Name: Jack Babey

Candidate Number: 8505

Centre Name: Wymondham College

Centre Number:

# Glossary

* [Analysis](#_Analysis)
  + [Product Definition](#_Product_Definition)
    - [Background](#_Background)

* + - [The problem](#_The_problem)
    - [Intentions](#_Intentions)
    - [The reason the solution is needed](#_The_reason_the)
    - [Next steps](#_Next_Steps)
  + [Stake Holders](#_Stakeholders)
  + [Research](#_Research)
    - [Interview](#_Interview)
      * [Questions](#_Questions)
      * [Small corner store owner](#_Small_corner_store)
      * [Theme Park](#_Theme_Park)
      * [Rapidly growing coffee shop](#_Rapidly_growing_Coffee)
    - [Underpinning knowledge and calculations](#_Underpinning_knowledge_and)
    - [Existing systems](#_Existing_Systems)
      * [Tillpoint](#_Tillpoint_[3])
      * [Lightspeed](#_Lightspeed_[4])
      * [EPoSnow](#_EPoSnow_[6])
      * [SumUp](#_SumUp_[10])
    - [Stakeholders and user needs](#_Stakeholder_and_user)
    - [My ideas and approach](#_My_Ideas_and)
    - [Essential features of the Solution](#_The_Essential_Features)
      * [Limitations](#_Limitations)
      * [Computational methods](#_Computational_Methods)
      * [Order of design and creation](#_Order_of_design)
      * [Hardware and software requirements](#_Hardware_and_software)
    - [Measurable success criteria](#_Measurable_Success_Criteria)
* [Design](#_Design)
  + [Branding](#_Branding)
  + [Breaking The Problem down](#_Breaking_The_Problem)
    - [Key for the structure diagrams](#_Key_for_the)
    - [Client Structure Diagram](#_Client_Structure_Diagram)
    - [Server Structure Diagram](#_Server_Structure_Diagram)
    - [Usability Features](#_Usability_Features)
  + [Defining the structure of the solution to be developed](#_Defining_the_structure)
    - [Interfaces](#_Interfaces)
      * [Login Screen](#_Login_Screen)
      * [Main Menu Screen](#_Main_Menu_Screen)
      * [Stock Lookup Screen](#_Stock_Lookup_Screen)
      * [Validation of inputs](#_Validation_of_inputs)
  + [Key classes, variables and Annotations](#_Key_classes,_variable)
    - [Database Request Format](#_Database_Request_Format)
    - [Server.cs](#_Server.cs)
      * [Modules and processes](#_Modules_and_processes)
      * [Pseudocode](#_Pseudocode)
      * [Flow Diagram](#_Flow_Diagram)
      * [Class diagrams](#_Class_diagrams)
      * [Variable Justification](#_Variable_Justification)
    - [Client.cs](#_Client.cs)
      * [Modules and processes](#_Modules_and_processes_1)
      * [Pseudocode](#_Pseudocode_1)
      * [Flow Diagram](#_Flow_Diagram_1)
      * [Class Diagram](#_Class_Diagram)
      * [Variable Justification](#_Variable_Justification_1)
    - [Encryption.cs](#_Encryption.cs)
      * [Modules and processes](#_Modules_and_processes_2)
      * [Pseudocode](#_Pseudocode_2)
      * [Class Diagram](#_Class_Diagram_1)
      * [Variable Justification](#_Variable_Justification_2)
    - [Item.cs](#_Item.cs)
      * [Modules and processes](#_Modules_and_processes_3)
      * [Pseudocode](#_Psuedocode)
      * [Class Diagram](#_Class_Diagram_2)
      * [Variable Justification](#_Variable_Justification_3)
    - [DatabaseManager.CS](#_DatabaseManager.CS)
      * [Modules and processes](#_Modules_and_processes_4)
      * [Pseudocode](#_Pseudocode_3)
      * [Class diagram](#_Class_diagram_3)
      * [Variable Justification](#_Variable_Justification_4)

# Analysis

## Product Definition

### Background

In everyday life everyone, no matter your age has been to a shop, be it a corner shop, a restaurant or a supermarket. All of these places sell you products. This means that they are constantly receiving stock, and selling products. This means that at times it is difficult for them to know how much of one item they have or even how much money they have made that day.

### The problem

Although modern technology is ever advancing this problem persists and many of the already existing solutions are either custom-made for a large chain, or simply not up to the standard that you should expect.

### Intentions

To assist store owners in keeping track of how many of a specific item they have left or how much money they have made, I intend to create an Electronic Point of Sale system. This system will be used to assist store owners and workers in taking orders and payments as well as keeping track of how much stock they have.

### The reason the solution is needed

It is much simpler for stores to use an electronic system to record their sales, this is because it can record and offer statistics that they don’t have time or resources to calculate by other methods. It also means that transactions will be more secure and better track can be kept of them. It also enables refunds or returns to be made simpler.

My program will allow a staff member of the store to create a new transaction, add items or products to this transaction and then complete the transaction. The transaction will then be stored in a database for easy access and information will be returned to a central server where a record of transactions will be stored. The item information will also be stored in a database on the central server.   
The client machine should be able to request information from the server, and the server will return it.

### Next Steps

The next step is to research the existing solutions that exist, understand their positives and negatives and understand how they interact with the staff member and the customer. As well as what other services or features they offer.

## Stakeholders

There are many potential stakeholders for my system, as it can be used in a variety of settings.

One potential stakeholder is the owner of a small corner store. They would benefit from using my system because it would allow them to keep track of their sales, profits, and transactions. This means that they would have more time, and ease of mind. They would need this system to be easy to use, and not take a lot of time or money to setup.

Another potential stakeholder is a theme park with various food and retail outlets. They would benefit from using my system because it would allow them to keep track of the most profitable outlets, their total sales and all the transactions. This would allow them to see the statistics of all the outlets, and will also allow them to have one centralised system, in which they know how much stock they have, which with multiple outlets can be difficult to keep track of. This system would be quick and easy to setup, meaning that they could quickly and efficiently deploy this software. This will be the primary case that I use for my development.

The last potential stakeholder I will be considering is a coffee shop that is rapidly expanding. They would benefit from using my system because it would allow them to track which items are the most popular, which aren’t performing very well and what they should therefore not sell. The system should be lightweight so they can implement it easily with low-spec machines. There should also be sufficient expansion available so that the system could be accessed remotely if a new coffee shop were to open.

All of the above stakeholders have requested that their company names remain confidential.

## Research

### Interview

#### Questions

* How will you practically use this program?
* What features do you think are essential for you to have in this program?
* What features would you like to have in this program in the future?
* How much would you consider paying for software of this nature?
* Do you have multiple sale points that require tills on your site?
* What sort of user interface would you like?
* What operating system does your current till system run on?
* What specific options would you like the user to have on the user interface?

#### Small corner store owner

* How will you practically use this program?  
  I will use this program to sell items in my shop and to keep track of how much money I have made each day.
* What features do you think are essential for you to have in this program?  
  The essential features are that the till enables me to total up an order quickly and effectively. It should also keep track of the total number of sales and the total money that I have made that day.
* What features would you like to have in the program in the future?  
  In the future, it would be useful to have a stock management system to allow me to keep track of which items are selling well each day. It would also be helpful if the till system had some way of helping me to take payments.
* How much would you consider paying for software of this nature?  
  I wouldn’t want to pay a monthly fee for this type of software. I would pay around £250 upfront as a maximum cost for the system. However, I wouldn’t mind paying for technical assistance, such as a setup fee and maintenance if it goes wrong.
* Do you have multiple sale points that require tills on your site?  
  I do not, I have only 1 till in my store.
* What sort of user interface would you like?  
  I would like an easy-to-use interface that is quick and easy to pick up. This means that I could continue my trade uninterrupted and just as fast as I am now.
* What operating system does your current till system run on?  
  My current till is a digital till which doesn’t have an operating system to speak of. If I was to have a computerised till I would like it to have a Windows [18] operating system as this is the one I am used to and it is easy to use.
* What specific options would you like the user to have on the user interface?  
  The user should be able to keep track of the order total and see which items have been added to the order.

From this interview, I took away that this system needs a simple interface that is easy to understand. The system should be compatible with the Windows operating system. A store owner should be able to use the system effectively with little to no training.

#### Theme Park

* How will you practically use this program?  
  We will use this program in our day-to-day operations to allow us to sell food retail items at different outlets across our park.
* What features do you think are essential for you to have in this program?  
  We believe that there should be the ability to sell items, and keep track of which items have been sold that day. All quickly and effectively so that we aren’t slowed down by the system when it is peak time. It should keep track of the performance of each outlet.
* What features would you like to have in the program in the future?  
  In the future, it would be useful if the system had a way of automatically printing tickets in a kitchen. It would also be useful to have a system that tracks trends over time, allowing us to plan our stock around the time of year.
* How much would you consider paying for software of this nature?  
  If the system is effective we are prepared to pay around a £300 upfront cost, with setup. And then an additional fee each month. Be this a flat rate or a percentage of our sales. The flat rate would be ideally around £15-£25 a month.
* Do you have multiple sale points that require tills on your site?  
  We have multiple outlets around the site, and therefore will need multiple tills that connect to one central system.
* What sort of user interface would you like?  
  We would require a modern user interface that is easy for our staff to use. Ideally, they would only require a small walk-through to be able to use the basic functions of the system. As with over 250 members of staff, it is impractical to have to deliver formal training on the till systems. Ideally, the User Interface would also be adaptable to the park’s astatic, however, this isn’t a necessity.
* What operating system does your current till system run on?  
  Our current till system runs on Windows [18] as it has good security features and is easy to use. It would be ideal for us if the system did run on Windows [18].
* What specific options would you like the user to have on the user interface?  
  There should be the option to add a name to the order as we have kitchens that make food and when serving it is easier to take a name for the order. There should also be a way of increasing the amount of an item that is in the order.

From this interview, I took away that the system should be easy to use, and have a networking aspect so that multiple till points can connect. I also think that an analytics feature would be nice to have, however it will take very long to implement.

#### Rapidly growing Coffee Shop

* How will you practically use this program?  
  We will use this program to make coffee and cake sales in my café. These will need to be taken quickly.
* What features do you think are essential for you to have in this program?  
  The essential features are quick processing, and sale and transaction management. These will make it easy to use and be able to do everything I need to do.
* What features would you like to have in the program in the future?  
  In the future, it would be useful to have a system that sends tickets to the kitchen when tickets are ordered. This would mean we could take orders quickly and keep track of them.
* How much would you consider paying for software of this nature?  
  We would consider paying around £200 up front and a small monthly fee. This would include first-time setup and telephone diagnostics. If it was required that an engineer came to the site we could understand an extra fee.
* Do you have multiple sale points that require tills on your site?  
  Currently, we have 2 points of sale in our café however, we are hoping future to open another café in. It would be ideal if the two sites could be interconnected on the system.
* What sort of user interface would you like?  
  We would like a modern interface that displays everything you need on one screen. This would make it easy to use. It would ideally require no instructions to use. However, this isn’t too big of a deal as we have a small team so would be easy to train them all.
* What operating system does your current till system run on?  
  Our current till system is an embedded system, which doesn’t include a digital operating system. However, if I were to upgrade to a digital system, I would like either a Linux [17] version or a Windows version. I would like Linux [17] because it is lightweight, the system could run slightly faster, and the staff members shouldn’t need access to anything other than the till software so usability isn’t an issue. Windows [18] would also work however I would prefer Linux [17]. We wouldn’t use macOS [16] as the systems themselves are too expensive for us to use.
* What specific options would you like the user to have on the user interface?  
  The user should be able to select which size of coffee the customer wants, as well as any extras and these are added to the order. A name or table number should be able to be added to the order so that we can deliver the customer’s order.

From this interview, I took away that the system should be easily expandable and should work across multiple locations. It should be compatible with Linux and have a Graphical User Interface that includes everything that the basic user might need on one screen. There should be a way to add a name to the order as well.

### Underpinning knowledge and calculations

Several different pieces of knowledge are needed to design this solution. The main pieces of knowledge required are:

* Databases  
  this includes a knowledge of how to create databases and tables, and how to manipulate that data inside the tables this includes Structured Query Language (SQL). How to manage a database for it to remain secure and normalised. For this example, I will be utilising SQLite [1] my reasoning behind this is explained later.
* Networking:  
  this includes a knowledge of how Local Area Networks (LANs) are setup and operated. In-depth knowledge of how ports, firewalls and connections work. This is so that we can connect the client and the server and allow them to communicate with each other. As well as Port Forwarding, Domain Naming Services and how they work and how to set up a Dynamic Domain Naming Service.
* Graphical User Interface (GUI):   
  this includes basic knowledge of Human-Computer Interaction (HCI) and different types of input methods that can be utilised. This is so that we can create an interface that the user can interact with and be able to access the system easily. This also allows us to create an interface that the user can use easily and without instructions.
* Object-Oriented Programming (OOP):  
  this includes being able to create a class and instantiate objects with that class type, as well as being able to link classes together using references, we may also need to be able to import libraries. This is so that we can effectively program an efficient solution. For this, we will be using C# as it suits our needs, with integration into Unity Engine [2]and with SQLite [1].
* Unity Engine [2]:   
  my solution requires a base level of knowledge of the Unity Engine {2}[4]. I am using unity as it allows me to easily manipulated and design complex Graphical User Interfaces (GUIs) as well as to visibly see what objects I have currently in the scene, which can help debug.
* Data Structures:   
  this includes Arrays, Dictionaries and Lists. This allows us to effectively retrieve data from the database, as well as being able to manipulate data.
* Co-routines/ IEnumerators:   
  this includes being able to program, start and use co-routines. As well as understand how they fit into the Unity Engine [2] System.
* Encryption  
  this includes the basics of encryption and decryption. As well as how to achieve this over a network. This means that when transmitting transaction details, or other important information the data is secure, even if it has been intercepted.

### Existing Systems

There are many different examples of exiting systems that are already available.

#### Tillpoint [3]

Tillpoint [3] is a versatile EPoS which has a variety of uses. It offers many different features, some of which go above the ‘Point of Sale’ title. The main ones include Point of Sale, Orders, Price Lists, Invoices, Discounts, Loyalty Points and Punch cards, Gift Cards, Offers, Inventory, Cash management, Balance Sheets, Profit and Loss along with expenses. They also offer Timesheet, Payroll and other Human Resource features.

Tillpoint [3] goes above the standard for EPoS systems with these features. The most relevant ones to my scenario are the Point of Sale, Price List, Discounts, Loyalty points and Punch cards, Gift cards, Offers, profit and loss, expenses and inventory. These are the features that I am looking at implementing.   
Tillpoint [3] offers its suite of products from $29 a month for their ‘standard’ retail package, however, this limits you to 3 staff, 1 register and 1 store\*. The closest one to the system that I am designing is their ‘advanced’ package which is $49 a month this allows unlimited users across 1 store\*.

Its Graphical User Interface has a modern layout with everything on one screen. They have images of the items they are selling. Their Point of Sale System has integration with mobile payment providers, such as PayPal [8], iZettle [9] and SumUp [10]\*

They offer a loyalty point program where the point earnings are set by the user. They have a gift card system where you can enter the gift card value, it is activated on the system and can then be used in-store with a set value. Their punch card system is for example you have a coffee, you can then gain ‘punches’ which can then give you a free coffee.

They also allow you to have demographics and statistics about your customers and sales, this is for example your frequent customers or the average age of your customers.

They allow you to manage your inventory, meaning you can keep track of the stock that you have.

Some screenshots of the website are below: 



I asked each of the potential stakeholders to look over the Graphical User Interface, and this is what they had to say:

Small corner store owner:

The user interface looks good, however, it has lots of features that I wouldn’t use, meaning I wouldn’t make full use of the features of the program.

Theme Park:

The user interface is modern and simple, however, most of our users won’t be using all the features, only our senior staff will use most of these features. We would make full use of the features, however, we wouldn’t want certain users accessing different parts of the system.

Rapidly expanding coffee shop:

The user interface is good looking, and won’t require much instruction to be able to use. We will be using most of the features in this system as we have a small team, and all of our staff can be given increased responsibilities.

#### Lightspeed [4]

Lightspeed [4] is a lightweight customisable EPoS system that can be used for various types of business.

Their stock management system allows you to categorise your items to make searching for them easier.

