in Section 4.5.2 and NuPIC in Section 4.6, which is the orders, the order schedule, utilisation status, number of employees and the prediction data. This data is fetched from the proactive module every 3 seconds by the web application.

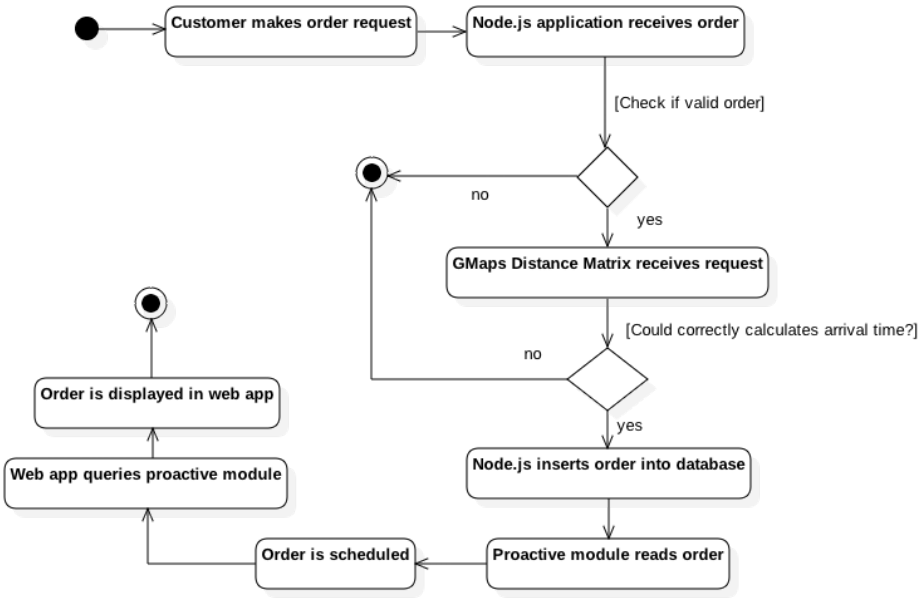

Figure 4.49: Order activity diagram

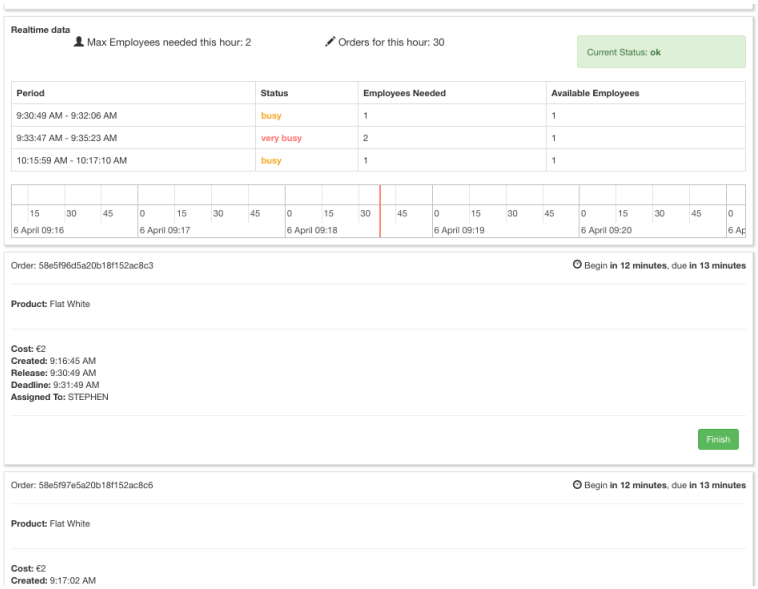

Figure 4.50: Orders Page

4.7.4 Git and GitHub

Version control was used throughout the development of all software components for this system. Git was used for version control and GitHub was used as the remote server to which the code was pushed to. While using Git, developments were made on the "dev" branch keeping the master branch only for releases, free of development code. The repositories tracked by Git can be seen in Figure 4.51.

