Software Document

Table of Contents

[Vision statement - Jenni 2](#_Toc71743749)

[Objectives from the project: 2](#_Toc71743750)

[Objective 1: 2](#_Toc71743751)

[Objective 2: 2](#_Toc71743752)

[Objective 3: 2](#_Toc71743753)

[Advantages 2](#_Toc71743754)

[Requirements - Jack 3](#_Toc71743755)

[Component – Saidhbh 4](#_Toc71743756)

[Use cases - Jenni 7](#_Toc71743757)

[Case 1 - User wants to log into the system 7](#_Toc71743758)

[Case 2 - Adding an employee to the system 7](#_Toc71743759)

[Case 3 - Searching for a staff member 8](#_Toc71743760)

[Case 4 - Deleting a staff member from the database. 9](#_Toc71743761)

[Case 5 - Updating a user 9](#_Toc71743762)

[Project Blueprint – Chloe 11](#_Toc71743763)

[Frames 11](#_Toc71743764)

[Log In Page 11](#_Toc71743765)

[Features 11](#_Toc71743766)

[Main Functions 12](#_Toc71743767)

[Administrator Main page 13](#_Toc71743768)

[Data Grid 13](#_Toc71743769)

[Add Employee 14](#_Toc71743770)

[Search Record 15](#_Toc71743771)

[Update Asset 16](#_Toc71743772)

[Delete Record 17](#_Toc71743773)

[Exit Button 18](#_Toc71743774)

[Employee Main Page 18](#_Toc71743775)

[Other Functions 19](#_Toc71743776)

[Clear Function 19](#_Toc71743777)

[Select Function 19](#_Toc71743778)

[Find Function 19](#_Toc71743779)

# Vision statement - Jenni

Our vision for the project was to create an informative, easily usable GUI which all users would be able to understand clearly to control assets within and out of the company.

Our main goal is to keep track of Assets on a day-to-day basis, in line with the company’s policy. The next thing that we want to achieve is to create a platform that all companies looking to track their assets. This could then be sold to the wider market which would expand the opportunities that the company has to offer.

## Objectives from the project:

### Objective 1:

Create a graphical user interface that is easy to navigate around.

Aim:

By creating an interface for the business that was easy to use, we saw this as an opportunity to expand into other areas of the business, linking to areas such as disciplinary or payroll to gather information regarding the employee’s usage of assets and records.

### Objective 2:

Connecting our GUI to a database and joins to other areas.

Aim:

By connecting the GUI to a database, this can also be used as a form or tracking system, in the future, we could add things like date in and out so that the user can clearly see the time that each employee has on their particular asset. This can then be seen as a contract.

### Objective 3:

Using the interface as a physical contract

Aim:

Converting the GUI to a PDF for every individual that takes out an asset means that there is not just stored data but a physical copy for proof or loans. This means in the event of a natural disaster, there are always backups of everything.

## Advantages

One of the main advantages of the GUI is that we were able to create everything that we set out for it to do on one main page. This meant that the user can do searching, updating, deleting, and exiting all from one interface. This causing less confusion and less training is required to get the job done.

Another advantage would be that creating a database, we were able to do all of the queries through the code and means that it reduces the risk of SQL injection which was one of the main vulnerabilities that we picked up on as Asset tracking.

# Requirements - Jack

The requirement for Group 2 – Team 6 is to create an asset tracking system for the company Brainvire. The purpose of the tracking system is to create an inventory for the company’s assets where staff members are able to hire company assets such as laptops or cars, and can also hire rooms if needed. Therefore, an appropriate tracking system must be created to keep track of what is being hired and used to keep an organised environment within the company.

The main requirements for the functionality of the asset tracking system are:

* **Create a Graphical User Interface that is easy to navigate**

In order to create a successful asset tracking system. A simple but effective graphical user interface must be created which can be easily navigated. The system must have the appropriate functionality such as adding, searching and deleting records from a database.

* **Connect the GUI to the company’s database**

The GUI must be connected to the company’s database, because it is essential to be able to retrieve employee details, where they can be updated if that employee has decided to use one of their assets. By updating the database with this information, the company can keep track of what assets the employee has hired, whether it is a room, laptop or a company car.