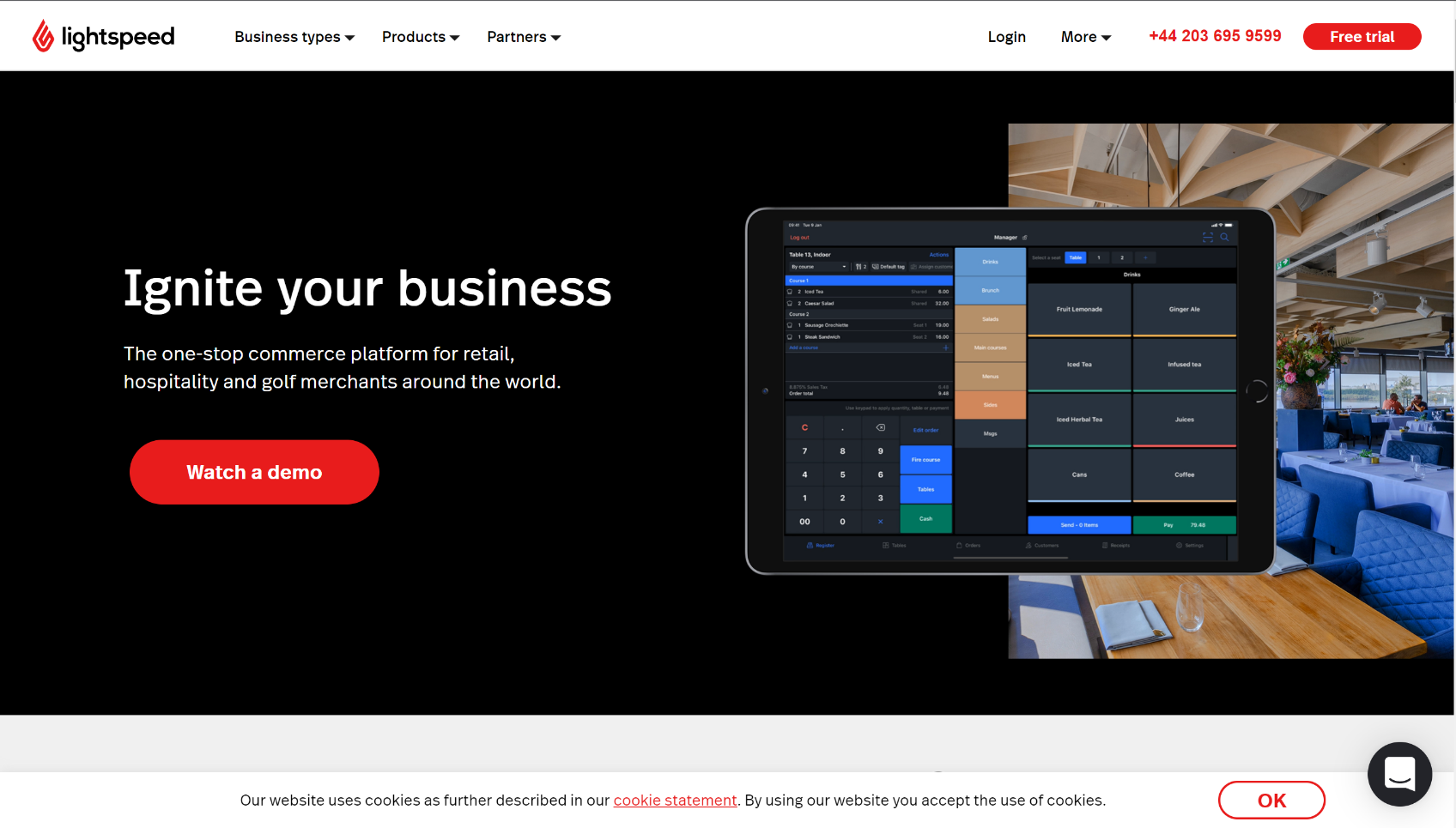
They have a modern Graphical User Interface, which is customisable depending on your business.

They offer insights and statistics into your sales, profits and locations of these.

The system is iPad [5] based meaning that it is lightweight and portable, this is useful for example in a restaurant. They offer card readers and printers to enable easy setup and ensure compatibility.

There is no price given for this, as it is a more custom solution.

I could use the insights and statistics that this offers as part of my solution.

Some screenshots of the website are shown below:

#### 

I asked each of the potential stakeholders to look over the Graphical User Interface, and this is what they had to say:

Small corner store owner:

I like the look of the user interface, with its simple layout and photos of the items. This would be helpful however this system also has lots of features that I won’t be using.

Theme park:

This system as a modern user interface, it looks easy to use, with examples of each of the products being used. This system looks like it has lots of useful features that can be used. It also has integrated printers and card readers which is useful to keep everything the same.

Rapidly expanding coffee shop:

The modern user interface is very appealing, with the ‘dark theme’ layout. It has lots of features and looks easy to use.

#### EPoSnow [6]

EPoSnow [6] offers full solutions for EPoS systems, they come with a till draw, computer, printer and card reader. This again ensures compatibility

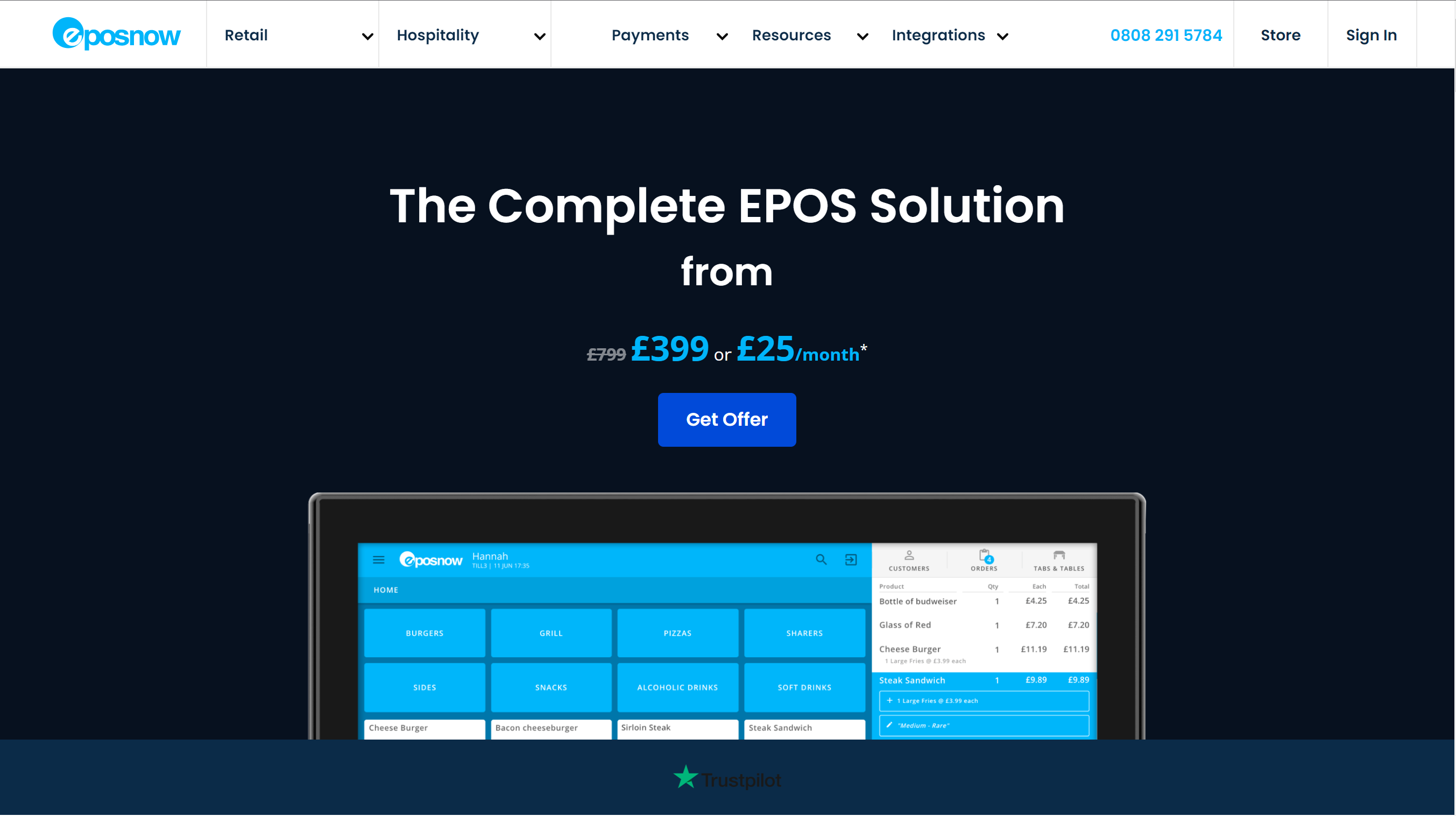
They offer a Point of Sale system with inventory management and the capability to have multiple stores. They also have E-commerce integrations.

They offer this for £25 a month\*. With this, you get access to all of their features.

They have a modern Graphical User Interface with a split layout. They also offer analytics with many different

They also offer a remote app for laptops and phones which allows you to access the system. They have integrations with WorldPay [7] and other mobile payment methods.

I could use the inventory management system in my software.

Some screenshots of the website are shown below: 



I asked each of the stakeholders what they thought of the graphical user interface of this system:

Small corner store owner:

This system looks modern with a colourful interface, however, this looks more toward restaurants rather than small stores. Looking at the website, the system has lots of useful features, but most of which I wouldn’t take advantage of.

Theme park:

The modern user interface and wide range of features make this a very suitable system. It has a bright and easy-to-use user interface.

Rapidly expanding coffee shop:

The layout of the system and the features are useful, with the table service and layouts, this makes it easy to keep track of which orders are completed and which are still left to do.

#### SumUp [10]

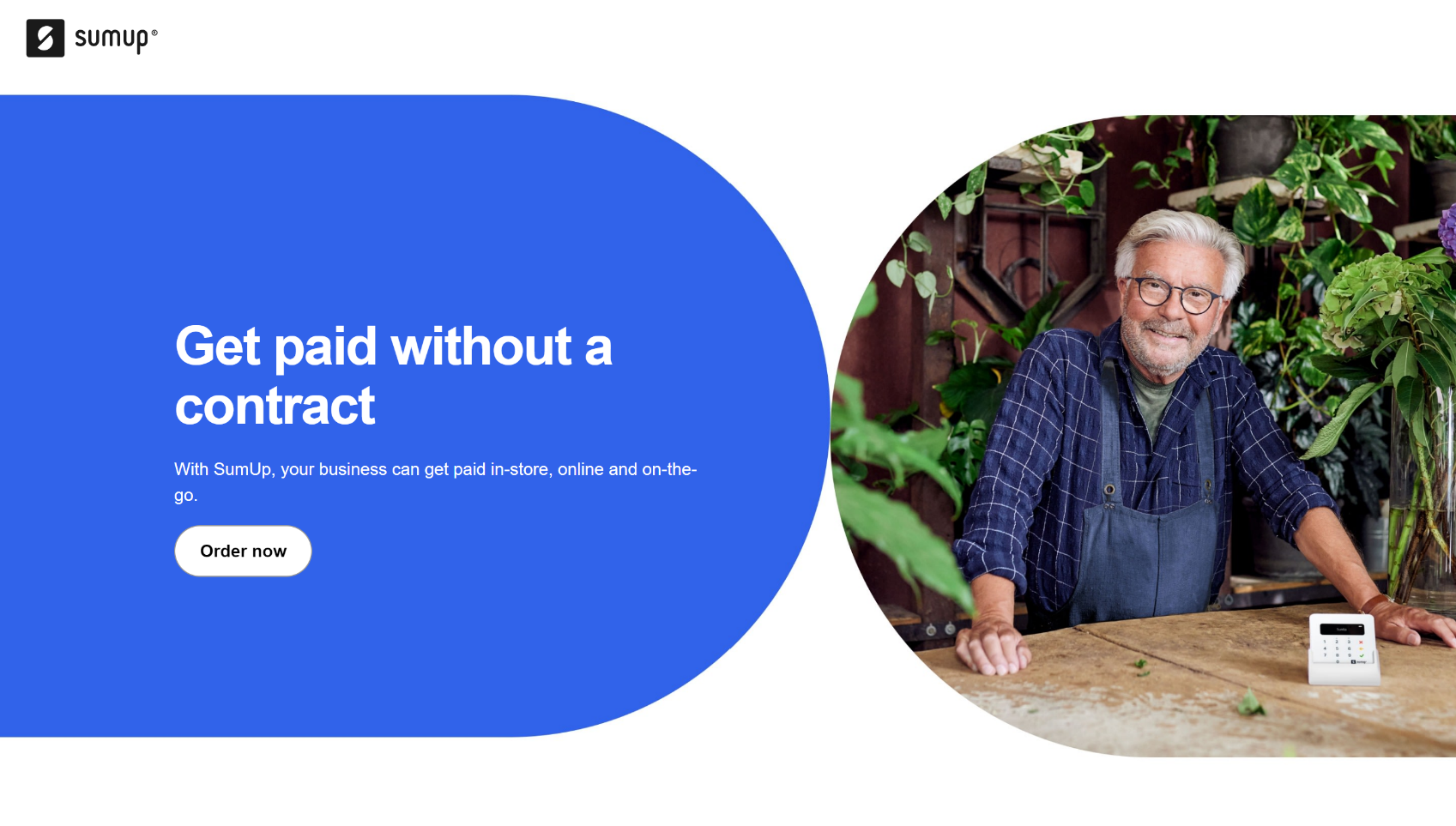
SumUp [10] is a portable system that offers card readers of full Point of Sale systems. It offers analytics and insights into the customer.

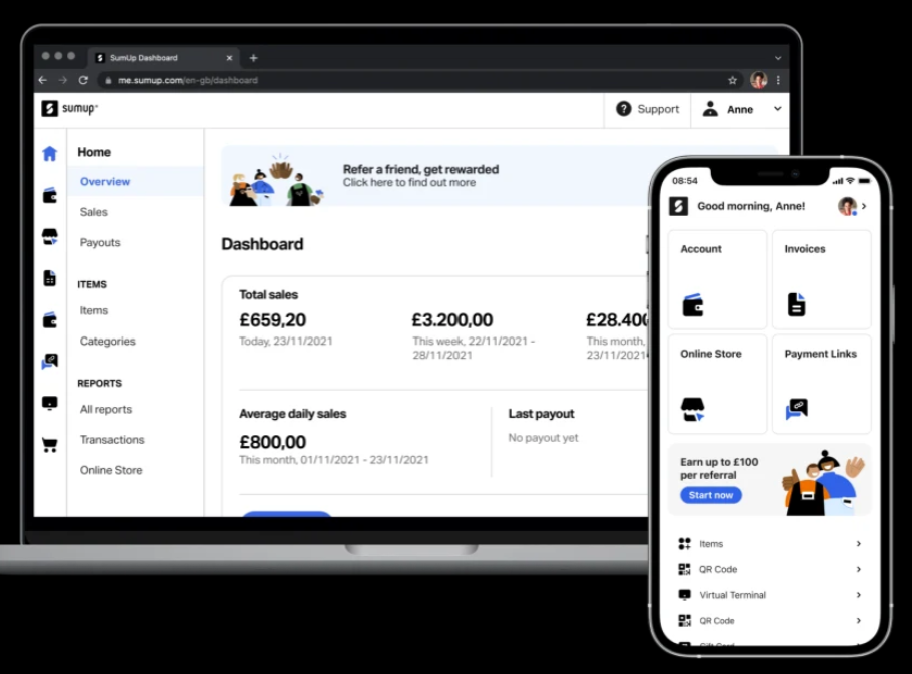
Their point-of-sale system comes with their card reader to ensure compatibility.

The card reader is portable and is designed for small businesses with no monthly fees. Just an initial payment.

They use a very modern Graphical User Interface. You can use it standalone or via an app on a mobile device. They charge a 1.69% transaction fee. Their basic card reader is £29\*

I can take inspiration from their modern GUI.

Below are some screenshots of the website:



Small corner store owner:

This system looks good, it has a simple user interface, with a mobile app to control the card reader, and this means that I wouldn’t have to buy any more equipment other than the card reader itself. However, this system doesn’t allow for cash payments, which around half of my trade is made through.

Theme park

This system looks simple to use, however it looks designed for more of a single point of sale, compared to having multiple outlets. However, I could be wrong about this. There is no way to take cash payments with this system, which means that we would lose some trade. We would have to buy devices to run this software, as we currently don’t have mobile devices as our payment points.

Rapidly expanding coffee shop

The modern user interface and simple layout are big advantages of this system. Not having a monthly charge is good, as some months we might not get as much custom as others. We don’t take many cash payments so this wouldn’t be much of a disadvantage for us.

\* All values and information are correct as of 30/06/2022

### Stakeholder and user needs

The main problem we trying to solve is a way to effectively take orders from customers, process payments and have a record of the transaction. All while making it quick and easy to use for the staff member operating the till.

The main stages of the process are:

1. The customer makes an order.
2. The staff member enters the order into the system.
3. The system checks the prices of the items against a database on the server.
4. The system checks if there are any restricted items in the order, such as alcohol or energy drinks that require extra validation.
5. The staff member takes payment.
6. The staff member completes the transaction on the program.
7. The transaction details are sent to the server and stored.
8. Other information is sent to the server, such as the number of each item bought and the profit that was made on the order and which sale point it was from

There will need to be various inputs, some of these are:

* Which items the customer would like to purchase.
* How much money the customer has given the staff member.
* Whether the customer is paying by cash or card.
* If the customer is over the age limit for restricted items and can the current staff member serve these items.

There will also be various outputs that will be required, some of these are:

* Which items are in the current order?
* How much does the current order cost?
* Has the transaction been authorised?
* Print a receipt.
* Print a ticket in the kitchen or coffee-making area if appropriate.

The Graphical User Interface should be modern looking with easily understandable navigation. The user shouldn’t need an operating manual for normal operation. It should conform to standard screen resolutions and shouldn’t be deformed at different aspect ratios or resolutions.

The user will interact with the product via a touch screen or mouse-based interface, this means that the orders can be taken quickly and efficiently. A keyboard will also be required for text inputs such as a manual cash amount input or a stock lookup.

The data that needs to be inputted is the user’s ID, to allow them to log into the system. Then the items that will be part of the order, should’ve been inputted into the server-side database before they are attempted to be bought. This will all be done via the Graphical User Interface.

The data that needs to be outputted is the cost of the order, the order name if applicable, items that were in the order, payment type and the name of the staff member that served them. All of this should be outputted ideally on a receipt for the user, however, this could also be temporarily outputted to the screen due to the limitations of this project.

The program should also be compatible with different operating systems however this isn’t the most important factor.

### My Ideas and approach

Ideally, this system would be completely commercially viable, but due to the limitations of the project, this won’t be possible. However, I can get it as close as I can to this with the resources available.

I would like the Graphical User Interface to be modern looking, well laid out and self-explanatory. This could be done by considering Human-Computer Interaction and how users will interact and the main functions they will use.

I would also like there to be certain features that are restricted to different levels of permissions, such as selling restricted items, discounts or editing items in the database. This could be done by using an attribute in the staff table which holds permission levels. `

There should also be a septate server-side client, which has a different layout as this won’t be used for selling items, it will only be used for checking and modifying the database, checking the state of the other clients and producing reports at the end of the commercial day. This will be implemented using a separate Graphic User Interface.

Ideally, I would also set this up so that it works with a compatible printer, so when a transaction is completed, a receipt would be printed. This would be done by using a Library that allows connections to printers and allows the program to send print requests. However, getting a working system is a higher priority.

### The Essential Features of the solution

There are a few features that are essential to the solution being successful. These are the baseline requirements that must be added to the system.

The main feature of the system is the Point of Sale. This is the main idea behind the system. This allows a staff member to create a transaction which can be completed with payment, and the transaction is recorded in a database.