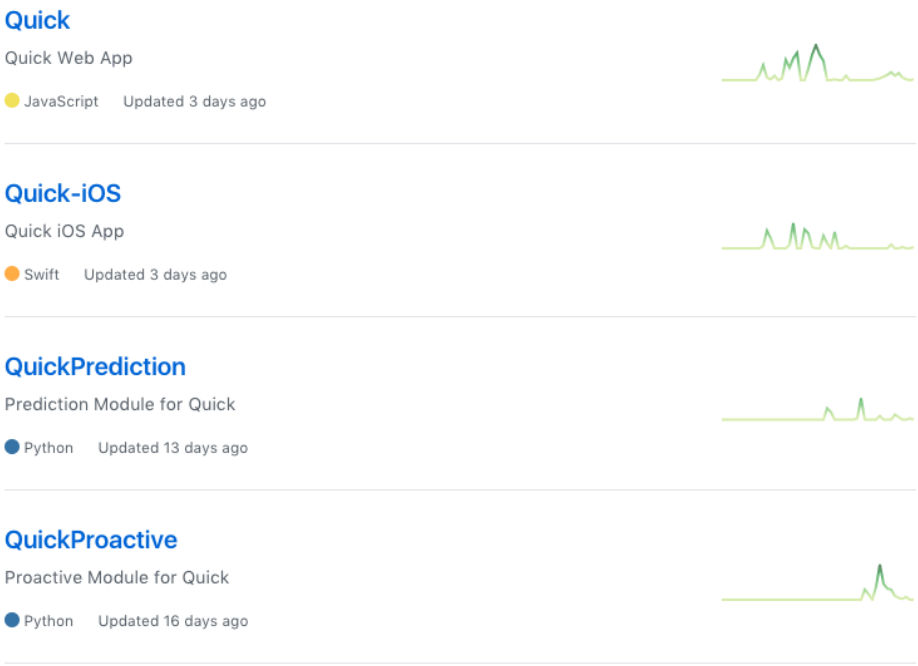

Figure 4.51: GitHub repositories

Chapter 5

Testing

5.1 Overview

This section contains the testing carried out on the system. Throughout development of the system constant testing was carried out. Once a functionality had been completed it would be tested. The testing techniques used include unit testing, continuous integration and test cases.

5.2 Testing

5.2.1 Unit testing

Introduction

Unit testing was used in this project to ensure correctness and reliability for the code developed. Martin Fowler describes a "unit" [20] as the following 'Object-oriented design tends to treat a class as the unit, procedural or functional approaches might consider a single function as a unit'. This section explains how unit testing was used in the proactive module and the Node.js application.

Proactive Module Unit Testing

Unit testing was carried out the proactive module using Python's unittest[1] package. This package is the de facto standard unit testing framework for Python and is based on JUnit[2], Java's unit testing framework by Kent Beck and Erich Gamma. The proactive module has a mixture of object oriented code with classes and some procedural code with just functions. The proactive module has the most unit test written as it contains the most complex source code. There are a total of 53 unit tests written for the proactive module all of which cover everything from scheduling to task allocation. The scheduling and task allocation features have been tested extensively, as they are vital to the correctness of this project.

Figure 5.1: All tests passing for proactive module

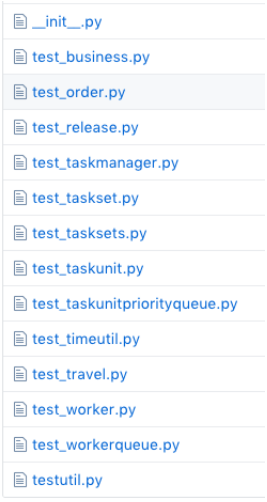

Figure 5.2: Test files for proactive module

Figure 5.2 shows all the test files for the unit tests written. The tests were run every time a commit was pushed to GitHub using Travis CI with is explained in Section 5.2.2. Listing 5.1 shows an example of a unit test written for the proactive module to check that a worker is available within a specific time period.

```
1 def test_availableInPeriod(self):
2     worker = Worker(workerID="W1", begin=tHour(0, 00), end=tHour(23, 59)
      , multitask=2)
3     available = worker.availableInPeriod(begin=tHour(12, 00), end=tHour
      (13, 00))
```

[1] https://docs.python.org/2/library/unittest.html
[2] http://junit.org/junit4/

```
4     self.assertTrue(available)
```

Listing 5.1: Example of unit test for proactive module

Node.js unit testing

For unit testing the Node.js application MochaJS[3] was used, which is described as a "feature-rich JavaScript test framework" on the MochJS website. Most of the tests written for the Node.js application were to test the REST API, as this was the most important feature to ensure the overall success of this component of the system. The test files can be see in Figure 5.4.