* **Searching for employees**

The asset tracking system must be able to successfully search for an employee. When searching for an employee, admin can search by using the employee ID. If the employee ID matches with a record in the database, the correct and relative information will be displayed, including whether or not they have hired any company assets and what those assets are.

* **Update records in the database**

The system must be able to update information correctly as outdated information can lead to major problems. Every time an employee wishes to hire an asset, the employee details should be updated to reflect what assets they have hired. The same must happen when the employee returns a company’s asset. The information should be updated to reflect that the asset is no longer in the employee possession, but either in the company’s or another employees possession.

* **Delete records in the database**

Deleting records in the database will become an essential part of the asset tracking system. A record should be able to be deleted by inserting the employee ID details and then pressing a delete button which will remove the record from the database. This will reduce the number of records in the database because not all employees might hire a company’s asset, which means their details do not need to be stored in the asset tracking database. And if an employee has brought the asset that they have hired back, that record can then be deleted.

# Component – Saidhbh

Throughout the development of our Asset Tracking GUI we programmed exclusively in the Python programming language. As a high-level language it uses a more natural syntax in comparison to many of the lower-level programming languages. Lower-level languages sit close to the computer’s instruction set and maintain little, if any, abstraction from the actual processor’s instructions. Languages such as the Assembly Language and Machine Code etc. often consist of little more than 1’s and 0’s, with the code, as it is, comprising of a series of mnemonic instructions (e.g. MOV, ADD, LDA etc.) Combined with opcodes and operands it can be represented as a hexadecimal number which is decoded by the processor into its base instructions.

While the use of these mnemonics creates a more ‘user friendly’ method of forming instructions, this syntax would be rather difficult for a developer to write with and debug, let alone read and follow. They would essentially need to constantly work in binary, having very few ‘human recognizable’ words to their arsenal.

In contrast, high-level programming languages use a syntax and ‘grammar’ much closer to the human written/spoken form. A developer does not need to know about the 1’s and 0’s, the intricacies of how a keyword translates to machine code deep in the processor, only how to implement it correctly in a program. There is a high level of abstraction from the machine language. So, rather than dealing in registers, memory addresses, stacks and mnemonics, the programmer can use more higher-level abstract concepts such as functions, arithmetic, loops and threads. Essentially the grunt work is left to the compiler to translate the high-level computer code into lower-level instruction sets.

With modern computer programs often consisting of millions of lines of code, speed is of the essence and English-like statements provide a more ‘programmer friendly’ way to develop robust and efficient software.

Python, as a language, is a world away from the lowest-level languages. It was designed for its readability and is linguistically familiar making it much easier to code and debug.

Unlike other higher-level languages such as C++ and C#, it is dynamically typed. This allows the user to omit any explicit type of declaration when creating variables as this is assigned at runtime. While this type checking at runtime can affect the performance it overall, it can be more efficient at compiling. Python is also a strongly typed language with variables being assigned their types at runtime. These cannot be easily converted from one type to another, hence this typing matter when performing operations on these variables. Put simply, Python understands it is mathematically possible to add an integer and a floating-point number without being explicitly told it is allowed to. However, if an integer were to be added to a string without concatenation it would recognize the type-mismatch and produce an error.

When developing code using Python it is important to remember that the language relies on code indentation, with leading white space used as part of its grammar to define scope. In other high-level languages such as Java or C++ curly braces/brackets are used to enclose meaningful units of code, with spacing and indentation only used to enhance practical code readability. New lines define the end of a completed command, rather than the semicolons used in the C languages etc. It is worth noting that while a semicolon can be used to indicate the end of a Python command, this is rarely (if ever) used in practice. Overall, the Python language’s visual structure correlates directly with the program’s actual semantic structure, unlike other languages where indentation and white space are purely incidental and, for the most part, ‘cosmetic’.

While there are plenty of obvious differences between Python and other commonly used programming languages, there are a great many similarities when it comes to syntax, semantics and control flow. While Python is generally considered to be an Object-Orientated language, in many ways it can also be considered to be a Procedural language. Outside of class-objects etc. code is broadly read in sequence from top to bottom, with control flow being regulated and directed by a number of conditional statements, loops and function calls.