Another main feature is the server connection. This is where each client connects with a central server which has a database. The server completes requests sent to it by each client, this could be a read request from the database or a write request to add a transaction to the current day’s table. This is featured in all of the research examples that I used and is also one of the features I wanted to include in my system. This server connection could also facilitate an inventory system, which keeps track of how many different types of items the store has.

The system should have a modern, clean Graphical User Interface that is easy to use. It should require no instructions for basic use. However, it should have an administrator feature with more advanced settings - these may require some instruction. Once again, this is a feature included in all of the research examples that I used.

There should be a system in place that allows you to calculate profits across the whole system and per individual client. The clients should be identifiable and nameable by the system so a company (stakeholder) can track employee performance over a period. This is another feature that I found was present in all of the software examples that I have used.

There should be a permissions system in place that restricts some users from accessing certain features such as the sale of restricted items (for example an underage employee can’t sell alcohol to a customer without the permission of a manager to adhere to current UK law.) or the modification of items in the database. This is a feature that is important to the stakeholder as they require that the system is in accordance with the law.

Some features we rejected and decided aren’t essential to the solution of the problem. These could be added later in the development period but should only be added after the main system is complete, and if the time constraint allows.

An example of this is the discount system, which is where if a certain combination of products is bought then the customer gets money off. This would also include a loyalty card system or a ‘punch card’ system. This would be complex to program and isn’t essential to the running of the program.

Another example is the payroll system. Most of the examples shown have integrated payroll systems. This is where the system calculates the staff member’s pay and is integrated into the Point of Sale System to make it a centralised hub to control your business. However, this is complicated to program and wouldn’t be essential to the program solving the problem.

#### Limitations

Several limiting factors could affect the development of the solution. The main ones are time and resource constraints.

We are only given a limited time to complete the project. However, a longer time frame would enable me to implement more features that I otherwise couldn’t. This may limit the number or effectiveness of features that are implemented into the system.

We also have limited resources to use for this system. In the context of my system, this means that I can’t have a card reader, for example, that I can integrate into my system, making it fully functional. Or a receipt printer, for testing the output of a receipt to a printer and how to format this. These provide limitations on the real-world effectiveness of my solution.

Along with this, there may be limitations on the design of the Graphical User Interface, due to the limitations on the assets that I can use. Again this is due to resource constraints as most ‘GUI Art Packs’ are paid for assets.

Compatibility might become an issue as I am creating a pure windows based system, which means that this system wouldn’t currently work on macOS or Linux. However, these features could easily be added later, thanks to Unity Engine [2]

#### Computational Methods

The features of the project that make it suitable to be solved by a computer are that, on a busy day, the number and nature of transactions that are made in a shop, are difficult to keep track of by a human. Especially if you have multiple stores or outlets like our primary stakeholder.

By using a computer system, you can easily keep track of transactions as well as be able to provide insights into the nature of the transactions, profits or other statistics that would be complex and take time to work out by humans.

It also makes it easier to keep track of the stock that we have, this means that we know how much of each product we have and can see the trends for this item. This provides the owner with greater insights into the products that they sell and how they can make their sales more efficient.

We will use concurrency to allow multiple different functions to run on the client system at once, this means that the client system won’t slow down the transaction process.

Abstraction means that we can take out unnecessary features. This will allow us to create a functional and efficient program in the time given.

Decomposition is one of the main principles of computational thinking. This allows us to break down a problem into smaller parts. These smaller part then may also be able to be broken down even further. This allows us to accurately say exactly what we are working on and what we need to do. Below I have used decomposition to break my problem down into different steps.

I will be using classes and object to create this program by using an Object-Oriented Programming Language. This means that we can create objects that can be instantiated allowing us to keep track of all the items in the current order.

#### Order of design and creation

I will be splitting the project into many sections, allowing me to focus on one part of the program at a time. These sections are shown below:

1. Server-Side Setup
   1. Creating the TCP server on the local Network
   2. Create a client class that will hold the attributes of each client that connects
   3. Create a function that allows the client to connect
   4. Creating a Listener to allow the server to receive requests
   5. Create a function that can send a signal to a client
   6. Create a function that requests the name of a client when the connect
   7. Create a function to receive a name from the client and set the name
2. Client-Side Setup
   1. Create a TCP connection to the server on the local network
   2. Create a function that sends requests to the server
   3. Create a function that receives information from the server
   4. Create a function to return the name of the client to the server when it is requested
   5. Create a function that saves client-specific data
   6. Create a function that allows the input of the IP and Port on the first-time setup
3. Database and stock table setup
   1. Import SQLite
   2. Create a database connection using SQLite libraries on the server-side
   3. Create a table in the database for the stock items if it doesn’t already exist
   4. Create a function to read data from the database and returns it
   5. Create a function to write data to the database
   6. Create a function to receive a database request
   7. Create a function that returns data from the database to the client
   8. Create a function to send database requests to the server
   9. Create a function that receives database requests from the server
4. Security
   1. Create an encryption class
   2. Create a function to encrypt data
   3. Create a function to decrypt data
   4. Apply this encryption to all outgoing communications
   5. Apply the decryption to all incoming communications
5. Graphical User Interface Setup
   1. Create a main scene for the GUI
   2. Create a main controller class for the scene
6. Stock Lookup
   1. Create a ‘stock lookup’ button
   2. Create a popup when the button is pressed
   3. Create a layout including a scroll area, 2 inputs and 2 buttons
   4. Allow a button in the popup to search the database, depending on the 2 inputs
   5. Create a grid layout which will store buttons
   6. Create a function to instantiate buttons for every returned value
   7. Delete the existing buttons for successive searches
   8. Allow each item button to return details about the item they represent in a popup
7. Order
   1. Create a scroll view area to hold the order
   2. Create a button prefab which will represent the item in the order
   3. Create a script on the button which will hold the attributes of each item
   4. Create a function to select a button
   5. Create a function to add items to the order from the stock lookup page
   6. Create a function to return the total of the order
   7. Create a button to delete the item in the order
   8. Create a button to increase the amount of an item in the order
   9. Create a button to decrease the amount of an item in the order
8. Main page setup
   1. Create a panel for item buttons to be stored on the main page with categories
   2. Create a dictionary of categories with the key as the category name and the value as a fixed-length array of items id’s
   3. Create a function to select which category is active
   4. Create a function that instantiates buttons which add items to the order using the item ids from the dictionary.
   5. Create a function that instantiates these buttons in the correct area and order
   6. Create a button that sends the user to a payment page
   7. Create a table that stores layouts\*
   8. Add the layout selection to the initial setup after the client is connected to the server\*
   9. Creates a function that requests the layout from the server\*
   10. Creates a function that returns the layout to the client\*
9. Payment page
   1. Create a transactions table
   2. Create a function that can receive transaction details from a client
   3. Create a function that can write transactions to the table
   4. Create a layout for the payment screen
   5. Create a function that allows you to pay a specific value
   6. Create a back button
   7. Create a button that completes the transaction
   8. Create a function that sends the transaction details to the server
   9. Create a pop-up that blocks the user from completing the transaction if conditions haven’t been met.
   10. Create a pop-up if restricted items are in order.
10. Admin Page
    1. Create a Graphical User Interface for the server-side
    2. Create an admin login prompt
    3. Create an interface that adds items to the stock table
    4. Create an interface that adds staff members to the staff table.
    5. Create an interface that modifies items in the stock table
    6. Create an interface that modifies staff members in the staff table
    7. Create a function that allows refunds to be processed on the client-side
11. Networking
    1. Create a DDNS name for the server
    2. Setting up port-forwarding
    3. Enable the DDNS name to interact with the server
    4. Make sure the DDNS will always have the correct IP for the server

#### Hardware and software requirements

For this software to be able to be used universally, the system and hardware requirements should be kept to a minimum, while still allowing the programs to run efficiently.   
For the software side;

In development, I will be using Unity Engine [2] Version 2021.3.5f1 [11] This is because this version of Unity Engine has all the features that we will require and is very stable. This will make development much quicker as it allows Graphical User Interface elements to be ‘dragged and dropped’ into the scene, this means that the creation of the GUI will be much quicker and look nicer in the end. Unity Engine [2] has the primary use case of making video games, be these 2D, 3D, AR or VR games. However, it is also used to develop software thanks to its powerful UI features.

As well as Unity Engine [2] I will be using Visual Studio 2022 Community Edition [12] This is because it has very good integration with Unity Engine [2] as well as a host of other development features that will help me in my development. Some of these useful features are as follows:

* Intelli-Sense  
  This is Visual Studio’s [12] code auto-completion, it is one of the best that I have found, not just because it prompts you to what method you would like to use for a specific class, but it can also suggest full lines of code. I have found this to be very useful and it helps to speed up the development process.
* Code Highlighting  
  While you program, Visual Studio [12] will colour the different parts of your code depending on their function, this makes it easy to read and understand what each part of the code is. It also highlights Syntax errors, allowing you to spot them easily. This speeds up the debugging process.
* Command-line interface  
  Visual Studio [12] offers a direct command-line interface, this means that if you need to test part of your code, or need to use to command line to, for example, push a version update to GitHub [13] you can do this easily.

The programming language that I have selected is C# this is because it has direct integration into the Unity Engine [2] and versions of all the additional libraries I need are compatible with the C# language. I have developed an in-depth knowledge of this programming language by using it alongside Unity [2]. It is object-oriented and therefore is suited to our needs.

I will be using the Libraries included with Unity [2] as well as SQLite [1] which is a lightweight database management library. I chose SQLite over other database management software for a variety of reasons. These are listed below:

* It has integration with C# making it easy to integrate with Unity Engine [2].
* It is lightweight so won’t take up much storage.
* It isn’t too powerful for our needs.
* It has a database viewer (DB Browser [14) which makes development simpler as I can view the database.
* Uses Structured Query Language (SQL) to control the database

I will be developing the software on Windows 10. However, as an optional feature, this could be developed for Linux. This is because Windows 10 is the most widespread Operating System and therefore this ensures compatibility with the most number of systems.

We will also be using NoIP [15] and their Dynamic Update Client. This ensures that when using NoIP as a Dynamic Domain Naming Service, our public IP is kept up-to-date so that other clients can connect from outside of the Local Area Network using an alphanumerical hostname.

The hardware requirements to run my solution will be kept low, to ensure compatibility. These will be a monitor, keyboard, mouse, secondary storage (SSD is recommended) and the computer itself. However, if this was a full commercial project, a receipt printer, card reader and till would also be required.

Along with these hardware requirements, a Local area network is required to allow the connection between the server and the clients. This doesn’t have to have a connection to the Internet, as this won’t affect the function of the system.

Along with the requirements, I would personally recommend that client machines have a minimum of 8GB of 2600 MHz RAM and a dual-core 2 GHz processor. The server should have at least 16GB of 26000 MHz RAM and a quad-core 3 GHz processor. This ensures that the server can handle enough clients for our system.

### Measurable Success Criteria

|  |  |  |
| --- | --- | --- |
| No. | Criteria | Justification |
| 1 | The client and the server must form a connection over a Local Area Network | This is to make sure that the client and the server can communicate with each other |
| 2 | The server must open a connection to the database | This is so that the server can complete requests that require data from the database |
| 3 | The user should be able to use a ‘stock lookup’ function | This allows the user to be able to access all of the stock items in the database |
| 4 | The user can log in with a unique identifier | This allows the system to know who has completed a transaction. |
| 5 | The user can add items to the order | This means that users can add items to a transaction |
| 6 | The user can increase the amount of an item in the order | This means that if the customer orders more than one of the same item, it can be grouped together |
| 7 | The system should display a correct order total | This is so the user and the customer know how much the current order costs |
| 8 | A prompt should appear for the user to add a name to the order if there are food items or drinks | This means that if receipts were being printed there would be a name for the order |
| 9 | A prompt should appear for the user if there are restricted items in the order | This means that the program is kept lawful for example under 16’s cannot buy energy drinks |
| 10 | The user should be able to prompt a payment screen when they are ready | This means that the user can take payment from the customer and enter it into the system |
| 11 | The payment screen should show how much money is left to pay, this should update as payments are added | This means that the user knows how much the customer has left to pay |
| 12 | The payment screen should show the order total | This is so the user knows how much the order costs in total |
| 13 | If more than the amount due is entered, the text should change and display the change due | This is so the user knows how much change they must give the customer |
| 14 | If the correct amount hasn’t been paid, the transaction cannot be completed | This is to ensure that the user doesn’t accidentally submit an order that hasn’t been paid for |
| 15 | If the correct amount has been paid, the transaction can be completed and is sent to the server | This is so that the transaction can be stored in case it is needed later down the line |
| 16 | There is a back button to cancel the payment | This is just in case a customer needs to add something else to their order |
| 17 | When a transaction is completed the number of each item bought decreases in the stock table | This is to make sure that we can keep track of how much of each item is left |
| 18 | An admin can add items to the stock table | This is so new products can be added |
| 19 | An admin can change items in the stock table | This is so that if a product’s price increases this can be added to the system |
| 20 | An admin can increase the amount of stock there is of an item | This is so that when a new delivery comes in, they can make sure the database is kept up-to-date |
| 21 | An admin can add staff members | This is so that when they hire new employees they can be added to the system |
| 22 | An admin can change staff member details | This is if a staff member changes their name or address for example |
| 23 | An admin can search for specific transactions | This is in case a false transaction goes through or a customer wants a refund |
| 24 | An admin can create a refund transaction | This is in case a customer wants their money back |
| 25 | This refund should also be sent to the transactions table in the database | This is so we can keep track of the refunds that have been given |
| 26 | The layout of the client’s screen should be governed by the server | This ensures that the layouts are the correct one each time and don’t have to be manually programmed into each client's machine |
| 27 | An admin should be able to modify the layout of a client’s screen | This means that if new products are added they can be added to the client’s screen for easy access |
| 28 | The server should be able to be connected using the DDNS hostname | This allows the server to be accessed from outside of the Local Area Network |
| 29 | The client and server should be stable | This means that they don’t produce syntax errors that stop the program from running |

# Design

## Graphical user interface Description automatically generatedBranding

One important thing when designing a piece of software is the name, this is what people will use to refer to your software. I have come up with a name for the software, this name is personal yet gets the idea of the software across.

This name also has domain names available for a website, these include those from .com, .co.uk and .tech among others.

## Breaking The Problem down

The problem can be broken down into multiple processes, as I have done above, but they can also be broken down into some basic functions, shown below are the ‘front end’ Structure diagrams, this shows what functions the user should be able to do using the user interface of the system. Note that this isn’t a definitive diagram of all the scripts and classes that will be used.

I have split the structure diagrams into Client and Server to represent the two different sides of the solution.

#### Key for the structure diagrams

Yellow is the name of the program

Green is the name of the main interface screens the user will see

The blue is the main modules associated with each screen

The orange is the processes and subroutines that make up each module

### Client Structure Diagram

Along with this are back-end scripts that control the connection to the server and database requests.

### Server Structure Diagram

Along with this, there are back-end scripts that control the databases and the server connections.

### Usability Features

To make sure that my project is easy to use and accessible to all ability levels. I will be making sure that both the User Interface and Documentation are easy to navigate and understand. The Documentation will include instructions on how to use the software, however, if the user interface is created correctly, this will be redundant.

For the user interface, I will pull on principles from Human-Computer Interaction (HCI) to make sure that the software is usable for everyone. I will primarily be developing this for use in Great Britain. This means that with a few tweaks it will be able to be used in the United States of America. But this will influence the language, styling, colours and layout that I use.

My process for deciding the User Interface features will be explained during the development stage as I implement the user interface features.

For documentation, I will be uploading and updating this project on GitHub [13] as I go this allows me to keep a change log of features as and when I add them, as well as good documentation features. GitHub [13] also can provide an MIT Copyright Licence to all projects uploaded to it, meaning that any proprietary code that I use in this project will be kept secure and protected through the Copyrights, Designs and Patents Act 1998 [19]. I will be creating this project as a repository, this is the basic type of project on GitHub [13].

## Defining the structure of the solution to be developed

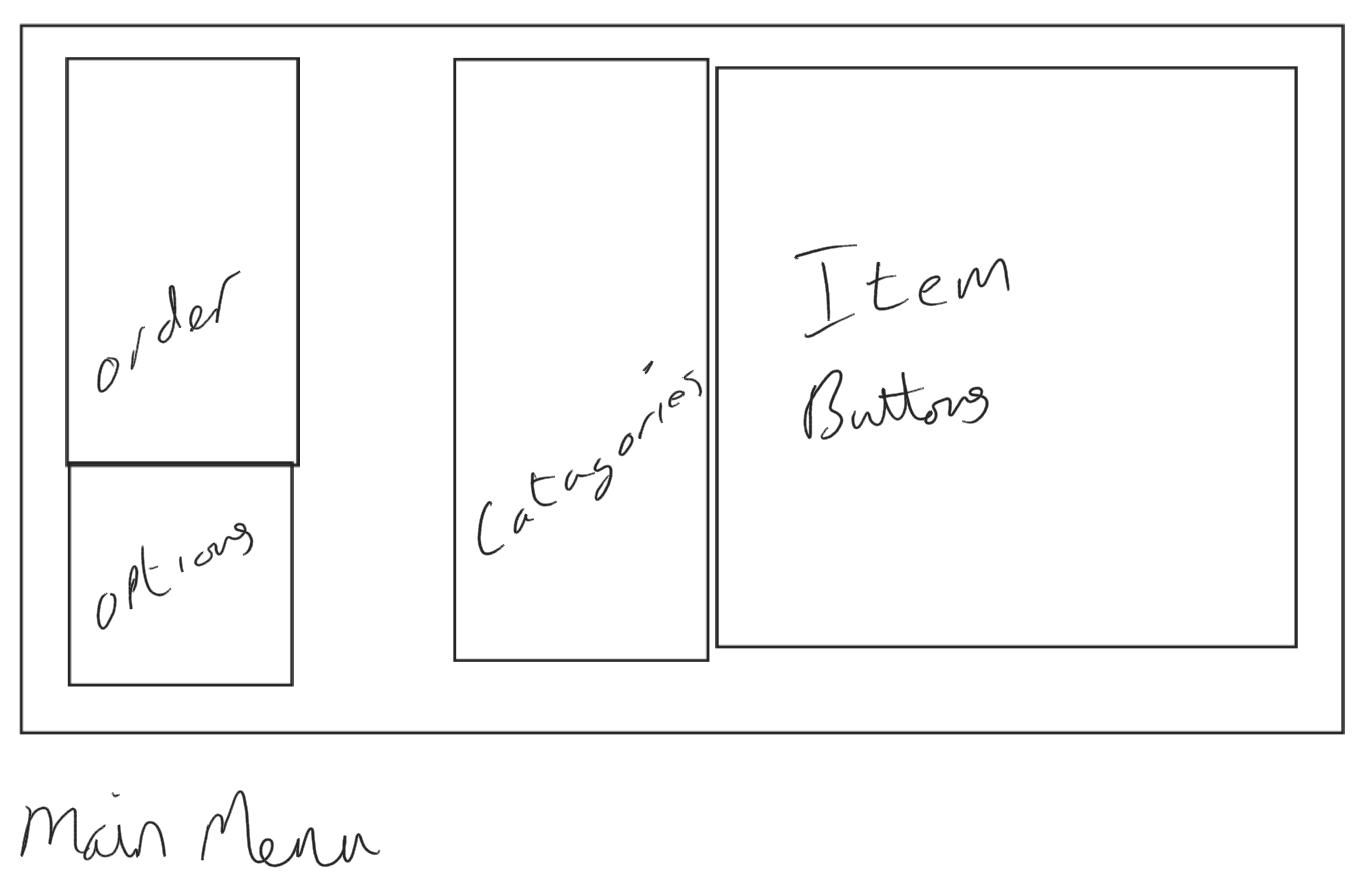
### Interfaces

Many different interfaces can be shown to the user. Therefore, the layout and design of the interfaces must be planned. This ensures that the interfaces are consistently looking and have all the required features, while also speeding up the development process.