(a) Test for routes (b) Tests for Prototypes

Figure 5.3: All tests for Node.js

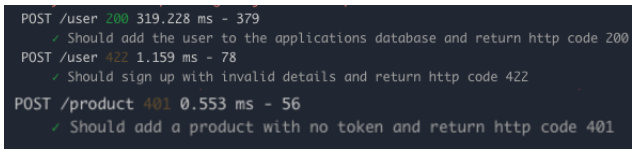

Figure 5.4: Sample output of tests for Node.js application

Listing 5.2 shows an example of a unit test for the product endpoint in the Node.js application.

```
1  it ('Should add a product with no token and return http code 401',
       function(done) {
2     // Make POST request.
3     request
4     .post('/product')
5     .expect(401)
6     .expect(function(res) {
7       expect(res.body.success).to.equal(false);
8     })
9     .end(function(err, res) {
10      done(err);
11    });
```

[3]https://mochajs.org/

```
12 });
```

Listing 5.2: Example of unit test for proactive module

5.2.2 Continuous Integration - Travis CI

Continuous integration is a method of developers integrating work into shared repository several times a day [21]. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. One of the key benefits of integrating regularly is that you can detect errors quickly and locate them more easily. Continuous integration was used for the proactive module using Travis CI[4]. Travis CI is a online continuous integration service used to build and test software projects hosted at GitHub. Every time a commit is push to GitHub, TravisCI will build and test the software.

Although this project had one developer continuous integration was still beneficial, the main benefit for using continuous integration for this project was the continued running of tests for every commit. For example, the proactive module has approximately 119 commits as of writing this, 75 of which have successfully passed all of the tests that have been run by Travis CI.

Figure 5.5: Commits for proactive module

To integrate successfully with Travis CI a *travis.yml* file is needed. This file tells Travis how to run the build and which tests should be run.

```
1  language: python
2  python:
3    - "2.7"
4  branches:
5    only:
6      - dev
7
8  install: "pip install -r requirements.txt"
9
10 script: python -m unittest tests
```

Listing 5.3: travis.yml file for TravisCI continuous integration

[4]https://travis-ci.org/

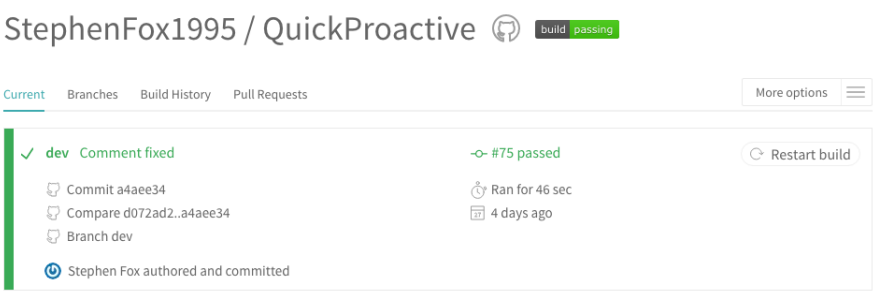

Figure 5.6: TravisCI for proactive module

By having TravisCI continuously run tests, it makes sure that any time a new commit is push to GitHub that a feature isn't broken. This helped with managing the code base as it got larger, more features were added and the complexity increased. This helped the overall correctness and reliability of the code base over the time of development.

5.2.3 Manual testing

Manual testing was performed on the web and iOS application to ensure the expected behaviours were executed when interacting with the interfaces. Table 5.1 shows all manual tests performed on the system.

5.3 Conclusion

Testing was performed to ensure the overall correctness and reliability of the system. For the proactive module which contains some of the most complex logic of the system, unit testing allowed for test cases to be run to check that features such as the scheduling algorithm and task allocation were implemented as they were set out in Section 4.4. Continuous integration through TravisCI allowed for tests to be constantly ran every time a change was committed. This helped ensure that each development didn't break other components within the proactive module. The REST endpoints in the Node.js application were also unit tested ensuring that they performed the correct logic. Manual testing was also used to test that the functionality of the system was correct from the perspective of the iOS application and web applications.

No	Component	Description	Steps	Expected	Result
1	Web application	Signup business	1. Go to business signup page 2. Enter business details 3. Click signup button	Business is signed up and logged into the system	PASS
2	Web application	Login business	1. Go to login page 2. Enter email and password 3. Click login	Business is logged into the system	PASS
3	Web application	Add product	1. Go to add product page 2 Enter product details 3. Click add product button	Product is added for a business	PASS
4	Web application	View products	1. Go to products page	All products for a business are loaded	PASS
5	Web application	Edit products	1. Go to products page 2. Select product 3. Click edit 5. Change product details 6. Click done	Product should reflect edits	PASS
6	Web application	Start proactive module	1. Go to orders page 2. Enter multitask value 3. Click begin	All order data is loaded	PASS
7	Web application	Add employee	1. Go to orders page 2. Enter employee id and shift 3. Click add	Employee is added to handle orders	PASS
8	iOS application	Customer Sign up	1. Go to signup page 2. Enter name, email, password 3. Click signup	Customer is signed up and logged into the app and brought to the homepage	PASS
9	iOS application	Customer login	1. Go to login page 2. Enter email and password 3. Click signin	Customer is signed in and brought to the homepage	PASS
10	iOS application	View business utilisation status	1. Click a business	Utilisation status of business is loaded	PASS
11	iOS application	Add product to order	1. Go to products page for a business 2. Click product 3. Click add to order	Product is added to order	PASS
12	iOS application	Add options to product before order	1. Go to product page 2. Click product 3. Select options 4. Click add to order	Product with options is added to order	PASS
13	iOS application	Make order	1. Click order 2. Select travel method	Success message appears with order collection time	PASS

Table 5.1: Manual testing performed

Chapter 6

Evaluation

6.1 Overview

This chapter describes the methods of evaluating the project. The evaluation methods included a comparison for the Earliest Deadline First scheduling algorithm from Section 2.4.6 and the scheduling algorithm from Section 4.4 which was developed for this project to evaluate how each algorithm will schedule orders in this project. The accuracy of the NuPIC predictions is also evaluated and an evaluation of the system by Urbanity Cafe.

6.2 Comparison of scheduling algorithms

6.2.1 Outline

The comparison is used to analyse the overall earliness of each task in the schedule produced by the algorithms. The earliness of a task is the amount of time remaining before its deadline remaining [3]. Lateness is the amount of time by which a task misses its deadline [3]. Earliness is used to compare the two algorithms instead of lateness because the effect of lateness is left to a business to decide how to deal and not the scheduling algorithm, the scheduling algorithm is used to advise businesses up until the deadline of each task, after that it is left to the business to decide what to do. This was also discussed in Section 4.4.7. The terminologies, assumptions, notations and parameters for the comparison are outlined below.

6.2.2 Terminologies

For the comparison the Earliest Deadline First scheduling algorithm will be referred to as algorithm1 and the algorithm developed for this project in Section 4.4.2 will be referred to as algorithm2.

6.2.3 Assumptions

- Tasks take no longer to process than specified by their C_i (processing time) property
- Tasks are not periodic, so once a task is scheduled, it will never be scheduled again according to some periodic rate. Periods are explained in Section 2.4.3.