#### Login Screen



#### Main Menu Screen

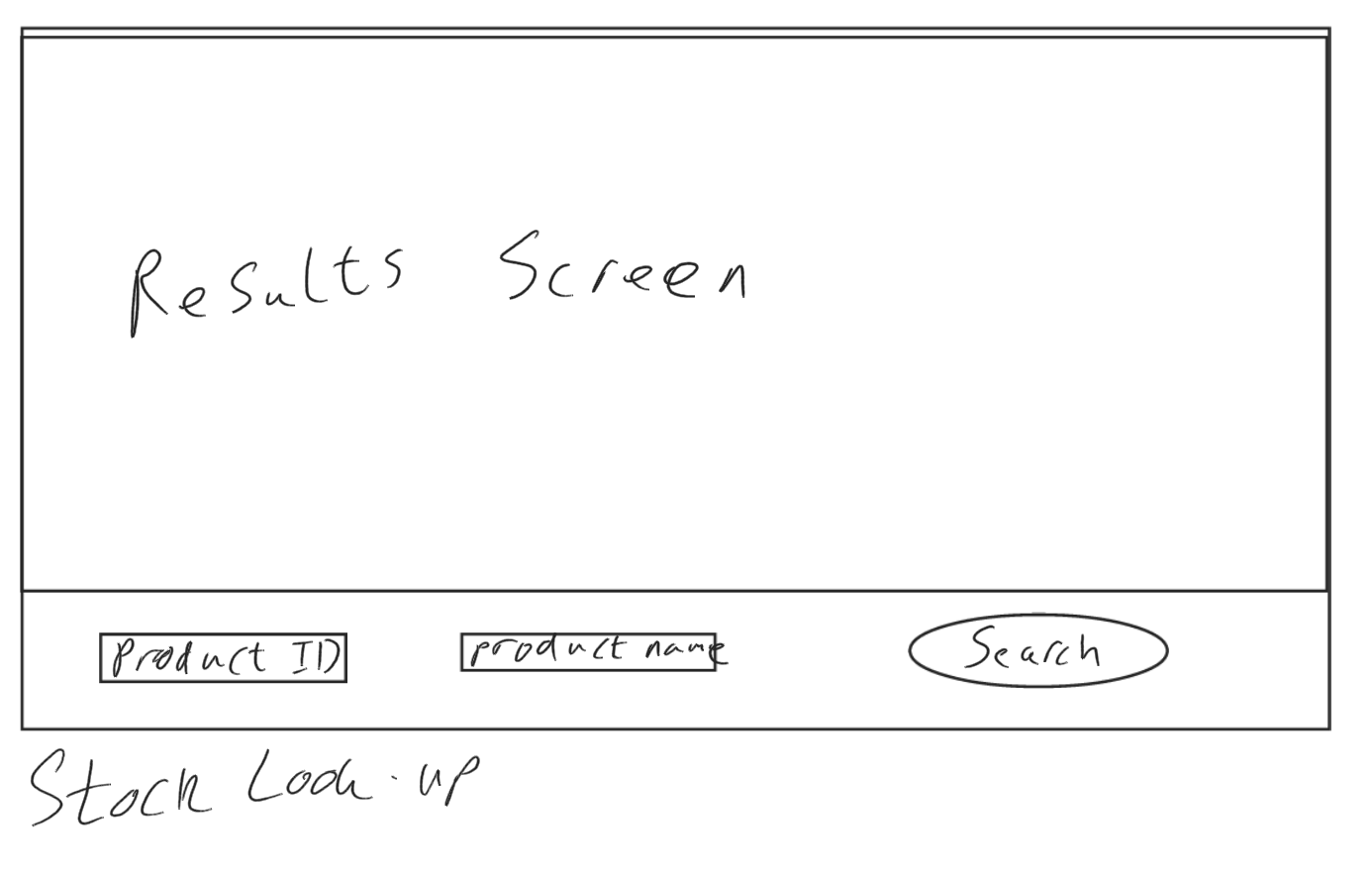


Item 2

Item 1

Order Item 1

#### Stock Lookup Screen



Item 2

Item 1

#### Validation of inputs

There are various types of validation that we can use to check that our inputs are correct. Some different types of validation checks are shown below:

|  |  |  |
| --- | --- | --- |
| Validation Check | How it works | Example |
| Format check | The data is the correct data type that follows an expected pattern. | If an integer is expected, then characters should be rejected. For example, a date should be given dd/mm/yyyy or that an @ appears in an email address. |
| Length check | The number of characters isn’t too long or too short | A password that needs to be eight characters or more. |
| Presence check | Data has been entered into a field. | A username has to be entered for the user to login. |
| Range check | Data must be within a valid range | The month cannot be less than one or more than twelve |
| Lookup table | Combo boxes and list boxed restrict the input to valid inputs | Choosing from a list of options. |
| Check digit/checksum | Checks the input is an acceptable entry before attempting to look it up in a database | Just because a username is valid doesn’t mean it exists. |
| SQL filtering | Checks that database requests are valid and in the correct format. Also checks that they don’t return every value in the database. | Someone enters “ OR 1 == 1” then it would return every value from the database |

We could also use data sanitisation to help secure sensitive inputs or to make sure the input is in the correct format. This helps our ease of use of the system.

Some of the inputs that will be used in my system are listed below, along with the validation checks that I will be using for them, to make sure that the user inputs the correct data.

|  |  |
| --- | --- |
| Input | Validation Check |
| String input | Format check, SQL filtering, Length Check, Check digit, Presence check |
| Integer input | Format check, Length check, Checksum, Presence Check |
| Date input | Format check, Presence check |
| Float input | Format check, Range check, Length check, Presence check |

## Key classes, variables and Annotations

When developing my software, some essential classes will need to be created to produce the basic function of the system. These include the scripts that control the server-client interaction, the client-side operations, encryption, stock lookup and order creation. Most of these scripts will be back-end scripts as these are the scripts essential to the programs’ function.

For each script or class, I will include a flow diagram and some pseudocode as well as some class structure diagrams

Most C# Scripts, when used with Unity Engine [2] have a start and an update procedure. The Start procedure is run once when the script is instantiated, or the program starts. The Update procedure (sometimes called the update loop) is run every time a new frame, or when the screen updates. This is usually around 60 times a second.

### Database Request Format

The formatting for the requests will be the following;

The type of data being sent to the server will be determined by the first section, each section is split using “|”. The types of data will also start with an ‘&’ to prevent incorrect requests from being sent.   
‘&NAME’ is the first type of data, this is when the client is sending the server its name.   
‘&STOCKLUDBR’ is when the client is requesting that a line of SQL is run against the stock database.   
‘&STAFFLOGDBR’ is when the client is requesting that a line of SQL is run against the staff database.   
‘%STOCKLURT’ is used when the server is returning the data from the previous stock request.   
‘%STAFFLOGRT’ is used when the server is returning the data from the previous staff request.   
‘&TRNASADD’ is when the client is requesting that a line of SQL is run against the transactions database, this line of code adds a transaction to the database.   
‘&TRANSDBR’ is when the client is requesting that a line of SQL is run against the transactions database and is expecting a value to be returned  
‘%TRANSRT’ is used when the server is returning data from the previous transaction request.

Below is a table of these requests and the type of data that should be sent with them.

|  |  |
| --- | --- |
| Request Type | Data to be sent |
| &NAME | String |
| &STOCKLUDBR | String |
| &STAFFDBR | String |
| %STOCKLURT | String |
| %STAFFRT | String |
| &TRANSADD | String |
| &TRANSDBR | String |
| %TRANSRT | String |

There is also a specific format for the type of transactions that can be made, these describe the method by which the client paid for their items. These codes are listed below:

|  |  |
| --- | --- |
| Code/Payment Type | Reason/Payment Method |
| 0 | Card Payment |
| 1 | Cash Payment |
| 2 | Part-Card, Part-Cash |

### Server.cs

#### Modules and processes

This is the script that controls the server-side of the TCP connection. It also handles the processing and returning of SQL requests. As well as handling new connections to the server.

If a new client is connecting the script creates a new object. It then requests the name of the client and sets this name to the object. This object is of a custom private class that will be made in this script, called client. Since the class is private to the script, it won’t get mixed in with another script that will be made otherwise.

The server should also be able to receive requests from clients. This will also be handled by this script, it should connect with the database management script to complete some of these requests. These requests will be made using a format that will be custom to the program. The client should be able to request data from the server about Staff members, Stock items, Transactions, Layouts and others. This will be done by splitting each message received by the server into sections, including type and data.

#### Pseudocode

The pseudocode I will write is all going to be inside a class structure, as is the standard for the C# programing language.

The only part out of the class structure is the ServerClient class, this is where all the information of each client will be stored.

The classes listed below are part of libraries built into Unity Engine [2] and C#:   
TcpListener, NetworkStream, DataWriter, TcpClient. I have included them in my Pseudocode as they are essential to the running of this program.

CLASS Server

port = 1803

server = new TcpListener

serverStarted = false

clients = []

disconnectList = []

PROCEDURE Start()

clients = NEW List OF ServerClient

disconnectList = NEW List OF ServerClient

server = NEW TcpListener

server.Start()

StartListening()

serverStarted = true

END PROCEDURE

PROCEDURE Update()

IF serverStarted == false THEN

RETURN None

END IF

FOR c in clients

IF c.IsConnected() == false THEN

c.Connection.close()

disconnectList.APPEND(c)

ELSE

NewtorkStream s = c.Connection.GetData()

IF s != None THEN

string data = s.ReadLine()

OnIncomingData(c, data)

END IF

NEXT c

FOR i = 0 TO disconnectList.Length()

clients.REMOVE(disconnectList[i])

disconnectList.POP(i)

next i

END PROCEDURE

PROCEDURE OnIncomingData(c, rawData)

data = Encryption.Decrypt(rawData)

IF data.CONTAINS("&NAME") THEN

c.clientName = data.SPLIT('|')[1]

ELSE IF data.CONTAINS("&STOCKLUDBR") THEN

DatabaseManager.StockLuDBR(c, data.SPLIT('|')[1], data.SPLIT('|')[2])

ELSE IF data.CONTAINS("&STOCKADD") THEN

DatabaseManager.StockAdd(data.SPLIT('|')[1], data.SPLIT('|')[2], data.SPLIT('|')[3], data.SPLIT('|')[4])

ELSE IF data.CONTAINS("&STAFFLOGDBR") THEN

DatabaseManager.StaffLogDBR(c, data.SPLIT('|')[1], data.SPLIT('|')[2])

ELSE IF data.CONTAINS("&TRANSADD") THEN

DatabaseManager.TransAdd(data.SPLIT('|')[1], data.SPLIT('|')[2], data.SPLIT('|')[3], data.SPLIT('|')[4], data.SPLIT('|')[5], data.SPLIT('|')[6], data.SPLIT('|')[7], data.SPLIT('|')[8])

ELSE IF data.CONTAINS("&TRANSDBR") THEN

DatabaseManager.transDBR(c, data.SPLIT('|')[1], data.SPLIT('|')[2], data.SPLIT('|')[3])

ELSE

OUTPUT (c.clientName + " has sent the following data: " + data)

END IF

END PROCEDURE

PROCEDURE StartListening()

server.BeginAcceptClient(AcceptTcpClient, server)

END PROCEDURE

FUNCTION IsConnected(c)

IF c != None AND c.Client != None AND c.Client.Connected == true THEN

IF c.Client.Poll(SelectMode.SelectRead) = true THEN

RETURN !(c.Client.Recieve(NEW BYTE[1]) == 0)

END IF

RETURN true

ELSE

RETURN false

END IF

END FUNCTION

PROCEDURE AcceptTcpClient(ar)

listener = TcpListener(ar).AsyncState

clients.Add(new ServerClient(listener.EndAcceptTcpClient(ar)))

StartListening()

Broadcast("%NAME", clients[clients.Length() -1])

END PROCEDURE

PROCEDURE Broadcast(data, cl)

dataToSend = Encryption.Encrypt(data)

writer = new DataWriter(cl.connection.GetData())

writer.WriteLIne(dataToSend)

writer.Flush()

END PROCEDURE

END CLASS

CLASS ServerClient

tcp = TcpClient

clientName = ""

PROCEDURE ServerClient(clientSocket)

clientName = "Unknown"

tcp = clientSocket

END PROCEDURE

END CLASS

#### Flow Diagram

Diagram

Description automatically generated

#### Class diagrams

|  |
| --- |
| Server |
| port : int server : TcpListener serverStarted : Boolean clients : List of ServerClients disconnectList : List of ServerClients |
| Start() Update() OnIncomingData(ServerClient, string) StartListening() IsConnected(TcpClient) AcceptTcpClient(IAsyncResult) Broadcast(string, ServerClient) |

|  |
| --- |
| ServerClient |
| tcp : TcpClient clientName : String |
| ServerClient(TcpClient) |

#### Variable Justification

Server Class:

|  |  |  |
| --- | --- | --- |
| Variable | Data Type | Purpose/Justification |
| port | Integer | This is the TCP port that the server will be able to be accessed, we want this to remain constant. |
| server | TcpListenter | This is how the server communicates and listens for data using the TCP and the port that was previously specified. |
| serverStarted | Boolean | This means that if the server connection hasn’t started then the script won’t be trying to listen for data. |
| clients | List of ServerClients | This is so the server can keep track of how many clients are connected and which client is which |
| disconnectList | List of ServerClients | This is so the server can keep track of which clients have disconnected so that it can remove them from the |

ServerClient Class:

|  |  |  |
| --- | --- | --- |
| Variable | Data Type | Purpose/Justification |
| tcp | TcpClient | This is to keep track of which client this object refers to. |
| clientName | String | This is to keep track of the name of the client this can be used for certain transactions. |

### Client.cs

#### Modules and processes

This script controls the client side of the TCP Connection. It also will control the SQL requests to the server and receive its response. As well as this, the script will handle connecting to the server.

If a new client program is opened, this script should handle the connection to the server. This is done by using a TCP connection. This connection is very specific as it only looks at a specific local IP address and port. The downside of this is that if the server moves IP or changes port then the clients won’t be able to find it. However, it does mean that the connection will be quick and simple.

If the client needs to request data from the server, this script also handles part of this process. The main use case will be requesting data from the server. The way this is handled is that another script creates a line of SQL to be sent to the server. This is then sent to the Client script which attaches a request type to the message and sends this to the server. The server will then respond with the data, which is passed back to the original script by this script.

#### Pseudocode

The pseudocode I will write is all going to be inside a class structure, as is the standard for the C# programing language.