6.2.4 Notations

Two new notations are needed in this section to compare the two algorithms. They are:

- f_i: Finish/ completion time of task t_i.
- e_i: The earliness of task t_i, which is the amount of time still remaining before its deadline (d_i) [3]. The earliness can be calculated using the following formula $e_i = d_i - f_i$, (deadline - finish time)

6.2.5 Parameters

The tasks used to compare the two algorithms can be seen in Table 6.1, along with the processing time, deadline and release time (only used for algorithm2).

t_i	C_i	d_i	$r_i = d_i - C_i$
t_1	1	3	2
t_2	2	6	4
t_3	1	7	6
t_4	3	10	7
t_5	1	12	11

Table 6.1: Tasks

6.2.6 Reminder

A reminder of how the two algorithms schedule tasks.

algorithm1 (Earliest Deadline First algorithm)

```
for each time-point:
      for each available task:
            schedule the task with earliest deadline time
```

algorithm2 (Algorithm developed for this project)

```
for each time-point:
      for each available task:
            schedule the task with earliest release time
```

6.2.7 Results

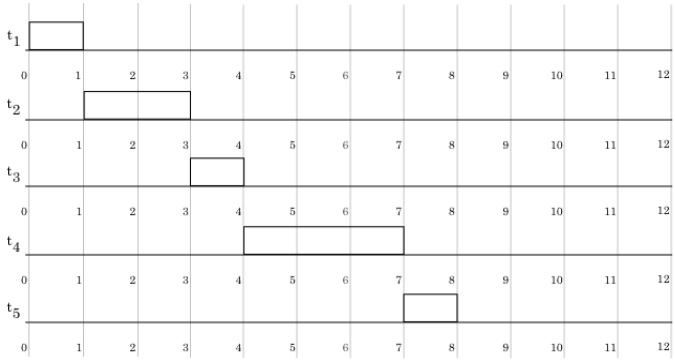

Figure 6.1: Schedule produced by algorithm1 of tasks from Table 6.1

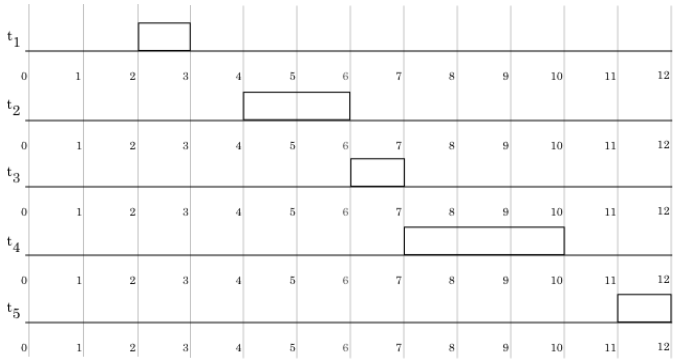

Figure 6.2: Schedule produced by algorithm2 of tasks from Table 6.1

Figure 6.1 and Figure 6.2 show a comparison of the two schedules that are generated by the two scheduling algorithms. Figure 6.1 shows the schedule produced by algorithm1

and Figure 6.2 shows the schedule produced by algortihm2. The schedule produced from algorithm1 schedules all the tasks as soon as possible, with an average earliness of 3, compared to the algorithm2 which has an average earliness of 0. The earliness calculations for each scheduling algorithm can be seen in Table 6.2 and Table 6.3. In Table 6.3 earliness has been completely reduced to 0 for all tasks. Earliness needed to be removed or reduced to ensure that tasks finish as close to their deadline as possible or at their deadline so that when customers arrive to collect their orders they are as fresh as possible. It can be seen from the Table 6.2 and Table 6.3 which algorithm performed better for this requirement, algorithm2 - which was the algorithm developed for this project.

t_i	C_i	d_i	f_i	$e_i = d_i - f_i$
t_1	1	3	1	2
t_2	2	6	3	3
t_3	1	7	4	3
t_4	3	10	7	3
t_5	1	12	8	4

Table 6.2: Earliness of tasks scheduled by algorithm1

t_i	C_i	d_i	f_i	$e_i = d_i - f_i$
t_1	1	3	3	0
t_2	2	6	6	0
t_3	1	7	7	0
t_4	3	10	10	0
t_5	1	12	12	0

Table 6.3: Earliness of tasks scheduled by algorithm2

6.3 Accuracy of NuPIC

This section explains the method of evaluating the accuracy of the predictions generated by NuPIC.

6.3.1 Parameters

For this evaluation approximately 150,000 order records were generated and put into the database for NuPIC to use for predictions. The records were generated for a three month period between December 2016 to March 2017. One week of data was chosen at random for evaluating the prediction on the number of employees predicted and the number of orders predicted.