Just like the Server script, the classes listed below are part of libraries built into Unity Engine [2] and C#:   
TcpListener, NetworkStream, StreamWriter, StreamReader, TcpClient. I have included them in my Pseudocode as they are essential to the running of this program.

clientName = ""

socketReady = false

socket = NEW TcpClient

writer = NEW StreamWriter

reader = NEW StreamReader

PROCEDURE ConnectToServer()

IF socketReady == true THEN

RETURN None

END IF

host = "192.168.1."

port = 1803

h = ClientDataSave.currentData.host

IF h != "" THEN

host = h

END IF

p = ClientDataSave.currentData.port

IF p != 0 THEN

port = p

END IF

socket = NEW TcpClient(host, port)

stream = socket.GetStream()

writer = NEW StreamWriter(stream)

reader = NEW StreamReader(stream)

socketReady = true;

END PROCEDURE

PROCEDURE Start()

ClientDataSave.LoadData()

ConnectToServer()

END PROCEDURE

PROCEDURE Update()

IF socketReady == true THEN

IF stream.DataAvailable == true THEN

data = reader.ReadLine()

IF data != None THEN

OnIncomingData(data)

END IF

END IF

END IF

END PROCEDURE

PROCEDURE OnIncomingData(rawData)

data = Encryption.Decrypt(rawData)

IF data == "%NAME" THEN

Send("&NAME |" + clientName)

RETURN None

ELSE IF data.Contains("%STOCKLURT") THEN

dataArray = data.split("|")

IF dataArray[1] == "~END" THEN

StockLookup.dataRecived = true

RETURN None

ELSE THEN

StockLookup.items.APPEND(NEW item)

itemNo = LENGHT(StockLookup.items) - 1

StockLookup.items[itemNo].id = INTEGER(dataArray[1])

StockLookup.items[itemNo].name = STRING(dataArray[2])

StockLookup.items[itemNo].price = FLOAT(dataArray[3])

StockLookup.items[itemNo].type = INTEGER(dataArray[4])

END IF

ELSE IF data.Contains("%STAFFLOGRT") THEN

dataArray = data.split("|")

IF dataArray[1] == "~END" THEN

StaffLogIn.dataRecived = true

RETURN None

ELSE THEN

StaffLogIn.staff.APPEND(NEW staffMember)

staffNo = LENGTH(StaffLogIn.staff) - 1

StaffLogIn.staff.id = INTEGER(dataArray[1])

StaffLogIn.staff.firstName = STRING(dataArray[2])

StaffLogIn.staff.lastName = STRING(dataArray[3])

StaffLogIn.staff.dob = DATETIME(dataArray[4])

StaffLogIn.staff.validDate = DATETIME(dataArray[5])

StaffLogIn.staff.startDate = DATETIME(dataArray[6])

StaffLogIn.staff.permsLv = INTEGER(dataArray[7])

END IF

ELSE IF data.Contains("%TRANSRT") THEN

dataArray = data.split("|")

IF dataArray[1] == "~END" THEN

TransLookup.dataRecived = true

RETURN None

ELSE THEN

FOR i IN RANGE 0 TO LENGTH(TransLookup.transactions) - 1

IF TransLookup.transactions[i].id == INTEGER dataArray[1] THEN

transNo = i

ELSE THEN

TransLookup.transactions.APPEND(NEW transaction)

transNo = LENGTH(TransLookup.transactions) - 1

TransLookup.transactions[transNo].id = INTEGER(dataArray[1])

TransLookup.transactions.date[transNo] = DATETIME(dataArray[2])

TransLookup.transactions[transNo].paymentType = INTEGER(dataArray[6])

TransLookup.transactions[transNo].transCost = FLOAT(dataArray[7])

TransLookup.transactions[transNo].serviceMemberId = INTEGER(dataArray[8])

TransLookup.transactions[transNo].itemsInTrans.APPEND(NEW TransItem)

transItemNo = LENGHT(TransLookup.transactions[transNo].itemsInTrans) - 1

TransLookup.transactions[transNo].itemsInTrans[transItemNo].itemId = INTEGER(dataArray[3])

TransLookup.transactions[transNo].itemsInTrans[transItemNo].quantity = INTEGER(dataArray[4])

TransLookup.transactions[transNo].itemsInTrans[transItemNo].cost = FLOAT(dataArray[5])

NEXT i

END IF

ELSE THEN

OUTPUT("Unknown Data recieved")

END IF

END PROCEDURE

PROCEDURE Send(rawData)

IF socketReady = false THEN

RETURN None

END IF

data = Encryption.Encrypt(rawData)

writer.WriteLine(data)

writer.Flush()

END PROCEDURE

PROCEDURE CloseSocket()

IF socketReady == false THEN

RETURN None

END IF

writer.Close()

reader.Close()

socket.Close()

socketReady = false

END PROCEDURE

#### Flow Diagram

#### Class Diagram

|  |
| --- |
| Client.cs |
| clientName : String  socketReady : Boolean  socket : TcpClient  writer : StreamWriter  reader : StreamReader |
| ConnectToServer()  Start()  Update()  OnIncommingData(string)  Send(string)  CloseSocket() |

#### Variable Justification

|  |  |  |
| --- | --- | --- |
| Variable | Data Type | Purpose/Justification |
| clientName | String | This is to make sure that the server can easily determine between different clients. This will remain constant. |
| socketReady | Boolean | This is to keep track if the client has connected to the server or not. |
| socket | TcpClient | This is to keep track of the IP and the port that the client is connected to. |
| writer | StreamWriter | This is to make sure that the client can send messages to the server. |
| reader | StreamReader | This is to make sure that the client can receive messages from the server. |

### Encryption.cs

#### Modules and processes

This script allows data to be encrypted before being sent over a network or stored. This ensures data security, and that people cannot access the data that is being sent, even if they intercept it.

This script will be found on both the server and the client and will have a standard key algorithm, which will make a new key every day which will be unique. As this algorithm will be built into the script, there is no need to send un-secure data to tell one side what their key is.

The alphabet that will be used will be the ASCII printable character set, this ensures that there are plenty of characters that can be used for the encryption. There are limitations to just using the ASCII character set, for example, there isn’t then the possibility of having accented characters, meaning that this software can only support a selected variety of languages. It could be a future development to include the Unicode character set instead.

I will be using a Vigenère Cipher, as this cypher is very secure. It uses index positions to be able to encrypt and decrypt the data. This is the most common way to achieve this encryption type.

You iterate through the ‘plain text’ (the input) and the key at the same time. Taking their index position in the alphabet, adding them together and finding the character at that index. This is your encrypted character. You do this for the whole of the plain text, if you reach the end of your key, this goes back to the beginning.

To decode you do the opposite of this. You iterate through the cypher text and the key at the same time, taking their index position in the alphabet, subtracting them this time and finding the character at that index.

The key for the encryption is based on the day. This is done by using the date followed by the day of the year, encrypted by itself. This ensures that it is secure, and no one can guess the encryption key.

I used the website dCode[20] for my research on how to implement this form of cypher.

#### Pseudocode

In this class, I have created a list variable for the alphabet we are using. I have written this out manually because this means that we have direct control over the different characters that can be used in our encryption. There is currently no way in C# to return a list of all printable ASCII characters that is more efficient than writing it out manually.

CLASS Encryption

currentDate = DateTime.minValue

key = ""

alphabet = ('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', '1', '2', '3', '4', '5', '6', '7', '8', '9', '0', '!', '"', '£', '$', '%', '^', '&', '\*', '(', ')', '\_', '-', '+', '=', '¬', '`', '|', ',', '<', '.', '>', '/', '?', ';', ':', '[', ']', '{', '}', '@', '#', '~')

PROCEDURE Update()

IF currentDate != DateTime.Now THEN

currentDate = DateTime.Now

SetKey()

END IF

END PROCEDURE

PROCEDURE SetKey()

key = STRING(currentDate.Date) + STRING(currentDate.DayOfYear)

key = Encrypt(key);

END PROCEDURE

PROCEDURE FindPosInArray(charToFind, array)

FOR i = 0 TO array.Length()

IF charToFind == array[i] THEN

RETURN i

END IF

NEXT i

RETURN -1

END PROCEDURE

PROCEDURE Encrypt(plainText)

cypherText = ""

FOR i = 0 TO plainText.Length()

keyCharNo = FindPosInArray(key[i % key.Length()], alphabet)

plainTextCharNo = FindPosInArray(plainText[i], alphabet)]

cypherNumber = keyCharNo + plainTextCharNo

cypherText += alphabet[cypherNumber % alphabet.Length]

NEXT i

RETURN cypherText

END PROCEDURE

PROCEDURE Decrypt(cypherText)

plainText = ""

FOR i = 0 TO cypherText.Length()

keyCharNo = FindPosInArray(key[i % key.Length()], alphabet)

cypherTextCharNo = FindPosInArray(cypherText[i], alphabet)]

plainNumber = keyCharNo + cypherTextCharNo

plainText += alphabet[plainNumber % alphabet.Length]

NEXT i

RETURN plainText

END PROCEDURE

END CLASS

#### Class Diagram

|  |
| --- |
| Encryption.cs |
| currentDate : DateTime  key : String  alphabet : Array |
| Update() SetKey()  FindPosInArray(char, array)  Encrypt(string)  Decrypt(string) |

#### Variable Justification

|  |  |  |
| --- | --- | --- |
| Variable | Data Type | Purpose/Justification |
| currentDate | DateTime | This is to keep track of the current date and time, so the encryption can be kept up to date, using this method increases efficiency. |
| key | String | This is to store the current key that is being used. |
| alphabet | Array | This is to store the alphabet that the encryption script will be using, this will remain constant. |

### Item.cs

#### Modules and processes

This script will contain a class which holds all the data required for an item in the stock database. This can then be used in various other scripts to store information about a specific item.

This allows the rest of the code I create to remain clean and tidy, with fewer variables in each script.

An example of its use will be when adding an item to the customer's order, the object for that item (an instance of this script) will be added to a list. This means that we can access all of the required data when we are completing our order, without having to make further requests to the server and database.

This is only a small script, however, it is quite important to the program and the way it will run

#### Pseudocode

CLASS Item

id = 0

name = ""

price = 0.1

type = 0

END CLASS

#### Class Diagram

|  |
| --- |
| Item.cs |
| id : Integer  name : String  price : Float |
| N/A |

#### Variable Justification

|  |  |  |
| --- | --- | --- |
| Variable | Data Type | Purpose/Justification |
| id | Integer | This is to store a unique identifier for each item, allowing this to be used as a reference in other scripts. This is also known as the barcode number for the item. |
| name | String | This is to store the name of the item so that it can be displayed or used. |
| price | Float | This is to store the price of the item so that we can create a subtotal of the items in the basket and can display the price. |

### DatabaseManager.CS

#### Modules and processes

This script allows the program to access and manipulate a database. This database will have various tables which will contain information that is vital to the running of the program. It allows the server to complete database requests that clients may send it.

Having a central database is useful in a case such as mine, this is because if any change is made to an item in the database, the clients will all receive this change. This means that we don’t have to change every client to have the new data. This increases the efficiency of the program.

We are achieving our database management by using a Third-Party Library called SQLite [1]. This is a light-weight library that is easy to use and allows us to use the database exactly as we would like. I chose SQLite [1] over other database management libraries because it doesn’t have unnecessary features that we won’t use. It also has easy integration into C# and Unity Engine [2] and doesn’t require the use of other programming languages.

This manages all of the tables that we will be using, this centralised system allows us to simplify the code by having classes that are then used to hold all of the information that is being brought to and returned from the database.

#### Pseudocode

At the start of this script we import the SQLite [1] library, there are many parts of this script that are from the SQLite [1] such as IMPORT SQLite

One of the tables isn’t included in the Pseudocode as it is very repetitive code and is essentially the same as anything shown here. Therefore, it wasn’t necessary to include it. This is for the layout\_table

IMPORT SQLite

CLASS DatabaseManager

IDbConnection dbcon;

PROCEDURE Start()

dbcon = OpenConnectionToDB()

CreateStockTable()

CreateStaffTable()

CreateTransTable()

END PROCEDURE

FUNCTION OpenConnectionToDB()

connection = "URI=file:" + Application.persistentDataPath + "/BPos\_Database.db"

stockDBConnection = new SqliteConnection(connection)

stockDBConnection.Open()

RETURN stockDBConnection

END FUNCTION

PROCEDURE CreateStockTable()

dbcmd = dbcon.CreateCommand()

q\_createTable = "CREATE TABLE IF NOT EXISTS stock\_table (id INTEGER PRIMARY KEY, name VARCHAR(255), cost FLOAT, type INTEGER);"

dbcmd.CommandText = q\_createTable

dbcmd.ExecuteReader()

END PROCEDURE

PROCEDURE InsertValuesIntoStockTable(itemToAdd)

cmnd = dbcon.CreateCommand()

cmnd.CommandText = "INSERT OR REPLACE INTO stock\_table (id, name, cost, type) VALUES(" + itemToAdd.id + ", '" + itemToAdd.itemName + "', " + itemToAdd.cost + ", " + itemName.type + ");"

cmnd.ExecuteNonQuery()

END PROCEDURE

FUNCTION ReadValuesInStockTable(id, itemName)

itemsReturned = []

cmnd\_read = dbcon.CreateCommand()

query = "SELECT \* FROM stock\_table WHERE id LIKE '%" + id + "%' AND name LIKE '%" + itemName + "%';"

reader = cmnd\_read.ExecuteReader()

WHILE reader.Read()

newItem = new item

newItem.id = System.Convert.ToInt32(reader[0])

newItem.name = System.Convert.ToString(reader[1])

newItem.cost = FLOAT(System.Convert.ToDouble(reader[2]))

newItem.type = System.Convert.ToInt32(reader[3])

itemsReturned += newItem

NEXT WHILE

RETURN itemsReturned

END FUNCTION

PROCEDURE CreateStaffTable()

dbcmd = dbcon.CreateCommand()

q\_createTable = "CREATE TABLE IF NOT EXISTS staff\_table (staffId INTEGER PRIMARY KEY, firstName VARCHAR(255), lastName VARCHAR(255), dateOfBirth DATE, validUntill DATE, startDate DATE, premissionsLv INTEGER);"

dbcmd.CommandText = q\_createTable

dbcmd.ExecuteReader()

END PROCEDURE

PROCEDURE InsertValuesIntoStaffTable(staffMemberToAdd)

formattedValidDate = "'" + staffMemberToAdd.validUntill.Year + "-" + staffMemberToAdd.validUntill.Month + "-" + staffMemberToAdd.validUntill.Day + "'"

formattedStartDate = "'" + staffMemberToAdd.startDate.Year + "-" + staffMemberToAdd.startDate.Month + "-" + staffMemberToAdd.startDate.Day + "'"

formattedDOB = "'" + staffMemberToAdd.dob.Year + "-" + staffMemberToAdd.dob.Month + "-" + staffMemberToAdd.dob.Day + "'"

cmnd = dbcon.CreateCommand()

cmnd.CommandText = "INSERT OR REPLACE INTO staff\_table VALUES (" + staffMemberToAdd.id + ", '" + staffMemberToAdd.firstName + "', '" + staffMemberToAdd.lastName + "', " + formattedDOB + ", " + formattedValidDate + ", " + formattedStartDate + ", " + staffMemberToAdd.permsLv + ");"

cmnd.ExecuteNonQuery()

END PROCEDURE

FUNCTION ReadValuesInStaffTable(id, lastName)

staffRetuned = []

cmnd\_read = dbcon.CreateCommand()

query = "SELECT \* FROM staff\_table WHERE staffId LIKE " + id + " AND lastName LIKE '%" + lastName + "%';"

cmnd\_read.CommandText = query

reader = cmnd\_read.ExecuteReader()

WHILE reader.Read()

newStaff = new StaffMember

newStaff.id = INTEGER(reader[0])

newStaff.firstName = STRING(reader[1])

newStaff.lastName = STRING(reader[2])

newStafff.dob = DATETIME(reader[3])

newStaff.validDate = DATETIME(reader[4])

newStaff.startDate = DATETIME(reader[5])

newStaff.permsLv = INTEGER(reader[6])

staffRetuned += newStaff

NEXT WHILE

RETURN staffRetuned

END FUNCTION

PROCEDURE CreateTransTable()

dbcmd = dbcon.CreateCommand()

q\_createTable = "CREATE TABLE IF NOT EXISTS trans\_table (transId INTEGER, date DATE, itemId INTEGER, quantity INTEGER, cost FLOAT, paymentType INTEGER, transCost FLOAT, serviceMemeberId INTEGER, PRIMARY KEY (id, itemId));"

dbcmd.CommandText = q\_createTable

dbcmd.ExecuteReader()

END PROCEDURE

PROCEDURE InsertValuesIntoTransTable(transactionToAdd)

formattedDate = "'" + transactionToAdd.date.Year + "-" + transactionToAdd.date.Month + "-" + transactionToAdd.date.Day + "'"

cmnd = dbcon.CreateCommand()

FOR i IN RANGE 0 TO transactionToAdd.itemsInTrans

cmnd.CommandText = INSERT OR REPLACE INTO trans\_table VALUES (" + transactionToAdd.id + ", " + formattedDate + ", " + transactionToAdd.itemsInTrans[i].id + ", " + transactionToAdd.itemAmounts[i] + ", " + transactionToAdd.itemsInTrans[i].cost + ", " + transactionToAdd.paymentType + ", " + transactionToAdd.totalCost + ", " + transactionToAdd.serviceMemeberId + ");"

cmnd.ExecuteNonQuery()

NEXT i

END PROCEDURE

FUNCTION ReadValuesInTransTable(transId, transDate, transTotalCost)

rtTrans = new transaction

formattedDate = "'" + transDate.Year + "-" + transDate.Month + "-" + transDate.Day + "'"

cmnd\_read.dbcon.CreateCommand()

query = "SELECT \* FROM trans\_table WHERE transId LIKE " + transId + " AND date LIKE " + formattedDate + " AND transCost LIKE " + transTotalCost + ";"

cmnd\_read.CommandText = query

reader = cmnd\_read.ExecuteReader()

WHILE reader.Read()

rtTrans.id = INTEGER(reader[0])

rtTrans.date = DATETIME(reader[1])

rtTrans.itemsInTrans.ADD(new transItem)

itemNo = LENGTH(rtTrans.itemsInTrans) - 1

rtTrans.itemsInTrans[itemNo].itemId = INTEGER(reader[2])

rtTrans.itemsInTrans[itemNo].quantity = INTEGER(reader[3])

rtTrans.itemsInTrans[itemNo].cost = FLOAT(reader[4])

rtTrans.paymentType = INTEGER(reader[5])

rtTrans.transCost = FLOAT(reader[6])

rtTrans.serviceMemeberId = INTEGER(reader[7])

NEXT WHILE

RETURN rtTrans

END FUNCTION

PROCEDURE StockLuDBR(client, idToFind, nameToFind)

endMessage = "%STOCKLURT|~END"

items = ReadValuesInStockTable(idToFind, nameToFind)

FOR i IN RANGE 0 TO LENGTH(items)

messageToSend = "%STOCKLURT|" + items[i].id + "|" + items[i].name + "|" + items[i].cost + "|" + items[i].type

Server.Broadcast(messageToSend, client)

NEXT i

Server.Broadcast(endMessage, client)

END PROCEDURE

PROCEDURE StockAdd(id, name, cost, type)

itemToAdd = NEW Item

itemToAdd.id = id

itemToAdd.name = name

itemToAdd.cost = cost

itemToAdd.type = type

InsertValuesIntoStockTable(itemToAdd)

END PROCEDURE

PROCEDURE StaffLogDBR(client, idToFind, lastNameToFind)

endMessage = "%STAFFLOGDBR|~END"

staffMembers = ReadValuesInStaffTable(idToFind, lastNameToFind)

FOR i IN RANGE 0 TO LENGTH(staffMembers)

messageToSend = "%STAFFLOGRT|" + staffMembers[i].id + "|" + staffMembers[i].firstName + "|" + staffMembers[i].lastName + "|" + staffMembers[i].validDate + "|" + staffMembers[i].startDate + "|" + staffMembers[i].permsLv

Server.Broadcast(messageToSend, client)

NEXT i

Server.Broadcast(endMessage, client)

END PROCEDURE

PROCEDURE TransAdd(id, date, itemId, itemAmount, cost, type, totalCost, serviceMember)

newTrans.id = id

newTrans.date = DATETIME(date)

newTrans.itemsInTrans.Append(new TransItem)

newTrans.itemsInTrans[0].itemId = itemId

newTrans.itemsInTrans[0].quantity = itemAmount

newTrans.itemsInTrans[0].cost = cost

newTrans.paymentType = type

newTrans.transCost = totalCost

newTrans.serviceMemeberId = serviceMember

InsertValuesIntoTransTable(newTrans)

END PROCEDURE

PROCEDURE TransDBR(client, id, date, totalCost)

endMessage = "%TRANSRT|~END"

transRq = ReadValuesInTransTable(id, date, totalCost)

FOR i IN RANGE 0 TO LENGTH(transRq) - 1

formattedDate = "'" + transRq.date.Year + "-" + transRq.date.Month + "-" + transRq.date.Day + "'"

FOR j IN RANGE 0 TO LENGTH(transRq[i].itemsInTrans) - 1

messageToSend = "%TRANSRT|" + transRq[i].id + "|" + formattedDate + "|" + transRq[i].itemsInTrans[j].itemId + "|" + transRq[i].itemsInTrans[j].quantity + "|" + transRq[i].itemsInTrans[j].cost + "|" + transRq[i].paymentType + "|" + transRq[i].transCost + "|" + transRq[i].serviceMemeberId

Server.Broadcast(messageToSend, client)

NEXT j

NEXT i

END PROCEDURE

PROCEDURE CloseConnectionToDB()

dbcon.Close()

END PROCEDURE

END CLASS

#### Class diagram

|  |
| --- |
| DatabaseManager.cs |
| dbcon : SQLite.IDbConnection |
| Start()  OpenConnectionToDB()  CreateStockTable()  InsertValuesIntoStockTable(item)  ReadValuesInStockTable(int, string)  CreateStaffTable()  InsertValuesIntoStaffTable(staffMember)  ReadValuesInStaffTable(int, string)  CreateTransTable()  InsertValuesIntoTransTable(transaction)  ReadValuesInTransTable(int, dateTime, float) |

#### Variable Justification

|  |  |  |
| --- | --- | --- |
| Variable | Data Type | Purpose/Justification |
| dbcon | SQLite.IDbConnection | This is the connection between the script and the database. This is required for the manipulation of the database using SQLite. |

### StockLookUp.cs

#### Modules and Processes

This script handles the user’s ability to look items up in the database. This will be on the client side and is a specific section, meaning that this code doesn’t need to be reused. All users should be able to access this feature so there will be no requirement for a permission level check.

This will handle the receiving of inputs from the user and then the sending of the request to the database. It will then also handle the response from the server back to the client.

#### Pseudocode

CLASS StockLookUp

items = []

dataRecieved = false

buttonPrefab = stockLuButton.PREFAB

PROCEDURE LookUp(name, id)

Client.Send("&STOCKLUDBR|id|name")

WAIT UNTIL dataRecieved == true THEN

FOR i IN RANGE 0 TO LENGTH(items)

AddStockButton(items[i].name, items[i].id, items[i].price, items[i].type)

NEXT i

END PROCEDURE

PROCEDURE AddStockButton(buttonText, buttonId, buttonCost, buttonType)

newButton = Instantiate(buttonPrefab)

newButton.text = buttonText

newButton.id = buttonId

newButton.itemName = buttonText

newButton.cost = buttonCost

newButton.type = buttonType

END PROCEDURE

PROCEDURE OnButtonPress()

idToSearch = INT(INPUT("Id to search")

nameToSearch = STRING(INPUT("Name to search")

Lookup(nameToSearch, idToSearch)

END PROCEDURE

PROCEDURE ClearStockPage()

items = []

DESTROY ALL buttonPrefab

END PROCEDURE

END CLASS

#### Class Diagram

|  |
| --- |
| StockLookUp.cs |
| item : Array dataRecieved : Boolean buttonPrefab : UnityEngine.GameObject |
| LookUp(string, int)  AddStockButton(string, int, float, int)  OnButtonPress()  ClearStockPage() |

#### Variable Justification

|  |  |  |
| --- | --- | --- |
| Variable | Type | Purpose/Justification |
| Item | Array | This is to store the items that are returned from the database. |
| dataRecieved | Boolean | This is to ensure that all the data has been received before the program continues. |
| buttonPrefab | UnityEngine.GameObject | This is the prefab of the buttons that will be used in the UI to display the items that have been returned. |

### StaffLogin.cs

#### Modules and Processes

This script is used for staff members to log in to the system. This ensures that anyone that isn’t authorised cannot log in and sell items to customers. It also means that it is possible to keep track of staff members’ statistics and check how many sales they make.

This script is the first one that the staff member using the system will interact with. Its purpose is to receive an id and to check to see if that is a staff member and to log them into the system by communicating their details to the POSController script.

#### Pseudocode

CLASS StaffLogin

staffMembers = []

dataRecieved = false

PROCEDURE LookUp(id)

Client.Send("&STAFFLOGDBR|" + id + "| ")

WAIT UNITL dataRecieved == true THEN

IF LENGHT(staffMembers) != 1 THEN

RETURN NONE

ELSE THEN

IF staffMembers[0].validDate > DATETIME.CurrentDate THEN

POSController.staffMember = staffMembers[0]

ELSE THEN

RETURN NONE

END IF

END IF

END PROCEDURE

END CLASS

#### Class Diagram

|  |
| --- |
| StaffLogin.cs |
| staffMembers : Array  dataRecieved : Boolean |
| Lookup(int) |

#### Variable Justification

|  |  |  |
| --- | --- | --- |
| Variable | Type | Purpose/Justification |
| staffMembers | Array | To store the data that is received from the server and database in response to the requests made. |
| dataRecieved | Boolean | To let the script know when they have received all the data required from the server database. |

### POSController.cs

#### Modules and Processes

This script is the main script that controls the Point-Of-Sale features, this includes storing the items that are currently in the basket.

Although this script is quite simple compared to the database and server-client scripts it is still very important and will control most of the program.

#### Pseudocode

CLASS POSController

tillName = "Till 1"

staffMember = NEW StaffMember

itemsInOrder = NEW DICTIONARY[item, quanitiy]

selectedItem = NEW Item

subTotal = 0.1

PROCEDURE AddItem(itemToAdd)

subTotal += itemToAdd.cost

FOR i IN RANGE 0 TO LENGTH(itemsInOrder) - 1

IF itemsInOrder[i].index.id == itemToAdd.id THEN

itemsInOrder[i].value += 1

RETURN NONE

NEXT i

itemsInOrder.APPEND(itemToAdd, 1)

END PROCEDURE

PROCEDURE IncreaseItem()

itemsInOrder(selectedItem).value += 1

END PROCEDURFE

PROCEDURE DecreaseItem()

itemsInOrder(selectedItem).value -= 1

IF itemsInOrder(selectedItem).value <= 0 THEN

RemoveItem(selectedItem)

END PROCEDURE

PROCEDURE RemoveItem(itemToRemove)

itemsInOrder.REMOVE(selectedItem)

selectedItem = NEW Item

END PROCEDURE

END CLASS

#### Class Diagram

|  |
| --- |
| POSController.cs |
| staffMember : StaffMember  itemsInOrder : Dictionary(Item, int)  selectedItem : Item  subtotal : float |
| AddItem(Item)  IncreaseItem()  DecreaseItem()  RemoveItem(Item) |

#### Variable Justification

|  |  |  |
| --- | --- | --- |
| Variable | Type | Purpose/Justification |
| staffMember | StaffMember | It stores the current staff member that is logged in. |
| itemsInOrder | Dictionary(Item, int) | This stores the items that are currently in the order. |
| selectedItem | Item | This is when an item gets selected by the staff member it is stored in this variable so that it can be adapted. |
| subTotal | float | This stores the current subtotal of the current order. |

### TransactionController.cs

#### Modules and Processes

This script handles the transaction process and sends the transaction to the server to store in the database. This script would also handle the card reader process and the cash drawer, however, these exceed my project’s scope but could be something added later.

Before this script is used, the amountToPay variable will be set to the subtotal amount from the POSController.cs

#### Pseudocode

CLASS TransactionController

amountToPay = 0.1

partPayedCash = false

transactionNumber = 0

PROCEDURE Start()

transactionNumber = ClientDataSave.currentData.numberOfTransactions

END PROCEDURE

PROCEDURE Payment(typeOfPayment)

IF typeOfPayment == 0 AND partPayedCash == false THEN

OUTPUT("CARD PAYMENT REQUESTED")

CompleteTransaction(0)

ELSE IF typeOfPayment == 0 AND partPayedCash == true THEN

OUTPUT("PART-CARD PAYMENT REQUESTED")

CompleteTransaction(2)

ELSE IF typeOfPayment == 1 THEN

OUTPUT("CASH PAYMENT REQUESTED")

CashPayment()

ELSE THEN

OUTPUT("UNKNOWN ERROR")

END IF

END PROCEDURE

PROCEDURE CashPayment()

cashAmount = INTEGER(INPUT("Enter Amount Of Cash")

amountToPay -= cashAmount

IF amountToPay > 0 THEN

partPayedCash = true

RequestPayment()

ELSE IF amountToPay <= 0 THEN

CompleteTransaction(1)

END IF

END PROCEDURE

PROCEDURE RequestPayment()

paymentType = INTEGER(INPUT("Please enter a type of payment")

Payment(paymentType)

END PROCEDURE

PROCEDURE CompleteTransaction(transactionType)

transList = []

transId = ""

FOR i IN RANGE 0 TO LENGTH(POSController.tillName)

transId += STRING(POSController.tillName[i].GetNumericValue())

NEXT i

transId += transactionNumber

FOR i IN RANGE 0 TO LENGTH(POSController.itemsInOrder)

newTransItem = NEW TransItem

newTransItem.itemId = POSController.itemsInOrder[i].index.id

newTransItem.quantity = POSController.itemsInOrder[i].value

newTransItem.cost = POSController.itemsInOrder[i].index.price

transList += newTransItem

formattedDate = "'" + DATETIME.current.Year + "-" + DATETIME.current.Month + "-" + DATETIME.current.Day + "'"

messageToSend = "&TRANSADD|" + transId + "|" + formattedDate + "|" + newTransItem.itemId + "|" + newTransItem.quantity + "|" + newTransItem.cost + "|" + transactionType + "|" + POSController.subTotal + "|" + POSController.staffMember.id

Client.Send(messageToSend)

NEXT i

END PROCEDURE

END CLASS

#### Class Diagram

|  |
| --- |
| TransactionController.cs |
| amountToPay : float  partPayedInCash : boolean  transactionNumber : int |
| Payment(int)  CashPayment()  RequestPayment()  CompleteTransaction(int) |

#### Variable Justification

|  |  |  |
| --- | --- | --- |
| Variable | Type | Purpose/Justification |
| amountToPay | Float | This is the amount left to pay on the transaction, this is used when part paying or overpaying with cash |
| partPayedInCash | Boolean | This is used to represent if the user has part paid using cash and card |
| transactionNumber | Int | This is a counter for the number of transactions, this is then used for the transaction id. |

### ClientDataSave.cs

#### Modules and Processes

This script is used to save the client’s data that it will need to reconnect to the server each time. This is used to create a .json file. It will store the port, host IP and other important information that is required for the program to run. This means that this data will be able to be adjusted if needed and can be set for each unique network.

#### Pseudocode

IMPORT JsonUtility

CLASS ClientDataSave

saveLocation = ""

currentData = NEW ClientData

PROCEDURE LoadData()

startData = JsonUtility.FromJson< ClientData >(File.ReadAllText(saveLocation ))

currentData.host = startData.host

currentData.port = startData.port

currentData.numberOfTransactions = startData.numberOfTransactions

END PROCEDURE

PROCEDURE SaveData()

currentData.numberOfTransactions = TransactionController.transactionNumber

jsonData = JsonUtility.ToJson(currentData, true)

File.WriteAllText(saveLocation, jsonData)

END PROCEDURE

END CLASS

CLASS ClientData

host = ""

port = 0

numberOfTransactions = 0

END CLASS

#### Class Diagram

|  |
| --- |
| ClientDataSave.cs |
| saveLocation : string  currentData : ClientData |
| LoadData()  SaveData() |

#### Variable Justification

|  |  |  |
| --- | --- | --- |
| Variable | Type | Purpose/Justification |
| saveLocation | String | This is the location where the .json file will be stored. This also includes the files name and extension. |
| currentData | ClientData | This is a class which stores the client’s data ready to be stored or read from. |

### Flies

This section explains the files that this program will create, store and use. These include the files which contain information for starting up the server-client connection and the formatting of the database.

#### Client Starting Data (JSON file)

This JSON file will store the data that is used by each client as they start up, this means that they can connect to the server on the network easily. The networking method we are using requires a direct connection, this means that you need to explicitly specify the IP address (host) and the Port that you are hosting the server on.

Because of this, we need to have an efficient way of storing these pieces of information, but in a way that we can easily change if we move the server.

This file will also contain the number of Transactions and the name of the client, these are important because the number of transactions is required to ensure that we don’t name the transactions the same name, causing the database to become compromised. And the name is required so that we have a unique identifier when working server-side and to keep the transaction names standardised.

The format of the file will be in-line with the standard practice for JSON files. An example of what this file could look like is shown below

{

"clientName" : "Till 1",

"host" : "192.168.1.2",

"port" : 1234,

"numberOfTransactions" : 0

}

#### Main Database (db file)

This file will be the main database file for our system. It will store the items that are in stock and can be sold, the members of staff that have/are/will work at our store, and the transactions that have been made on our system. This database stores raw data, meaning that it is unencrypted. This means that the system it is on should have up-to-date Malware detection software, a robust password and be disk encrypted. This means that despite the raw data being unencrypted, there is very little chance of the data being stolen.

However, for more security, we could encrypt the database, but this makes database management more difficult, as we would have to create our own viewer with an individual encryption key on it. And this viewer may not have some of the features that other database management software might have.

In our database there are currently 4 tables, however, this could change as more features are added in the future. These tables are listed below, with their attributes and their data types.

|  |
| --- |
| stock\_table |
| id : Integer : Primary Key  name : String  cost : Float  type : Integer |

|  |
| --- |
| staff\_table |
| staffId : Integer : Primary Key  firstName : String  lastName : String  dateOfBirth : Date  validUnitll : Date  startDate : Date  permissionsLv : Integer |

|  |
| --- |
| trans\_table |
| transId : Integer : Primary Key  date : Date  itemId : Integer : Primary and Foreign Key  quantity : Integer  cost : Float  paymentType : Integer  transCost : Float  serviceMemberId : Integer : Foreign Key |

|  |
| --- |
| layout\_table |
| layoutId : Integer : Primary Key  catName : String : Primary Key  itemId : Integer : Primary and Foreign Key  catType : Integer |

While working with databases it is important to follow certain rules to maintain the database correctly and efficiently, and in a way in which the database will always work and be correct. Normalisation is a technique which helps to reduce data duplication and enforce data integrity. There are 3 stages of Normalisation, called Normal Forms. These are listed below:

1st Normal Form

* Each Record has a primary key
* Data is atomic
* No repeating groups of attributes

2nd Normal Form

* No Partial Dependencies

3rd Normal Form

* No non-key dependencies

While working with my database I will ensure that I am following these principles as well as I can.

One of the items in the database that could cause an issue is the ‘cost’ attribute in trans\_table, I included this so if prices change after a transaction, there is a record of how much the item cost when the transaction was made.

While making requests to the database we will also ensure referential integrity and that transaction processing is handled correctly. For this, we can use The ACID test. This test is commonly used when working with databases to ensure that transactions are being completed correctly. The ACID test is as follows

**Atomicity** means that all the components of a transaction are atomic. This means that the transaction must succeed or fail.

**Consistency** means that illegal or incomplete transactions are rejected so the integrity of the database is maintained.

**Isolation** means that each transaction doesn’t affect any other transaction

**Durability** means that data is saved once the transaction has been completed. This should remain true even if there was a hardware failure immediately after the transaction.

Most of these features are built into SQLite [1] meaning that it is easier to uphold these principles.

Another issue with databases over networks is multiple people accessing or modifying the same record at the same time. For our purposes, this shouldn’t be an issue. For the general use case, records in the database won’t be modified. This means that we won’t have the issue of multiple people writing to the same record at the same time. However, if we were to allow most users to modify records in the database I would implement the method of record locking. This would work as only one record would be modified at a time, so deadlocking wouldn’t occur.

In our database management script we use the INSERT command rather than the SET TRANSACTION command this is to simplify the programming process and as only one record is being modified at a time, the SET TRANSACTION command is unnecessary.

## Identifying test data to be used during iterative development

During our development process, there will be various points at which we should test that our code works how we expect. This means that we can evaluate what works and what doesn’t and make adjustments to our program. Below will be a set of milestones that I will reach during my development and the tests that should be carried out during this process. These could be adapted and changed as we add more features.