6.3.2 Results

The results measure the accuracy of the prediction generated for one hour into the future versus the actual data for that hour for each of the predictions. The results for the expected number of orders per hour can be seen in Table 6.4 with an average accuracy of 87% and the expected number of employees needed per hour can be viewed Table 6.5 with an average accuracy of 86%.

Hour	Mon	Tue	Wed	Thu	Fri	Sat	Sun
00:00	95	88	100	86	97	86	40
01:00	83	100	100	95	100	92	100
02:00	64	92	87	71	79	84	100
03:00	100	81	100	97	100	100	100
04:00	95	91	71	88	100	84	86
05:00	89	68	95	88	93	100	100
06:00	64	100	87	67	63	100	58
07:00	100	100	97	88	97	84	100
08:00	100	88	100	100	100	100	100
09:00	91	56	97	100	100	74	91
10:00	100	84	50	95	100	100	66
11:00	71	77	87	84	100	100	100
12:00	88	73	67	87	88	73	90
13:00	75	72	72	100	94	100	100
14:00	86	88	85	55	100	76	100
15:00	68	100	100	59	84	76	100
16.00	95	81	100	97	84	87	86
17:00	95	79	100	100	90	97	100
18:00	86	79	100	100	90	89	100
19:00	90	84	87	65	65	84	82
20:00	100	100	59	76	88	86	89
21:00	100	93	75	95	80	84	92
22:00	78	93	86	87	63	85	100
23:00	80	93	100	85	68	83	100

Table 6.4: Order amount prediction accuracy (%)

Hour	Mon	Tue	Wed	Thu	Fri	Sat	Sun
00:00	100	85	100	25	10	55	100
01:00	100	100	100	100	62	72	71
02:00	87	71	100	100	100	100	100
03:00	100	100	100	100	5	100	72
04:00	100	100	100	100	85	100	55
05:00	100	62	70	66	72	100	60
06:00	50	71	100	100	100	66	100
07:00	50	100	100	100	66	100	100
08:00	100	100	100	100	100	100	100
09:00	100	100	100	77	70	100	100
10:00	100	100	10	66	100	100	72
11:00	50	87	100	77	66	66	100
12:00	90	100	100	100	71	100	72
13:00	100	100	100	100	50	100	100
14:00	75	100	77	37	50	66	100
15:00	75	100	100	100	100	100	100
16:00	50	100	100	100	50	100	100
17:00	100	37	100	100	80	100	100
18:00	60	100	66	100	63	85	100
19:00	60	100	100	100	75	100	66
20:00	100	20	62	100	71	90	100
21:00	100	100	100	100	100	100	83
22:00	50	100	100	100	100	100	71
23:00	50	100	100	87	100	100	71

Table 6.5: Employee amount prediction accuracy (%)

6.4 User acceptance: Urbanity Cafe Feedback Session

Once the system had been developed, Urbanity Cafe gave some feedback of the system. Below is a summary of the feedback received after feedback session with Urbanity Cafe.

- The interface on the web application is clear and easy to navigate
- It is easy to see which orders should be prioritised
- The system could help with time management for employees as they can see which times of the day are busy
- Multitask value could be good for managers to set expectations for employees i.e how well they can be utilised
- Task allocation would be better to have on a product basis as some employees cannot process all products within an order. For example in Urbanity Cafe one employee handles coffee orders and another employee handles food item orders

- On the orders page remove all timestamps, the only one that is important is when employees should begin processing an order

6.5 Conclusion

This chapter discussed the evaluation methods used for the project. In Section 6.2, the Earliest Deadline First algorithm was compared to the algorithm developed for this project from Section 4.4 on the basis of how they performed scheduling a set of tasks with respect to earliness. The scheduling algorithm developed for this project removed the earliness time with the assumptions that tasks take no longer to process than their C_i on the tasks used from Table 6.1. For the Earliest Deadline first algorithm the time at which tasks were scheduled didn't matter so long as they were scheduled before their deadline, whereas the scheduling algorithm developed for this project took into consideration when a task should be scheduled by reducing how early each task is scheduled. For NuPIC prediction accuracy there was an average accuracy of 87% for the predicted number of orders and an accuracy of 86% the predicted number of employees. Some feedback was also given on the system from Urbanity Cafe during a feedback session which highlighted the areas of the system they liked and the areas the system could also be improved.