### Milestone Tests

#### Milestone 1

Milestone 1 is after we have set up the client and the server. (At the end of the 2nd section of programming)

|  |  |  |  |
| --- | --- | --- | --- |
| Test Number | What is being tested | Inputs | Expected Outputs |
| 1.1 | Server opening a connection on a given port if the port isn’t used by other functions such as HTTPS | 1803, 6421, 8888, 26656 | Open, Open, Failed to Create Connection, Open |
| 1.2 | The client can connect to a server, given a correct IP address and port, and will return an error when these details are incorrect | Correct IP and Correct Port, Correct IP and Wrong Port, Wrong IP and Correct Port, Wrong IP and Wrong Port. | Connected, Error, Error, Error |
| 1.3 | The Server can broadcast messages to a specific client | “&NAME” | Message Received By Client |
| 1.4 | The Client can send messages to the server | “&NAME|TestTill” | Message received by Server |
| 1.5 | The server sets the client name on the object for the correct client | “TestTill” | Client name correctly set Server Side |

#### Milestone 2

Milestone 2 is after the database and the stock table have been created.

|  |  |  |  |
| --- | --- | --- | --- |
| Test Number | What is being tested | Inputs | Expected Outputs |
| 2.1 | Connection to the database is being formed correctly | Correct File path and file name, incorrect file path and name | A connection to the database gets opened, new database file gets created |
| 2.2 | A stock table is created if it doesn’t exist | No Stocktable, a stock table with elements | A stock table is created, the stock table remains the same with the elements in it |
| 2.3 | Can read items from the stock table and are returned as an array of items | A valid item id, an invalid item id, part of a valid id | Returns an array of items with a single element, returns no elements into the array, returns no elements into the array. |
| 2.3 | Can add items to the stock table | A valid item, an invalid item | Item is added to the table in the correct format, the item isn’t added to the table. |
| 2.4 | Can read items from the stock table using client requests and return the data | A valid item, an invalid item | Item is returned correctly to the client, no item is returned to the client. |

#### Milestone 3

Milestone 3 is after the encryption class has been created.

|  |  |  |  |
| --- | --- | --- | --- |
| Test Number | What is being tested | Inputs | Expected Outputs |
| 3.1 | Data can be encrypted | Some data to be encrypted inside the alphabet, and some data to be encrypted with characters outside the alphabet | Some encrypted data with every character changed, and some encrypted data with some characters not changed. |
| 3.2 | Data can be decrypted | Some encrypted data is to be decrypted inside the alphabet, and some encrypted data is to be decrypted with some character outside the alphabet. | Some raw data with every character changed, and some raw data with some characters not changed. |

#### Milestone 4

Milestone 4 is after the stock lookup page has been created.

|  |  |  |  |
| --- | --- | --- | --- |
| Test Number | What is being tested | Inputs | Expected Outputs |
| 4.1 | The stock lookup page opens when the UI button is pressed | Button input to the UI Button | Stock Lookup page opens |
| 4.2 | A server request is made with custom inputs | {a valid id, a valid name}, {a valid id, no name}, {no id, valid name}, {no id, part of a name}, {no id, no name} | The database receives the request |
| 4.3 | The server returns the data to the client | {a valid id, a valid name}, {a valid id, no name}, {no id, valid name}, {no id, part of a name}, {no id, no name} | A single item, a single item, a single item, multiple items, every item in the table. |
| 4.4 | UI buttons are created for each stock item returned | {a valid id, a valid name}, {a valid id, no name}, {no id, valid name}, {no id, part of a name}, {no id, no name} | A single button, a single button, multiple buttons, a button for every item in the table. |

#### Milestone 5

Milestone 5 is after the order section has been created.

|  |  |  |  |
| --- | --- | --- | --- |
| Test Number | What is being tested | Inputs | Expected Outputs |
| 5.1 | Items can be added to the order | Add ‘Chips’ to the order | Chips should be added to the order array |
| 5.2 | The UI adds buttons for each item in the order | Add ‘Chips’ to the order | A new button should appear in the order section with the details of the item. |
| 5.3 | The ability to select items in the order | Click on the ‘Chips’ button | The button should be referenced as the selected item and be highlighted. |
| 5.4 | Increase the amount of an item in the order | Press the ‘increase’ button while having a selected item, press the ‘increase’ button without having a selected item | The amount of that item should increase in the script and on the UI, Nothing should increase and there should be no change in the UI |
| 5.5 | Decrease the amount of an item in the order | Press the ‘decrease’ button while having a selected item, press the ‘decrease’ button without having a selected item | The amount of that item should decrease in the script and on the UI but no lower than 0, nothing should decrease and there should be no change in the UI |
| 5.6 | Remove and item from the order | Press the ‘remove’ button while having a selected item, press the ‘remove’ button without having a selected item | The item should be removed from the order array and the UI button should disappear, Nothing should change. |

#### Milestone 6

Milestone 6 is after the main page setup has been completed.

|  |  |  |  |
| --- | --- | --- | --- |
| Test Number | What is being tested | Inputs | Expected Outputs |
| 6.1 | The ability to change categories | Press on a different category | The button screen should change to have a different layout of buttons. |
| 6.2 | The ability to change to the payment screen | Press the ‘payment’ button | The screen should change to the payment screen |
| 6.3 | The ability to request and receive a layout from the server | The layout number that should be returned | The server should return the categories and the items that they should contain. |
| 6.4 | The ability to set the category names | The category names from the server | The category name should be set to the category buttons in the correct order. |
| 6.5 | The ability to set the item buttons | The id’s of the items to be displayed by the buttons | The buttons should appear, different colours depending on the type of item. With the correct item name attached. |

#### Milestone 7

Milestone 7 is after the Payment page has been completed.

|  |  |  |  |
| --- | --- | --- | --- |
| Test Number | What is being tested | Inputs | Expected Outputs |
| 7.1 | The ability to display how much the customer owes | 1 portion of Chips at £3.95 | The UI element should display the correct amount owed |
| 7.2 | The ability to pay a specific amount by cash | 0, 0.1, 100 where the cost is 3.95 | 3.95, 3.85, -96.05 |
| 7.3 | The ability to pay the full amount by card | Press the pay by card button | Should wait for a second (to simulate the customer completing the card transaction) and then proceed to the completion screen. |
| 7.4 | The ability to pay part cash and part cash | 0.1 and then the card button, 2.0 then the card button | In both cases the transaction should go though and be completed |
| 7.5 | The ability to send transaction data to the server | A completed transaction of each payment type | The transaction should be added to transaction database |

#### Milestone 8

Milestone 8 is after the Admin page has been completed

|  |  |  |  |
| --- | --- | --- | --- |
| Test Number | What is being tested | Inputs | Expected Outputs |
| 8.1 | The Admin page can be opened | Pressing the admin page button with sufficient permissions, pressing the admin page without sufficient permissions | Opens straight to the page, opens to the confirm menu for a higher permissions level input. |
| 8.2 | You can add items to the stock table | A valid input to each field, and invalid input to one field but valid to the rest, all invalid inputs | Sends the request to the server and adds to the database, client denies the request, client denies the request |
| 8.3 | You can add staff to the staff table | A valid input to each field, and invalid input to one field but valid to the rest, all invalid inputs |  |
| 8.4 | You can enable refund | Press the button to enable refunds | Refunds are enabled and you can add a negative amount of items. |

#### Milestone 9

Milestone 9 is after the Networking section has been completed

|  |  |  |  |
| --- | --- | --- | --- |
| Test Number | What is being tested | Inputs | Expected Outputs |
| 9.1 | The public port is opened on the public IP address | Tester is used on the following ports: 1234, 6231, 8888, 25565 | The tester should display open, open, closed, open |
| 9.2 | The client can connect to the server over the network | Client connection request | The server accepts the request and can communicate with the client |

### Post-Development testing

Post-Development testing is used to show and check that all the different modules work together. This could be used alongside an alpha or beta test, these are where you allow different groups of people to try the program and provide feedback. There are several sections in my program that will require post-development testing as part of the development process.

|  |  |
| --- | --- |
| Post-Development Test | Purpose/Justification |
| Login as a user with standard permissions, add 2 items to the order, and increase one item to have a quantity of 2. Finish the transaction and pay by part cash, part card. Check to see if the transaction appears in the table. | This shows how most of the different sections work together, and ensures that you can do the most standard of operations using the system. This shows the user system, item system and the transaction systems working together to create a fully working till system. |
| Login as a user with standard permissions, add 2 items, one that costs more than the other. Go to the admin panel and enter the id of a user with admin permissions. Select refund mode and set the item that costs more to -1 quantity. Then complete the transaction with cash payment (refund). Check that the transaction appears in the table. | This shows us that refunds can be made. And that they can be processed properly. This is another one of the main features of the program. This also shows the Staff login system working and the transaction system working. |

For our alpha testing, I will use the system extensively and try to find any bugs or issues with the program, this testing process is more about finding out if the features that are already implemented work well.

For our beta testing, I will ask other students around my college to try my program, and two of my stakeholders to try. This means that I will have a wide range of opinions. The college student I will pick have or have had part-time jobs in which they have used a till Point-of-sale system. This will show how the average user would use the software. I will also get two stakeholders to try my software this means that they can experience it from a management point of view, observing the database structure and the admin features.

# Developing a programed solution

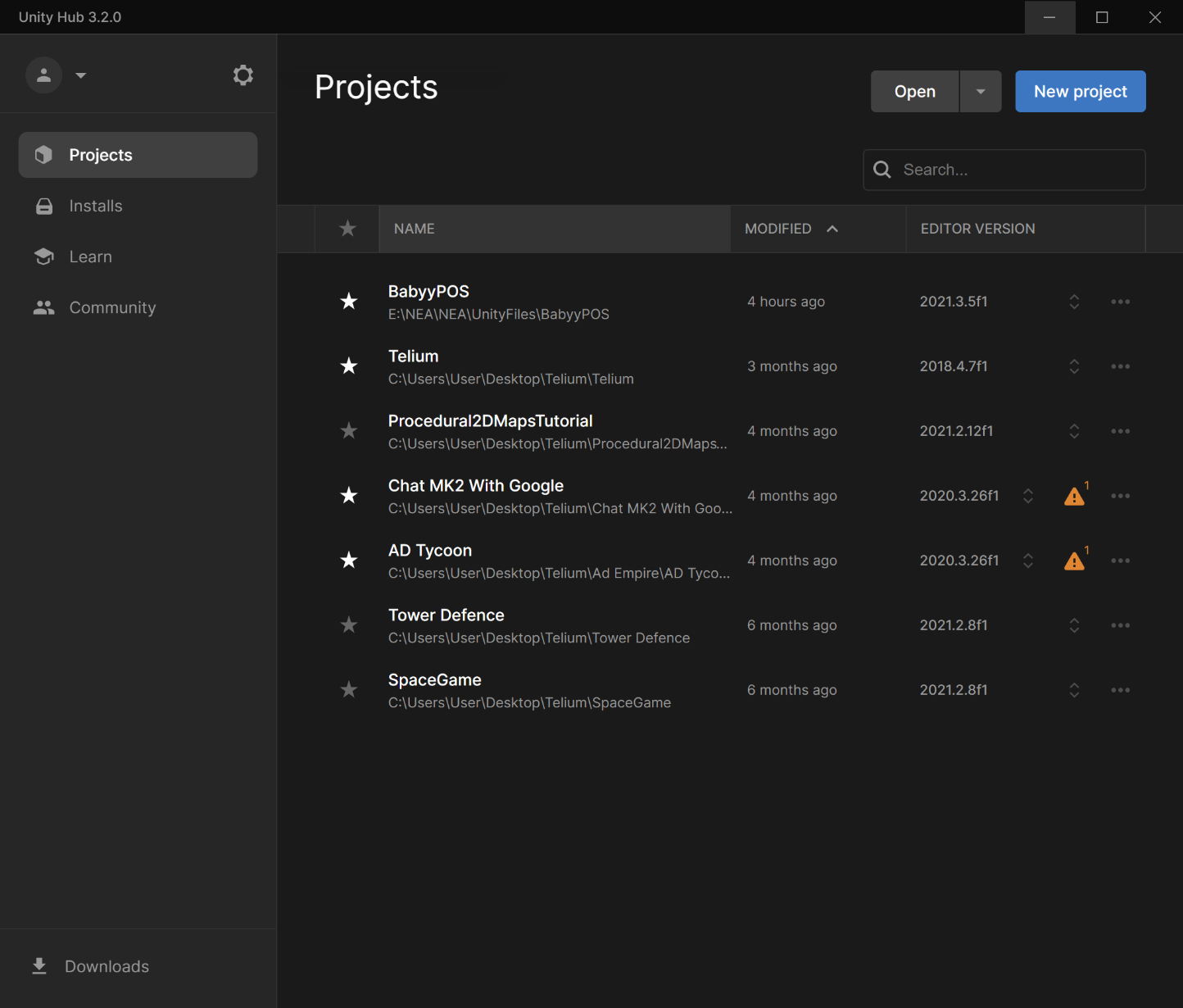
In this section I will be creating the program which I have designed in the previous sections. This will document every step of the development process and the challenges that I faced.

## Setup

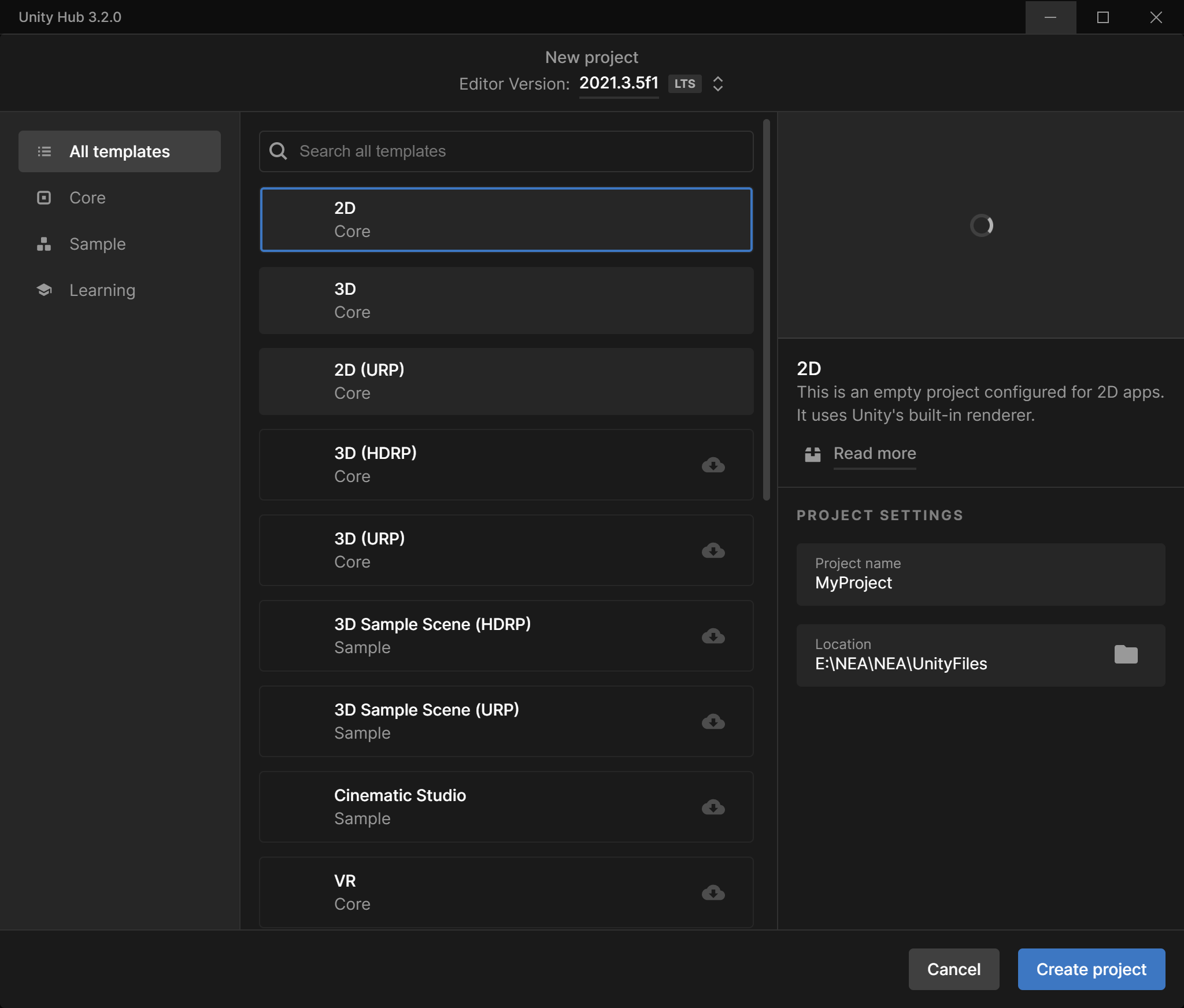
Before starting to program my solution, I have to set up my working environment. As previously stated I’m using Unity Engine [2] Version 2021.3.5f1 [11] this means that before I start I have to install this version and create my project.

### Unity

Installing the correct version of Unity Engine [2] was simple. I used the Unity Hub [21] which helps to organise different versions and projects in one centralised place. A screenshot of the main screen is shown below:



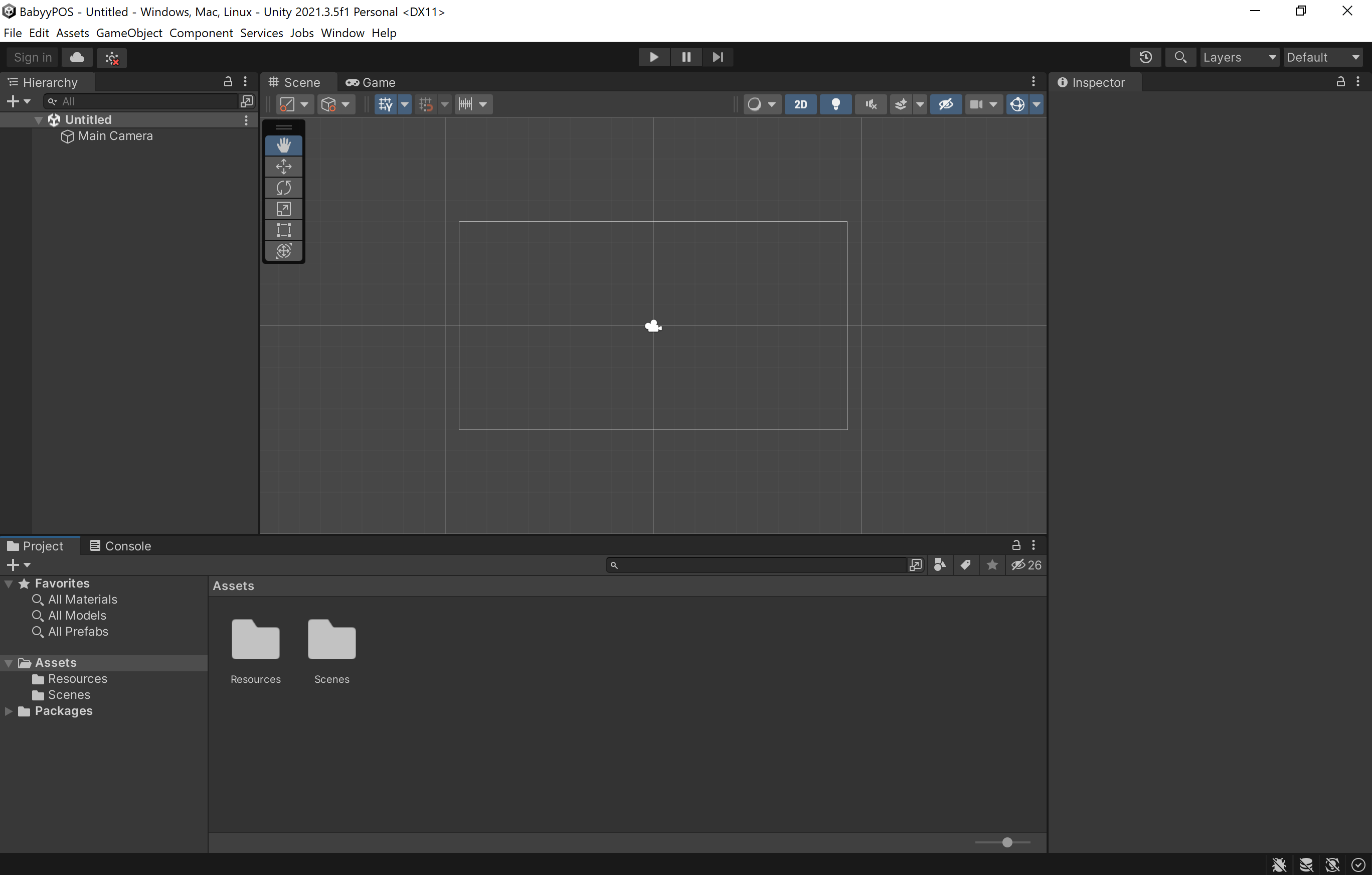
To install a new version you simply select the installs tab, press ‘Install Editor’ and select the version you would like.

Next you have to create your project. This automatically creates a file system which stores all the data needed for your game. We will be using the 2D default template, this creates a blank workspace, with a 2D environment for you to work in. 

One issue I encountered while generating my project for the first time is that it requires a vast amount of data to be downloaded and stored in order for you to use Unity Engine [2] effectively. I am storing my project on a USB Flash Drive. Although this drive has USB 3.2 capability, my laptop’s USB ports only support USB 2.0 this means that the download was slowed dramatically. Along with using my college’s WiFi, this meant that the creation of my project, and all the default files required took a while. In fact, the whole process from pressing ‘Create Project’ to Unity Engine [2]‑loading up took around 1 and a half hours.

This made me realise that next time I should create my project using a device that has USB 3.2 ports and a fast internet connection. Or create my project on the local drive of my computer as this normally has faster read/write speeds than any externally attached drive.

Once Unity finally loaded, this screen appeared:

And I was finally ready to start programming my solution.

### Files

Every project needs a robust system to store files. Without these systems in place, development can quickly become a confused mess, and difficult to find the correct files.

#### Scripts

The most essential part of my program is the scripts, these are all the program files and classes that I create to implement different features into my program.

The scripts will be organised into which sections they effect, this means that I can quickly find each file by looking at which part of the program it is for. An example of this would be a file for the networking, this will hold all of the scripts relating to the Server-Client connection.

#### Assets

Another important part of development is assets, these are files, images, sounds or other important files that are needed for the project. In this case, we will have different UI assets. Sound effects aren’t especially required. Strictly speaking we could use Unity Engine’s [2] default UI elements, however these don’t have the look or appeal of a professionally programed piece of software.

#### Scenes

When working with Unity Engine [2] you would normally work with scenes these are different screens or ‘levels’ in games. However, for our case we will only be using one scene for the client and one scene for the server. So these can be easily stored in the Scenes folder that is automatically created.

#### Graphical User Interface

Most of our project will be made using Graphical User Interface assets. In Unity this means that we have to create an area for our UI elements to be stored. This is called a ‘Canvas’ this will allow us to add UI elements to our scene and be able to interact with them using the inbuilt ‘Event System’ Almost all of our programming will be done inside this ‘Canvas’ and anything that isn’t will be specified.

### Assets

I am using 2 sets of assets for my project, these are both UI based asset packs from the Unity Store [22]

The first is ‘Clean UI Settings’ by Landan Lloyd [23] and the second is ‘Simple Icons – Classic’ by SHarmiD [24]

These will make up the UI elements that I use throughout my project. There are added to an Art folder and are separated.

## Programming

This section will document the journey I take to create my solution. This includes coded sections and UI elements. As well as troubleshooting and any issues I encounter.

### Server-Side Setup

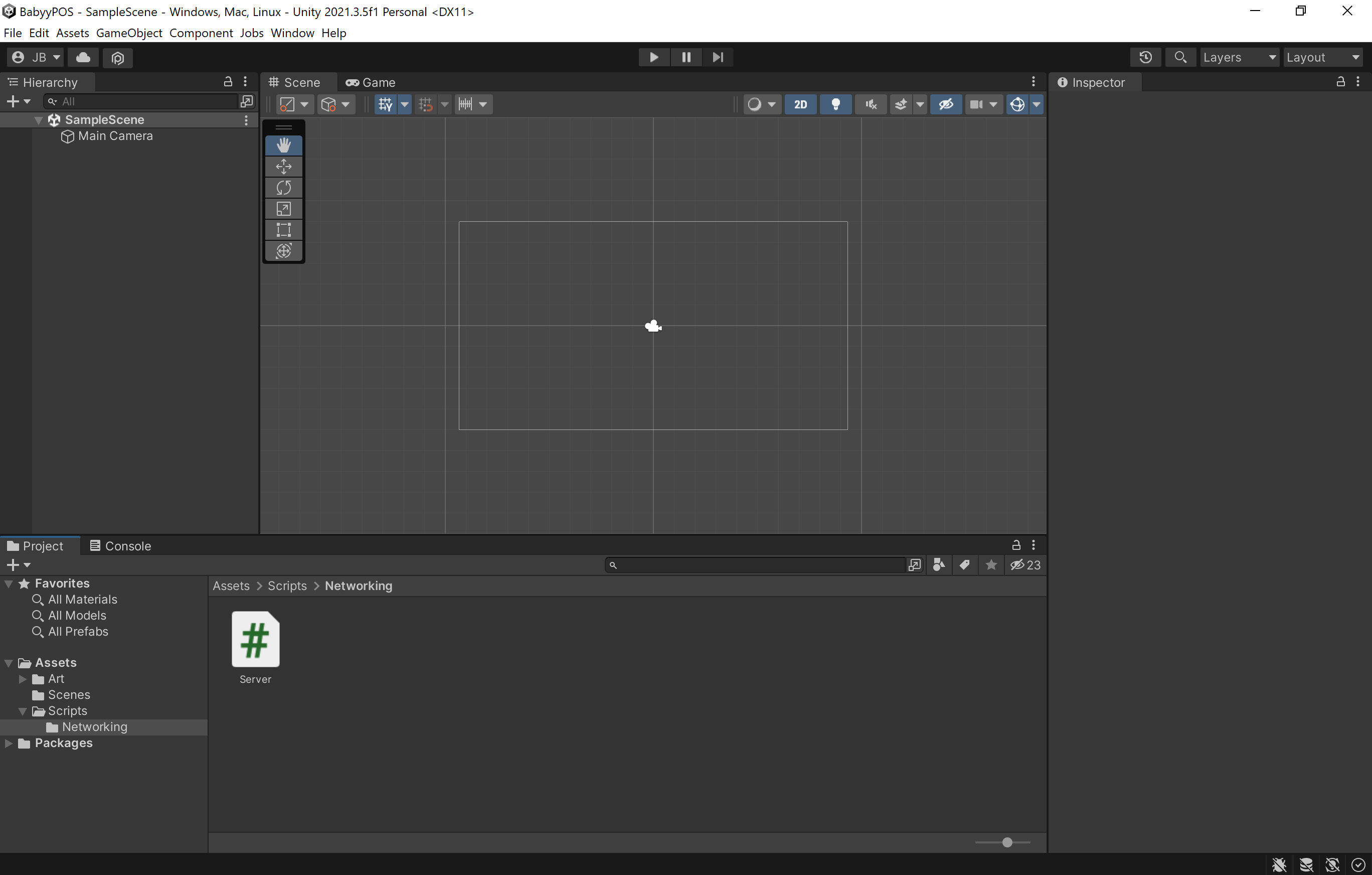
This section is about creating the server part of the Server-Client connection for our networking. To begin with this will only include the basics for connecting clients to the server, but will be added to in later sections.

#### Creating a TCP server on the LAN

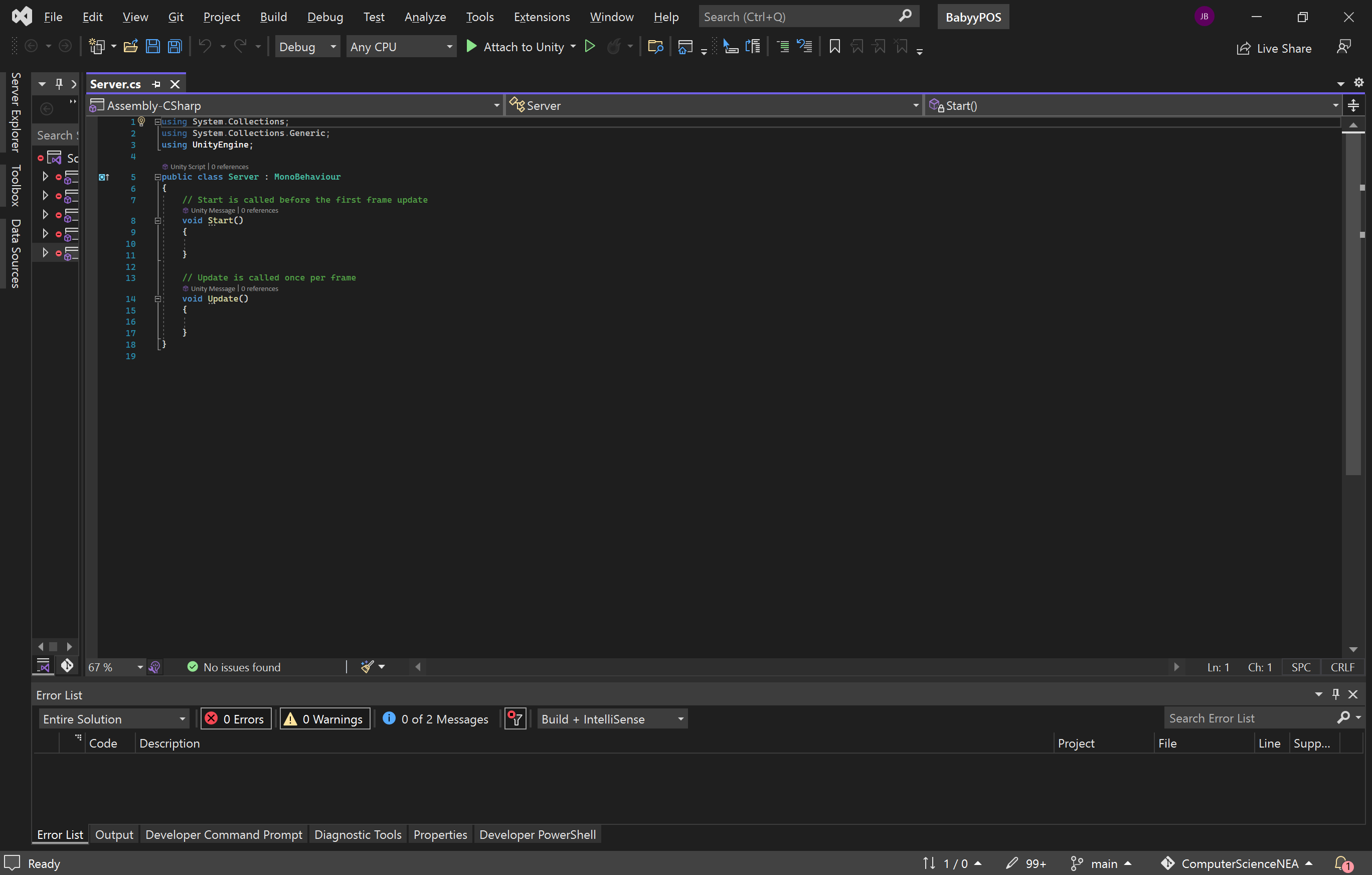
This is one of the most important parts of our programming section. We will start to create the server side of the networking connection. For this we are using some built-in C# libraries that allow direct connection using TCP. This is only designed to be used over a local network, however in later sections through Port Forwarding we will attempt to open this to the WAN.

To begin programming our server, we have to create a script. As this is the first script I am creating I will create a Scripts file and then a Networking folder to hold all of the scripts that are to do with our networking section.   
Then I will create a script called ‘Server.cs’ this is where we will program our server section.

Currently our Unity Project looks as follows:



Opening the ‘Server.cs’ File in Visual Studio [12] I am greeted with Unity’s default class structure. This is automatically added every time I create a new C# file.



Above shows how I have my Visual Studio [12] laid out, this gives me plenty of room to view my program and allows quick access to the command line and error list.

Before I can start anything with my Server script I need to import the relevant libraries that I will need. These are imported using the following lines of code:

using System.Net.Sockets;

using System.Net;

using System;

using System.IO;

Now I can initialise my variables as is the standard for all C-based languages. The lines of code for this are shown below:

public int port = 1803;

private TcpListener server;

private bool serverStarted;

public Server instance;

These are placed inside the Class structure, but outside of any procedures.

Inside the Start() procedure I have added a definition to the variable instance. The variable instance now referenced that specific script on that object. This means that I always have a specific reference to my Server class.

After this I added my ServerClient class this ensures that when the client connects we can keep track of them. Allowing us to easily communicate back to clients when the send requests to the server. This code is shown below:

public class ServerClient

{

public TcpClient tcp;

public string tillName;

public ServerClient(TcpClient clientSocket)

{

tillName = "Unknown";

tcp = clientSocket;

}

}

Next we will make sure that when the Script first runs it tries to start the server. We are only putting this in the Start procedure because we don’t want our script trying to start a server on every Update frame, before writing this piece of code, we need to new variables which will hold the connected clients and the clients that are waiting to disconnect.

private List<ServerClient> clients;

private List<ServerClient> disconnectList;

The code to start the server is shown below:

clients = new List<ServerClient>();

disconnectList = new List<ServerClient>();

try

{

server = new TcpListener(port);

server.Start();

serverStarted = true;

Debug.Log("Server has been stated on port " + port.ToString());

}catch (Exception e)

{

Debug.Log("Socket error: " + e.Message);

}

# Definitions

2D 2-Dimensional

3D 3-Dimensional

AR Augmented Reality

Array An indexed List.

ASCII American Standard Code for Information Interchange

DNS Domain Naming Service

DDNS Dynamic Domain Naming Service

C# An Object Oriented Programming language that is used with Unity Engine [2]

Co-Routines A function that can suspend its execution before reaching return. It can also indirectly pass control to another co-routine for some time

Dictionary An unordered list that is indexed using a unique identifier or key for each piece of information

GB Gigabytes or 1,000,000,000 bytes or 8,000,000,000 bits

GUI Graphical User Interface

HCI Human-Computer Interaction

ID Identification

IEnumerators C#s implementation of co-routines

IP Internet Protocol

IP address Internet Protocol address

EPoS Electronic Point of Sale

LAN Local Area Network

Linux An open-source operating system that has many different adaptations

List A store for multiple pieces of data of the same type

MB Megabytes or 1,000,000 bytes or 8,000,000 bits

OS Operating system

RAM Random Access Memory, also known as Main Memory

TCP Transmission Control Protocol

UI User Interface

Unicode An international encoding standard for use with different languages and scripts

Vigenère Cipher A type of cypher that was first described in 1586, used to encrypt text.

VR Virtual Reality

# Bibliography

[1] SQLite <https://www.sqlite.org/index.html> Accessed on the 30/06/2022

[2] Unity Engine <https://unity.com/> Accessed on the 30/06/2022

[3] TillPoint <https://www.tillpoint.com/gb> Accessed on the 30/06/2022

[4] Lightspeed <https://www.lightspeedhq.co.uk/> Accessed on the 30/06/2022

[5] iPad <https://www.apple.com/uk/ipad-air/>

[6] EPoSnow <https://www.EPoSnow.com/uk/> Accessed on the 30/06/2022

[7] Worldpay <https://online.worldpay.com/>

[8] PayPal <https://www.paypal.com/uk/home>

[9] iZettle <https://www.zettle.com/gb>

[10] SumUp <https://sumup.co.uk/> Assessed on the 30/06/2022

[11] Unity Engine Version 2021.3.5f1 <https://unity3d.com/get-unity/download/archive>

[12] Visual Studio 2022 Community Edition <https://visualstudio.microsoft.com/vs/community/>

[13] GitHub <https://github.com/>

[14] DB Browser <https://sqlitebrowser.org/>

[15] NoIP <https://www.noip.com/>

[16] macOS <https://www.apple.com/uk/macos/monterey/>

[17] Linux <https://www.linux.com/>

[18] Windows <https://www.microsoft.com/en-us/windows>

[19] Copyright, Designs and Patents Act 1998 <https://www.legislation.gov.uk/ukpga/1988/48/contents>

[20] dCode, Vigenère Cipher <https://www.dcode.fr/vigenere-cipher> Accessed on the 22/08/2022

[21] Unity Hub <https://unity.com/unity-hub>

[22] Unity Store <https://assetstore.unity.com/>

[23] Clean UI Settings by Landan Lloyd <https://assetstore.unity.com/packages/tools/gui/clean-settings-ui-65588> Accessed on the 15/09/2022

[24] Simple Icons – Classic by SHarmiD <https://assetstore.unity.com/packages/2d/gui/icons/simple-icons-classic-209472> Accessed on the 15/09/2022