Chapter 7

Conclusion

7.1 Overview

This chapter concludes the project by discussing some future work that could be done on the system, the learning outcomes and conclusion.

7.2 Future Work

7.2.1 More research into scheduling theory

Currently the scheduling algorithm developed schedules all orders in the system based on their release time, the earlier the release time the higher the priority according to the algorithm. More research into appropriately weighting the orders based on numerous factors such as cost and potential profit could yield better results in terms of revenue for businesses.

7.2.2 Deadline miss handling

Currently the system tries to advise employees of a business on which sequence to process the orders, the number of employees needed and utilisation status generation to aid with achieving all tasks by their deadline, however if an employees misses an order deadline the scheduling algorithm currently cannot recommend what to do apart from assuming an employee will finish as soon as possible. More research

into how other scheduling algorithms handle deadline misses might improve how the scheduling algorithm and task allocation could suggest what could be done for example by notifying a customer or by allowing the deadline to be missed as a customer has not yet arrived.

7.2.3 Allow customers to choose arrival time

Currently the system takes the arrival time of the customer by using the calculations Google Maps Distance Matrix by taking the customers location and the business location, however this method assumes that customers are going to travel to a business the moment they make the order which may not be the case every for every order that is made. If the iOS app could provide a mechanism for a customer to say exactly when they arrive then this could improve the accuracy the scheduled orders.

7.2.4 Longer prediction range

While NuPIC is integrated into the system, it currently only predicts one hour ahead for each of the predictions described in Section 4.6. NuPIC has the capability of generating predictions multiple hours ahead. By expanding the prediction range more insight into the number of employees needed and orders expected could be given. As well as expanding the prediction range NuPIC could also be used to predict the most popular types of products that will be ordered per hour.

7.2.5 Employee order completion time analysis

Once an employee finishes an order the completion time should be recorded and analysed. This analysis could include measuring the lateness or earliness with regards to the completion time of an order. For example if an employee is constantly missing deadlines then the system can send an alert mentioning that one employee keeps missing deadlines or alternatively if one employee keep completing orders very early then the system may send an alert advising them accordingly.

7.3 Learning outcomes

The learning outcomes of this project are extensive. By developing this system I have learned new areas within scheduling, data structures and open source technologies. I have been introduced into the area of real time systems and scheduling algorithms. Learning about the Earliest Deadline First algorithm and how it worked was extremely interesting and very valuable to learn for this project. By reading some of the literature on scheduling, it thought me new ways to express formulas and ideas such as using mathematical notations. By taking inspiration from the Earliest Deadline first algorithm and developing an alternative method to scheduling tasks, I have learned to seek out existing data structures which are used to solve problems in one domain and transfer it to another domain like scheduling, which was done, for example with the interval tree data structure. This project also thought me new technologies such as Node.js and AngularJS which had never been used before this project, which improved my knowledge on JavaScript. By using two different backend web frameworks - Node.js and Flask, it improved my understanding of HTTP, data serialisation and threading. My software management skills were also improved by modularising each component of the system into their own code base.

7.4 Conclusion

Overall the project could be labelled successful. All of the objectives set out in Section 1.2 have been reached and implemented. This project has used all of the knowledge I have gained over the previous four years during my studies in DIT. I have learned new technologies, data structures and algorithms. By undertaking this project I have grown as a student and as a computer scientist. I would like to continue to develop this project afterwards and implement the features discussed in future work in Section 7.2. This project has been a great experience an I am pleased with the outcome.

Bibliography

[1] Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to Algorithms. Third edition ed. MIT Press; 2009.

[2] Liu C, Layland J. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. Journal of the Association for Computing Machinery. 1973;p. 46–66.

[3] Stankovic J, Spuri M, Ramamritham K, Buttazzo GC. Deadline Scheduling for Real-Time Systems, EDF and Related Algorithms. Kluwer Academic Publishers; 1998.

[4] Top Order Management software;. [Accessed 23 November 2016]. Available from: `https://www.softwaresuggest.com/order-management-software`.

[5] Rosenheim. Food Tech and Media Industry;. [Accessed 23 November 2016]. Available from: `http://www.rosenheimadvisors.com/foodtechmedia`.

[6] Fortune. Starbucks Baristas Cant Keep Up With Mobile Orders;. [Accessed 3 March 2017]. Available from: `http://fortune.com/2017/01/27/starbucks-mobile-orders-slowdown/`.

[7] Google. Chrome V8 — Google's high performance, open source, JavaScript engine.;. [Accessed 24 November 2016]. Available from: `https://developers.google.com/v8/`.

[8] Express - Node.js web application framework;. [Accessed 4 November 2016]. Available from: `http://expressjs.com`.

[9] Web framework rankings;. [Accessed 6 November 2016]. Available from: `http://hotframeworks.com`.

[10] House C. Angular 2 versus React: There Will Be Blood;. [Accessed 3 November 2016]. Available from: https://medium.freecodecamp.com/angular-2-versus-react-there-will-be-blood-66595faafd51#.a60p5yo2w.

[11] Camp FC. The real reason to learn the MEAN Stack: Employability;. [Access 4 November 2016]. Available from: https://medium.freecodecamp.com/the-real-reason-to-learn-the-mean-stack-employability-29011ff6b2eb.

[12] Numenta. NuPIC — Numenta Platform for Intelligent Computing;. [Accessed 20 October 2016]. Available from: http://numenta.org/.

[13] Numenta. Hierarchical Temporal Memory MEMORY including HTM Cortical Learning Algorithms Version; 2011.

[14] Numenta. One Hot Gym;. [Accessed 21 October 2016]. Available from: https://github.com/numenta/nupic/tree/master/examples/opf/clients/hotgym/prediction/one_gym.

[15] Numenta. Swarming Algorithm;. [Accessed 3 March 2017]. Available from: https://github.com/numenta/nupic/wiki/Swarming-Algorithm.

[16] Halbert CL. interval tree 2.1.0: Python Package;. [Accessed 12 February 2017]. Available from: https://pypi.python.org/pypi/intervaltre.

[17] Prototyping Model in Software Engineering;. [Accessed 24 November 2016]. Available from: http://ecomputernotes.com/software-engineering/explain-prototyping-model.

[18] The MongoDB 3.4 Manual;. [Accessed 3 March 2016]. Available from: https://docs.mongodb.com/manual/.

[19] Heap queue algorithm;. [Accessed 4 February 2017]. Available from: https://docs.python.org/2/library/heapq.html.

[20] Fowler M. UnitTest;. [Accessed 16 November 2016]. Available from: http://martinfowler.com/bliki/UnitTest.html.

[21] Fowler M. Continuous Integration;. [Accessed 3 March 2017]. Available from: https://martinfowler.com/books/duvall.html.

Appendix A

A.1 Configuration Scripts

Overview

Throughout the development of the software components for this project, some aspects of configuration became repetitious, for example the configuration and storage of API keys, tokens or passwords which could not be checked into source control. These needed to be persistently stored and accessed from each of the software components that had dependencies on third party APIs such as Google Maps Distance Matrix or user names and passwords for connecting to a MongoDB instance. A Python script was created to solve this problem, as well as a Python class that could be used within the proactive module. The configuration script can be found in the QuickConfig directory.

Using the configuration script

The arguments for running the configuration script are listed below and can be list by running `python run.py -h`

- -mk, –make Makes configuration file in /etc directory.
- -m, –mongo [add, del] Add or delete a MongoDB details from config file
- –uri URI for new MongoDB database.
- -p, –password The password for the user.
- –port Port for new MongoDB database.

- -d, –db The MongoDB database name.
- -u, –username The username for the database.
- -g, –gmapskey [add, del] Add or delete Google Map API Key
- -s, –secret [add, del] Add or delete token secret for JSON Web Tokens.

The Configuration class handles all of the operations which can be executed from the configuration script. The Configuration class is also used in the proactive module for reading the MongoDB connection details and the Google Maps API key.

A.2 Building and running the components

To build and run each component, please refer to the README files located in each component, which gives details on how to build and run them.