The if, elif and else statements allow the programmer to execute blocks of code conditionally with only one block being executed mutually exclusively. As the Python language does not have a switch statement it is these ‘conditionals’ that are used in its place.

Looping is permitted, albeit with a simpler more user-readable syntax to the likes of Java and C++. Unlike the aforementioned languages, a Python for-loop is used to iterate over a sequence such as a list/array, dictionary or a string. In this way it functions more closely to both the ‘foreach’ statement in C# and the ‘iterator’ in C++, rather than the indexed and conditional style of looping found in Java/C++. A for-loop in Python does not require an index variable to be set in advance as it uses the number of items in the sequence as loop-terminator, as opposed to a pre-defined integer with incrementation.

The Python while-loop on the other hand functions almost identically to the other high-level languages mentioned above, with the loop executing a block of code as long as the specified condition is satisfied. Break is also used in Python in the same way as in C languages etc. to allow the control flow to break from the loop under specified conditions, even if the while condition remains true.

As with most other high-level languages, functions are used as ‘sub-routines’ within a program as a reusable grouping of code that can be implemented whenever that particular block is needed. Rather than specifying the return type, the key word def is used without one with this again being assigned at runtime.

One of the key differences between Python and the like of C++ is in its memory management. With C languages memory management can be quite a tricky skill to master. It is possible for the programmer to manually allocate memory as and when necessary, so if memory deallocation neglected and allocated memory is not freed after use it can lead to memory leakage. Also, as raw pointers are still commonly used in C/C++, there is the issue of attempting to use already deallocated memory and ‘dangling pointers’.

This is not an issue when using Python as there is no need for either allocation or deallocation of memory. Python has its own garbage collection function to it takes care of memory management on its own, so any unused allocation of memory is automatically freed. Likewise, memory allocation is taken care of without the user interfering in the process, and as there aren’t really any pointers in Python (although they can be simulated) this removes the added issue of ‘dangling pointers’ entirely.

Perhaps one of the Python language’s greatest strengths is its Standard Library, where the exact syntax and semantics of the language are defined. As a reference manual of sorts, it offers the programmer an extensive list of facilities and functionalities. It includes ‘import’ statements which allow the developer to include specific groups of modules and functionalities into their code.

With reference to our Asset Tracking GUI, it provides modules for creating an interactive display with which the user can interface with devices, and databases etc. This is achieved through buttons, icons and interactive textual navigation. The ‘tkinter’ framework is Pythons de-facto GUI package from its standard library and is included here so as to provide the toolkit necessary to implement widgets (buttons, text boxes, labels etc.) It also provides the classes necessary to allow the logical display, positioning and control of these widgets using geometry managers such as pack or grid.

Using tkinter.ttk it is possible to import specific themed widget sets as needed, rather than simply importing them all as a complete set. The treeview widget displays a hierarchal collection of items which allows the code to control the order in which data values are displayed. This can be done with rows and columns so as to create a grid-like data display, with each item being indexed by a specific column/row pair.

Data can also be accessed using column headers which can be user-specified, thus adding a user-friendly element to the display of data, while scrollbars are used to allow large datasets to be imported from a database, without the data displayed exceeding the bounds of the display grid. It is possible to use the insert function to insert (as with the add method) a new row of data into a specified index using a unique item identifier (iid).

SQLite is another useful tool that can be integrated with Python by importing the SQLite module. This comes bundled with Python, so it is possible to include data from tables into the code by creating a connection to an already existing database. This allows actual SQL queries to be executed to retrieve specific data from the database and provided us with a straightforward method to import our large dataset to be displayed in our treeview grid.

It is also possible to modify the database rows directly through INSERT, UPDATE and DELETE statements. Standard SQL queries can be hard coded into the program so that user input and widgets can be used to alter, not only the displayed data, but the database itself directly.

# Use cases - Jenni

## Case 1 - User wants to log into the system

|  |  |
| --- | --- |
| Use Case 1 | User wants to log into the system for further action. |
| Actor | Admin |
| Use Case Overview | Admin wants to log into the system with their username and passwords to keep track or add users and assets to the system |
| Subject Area | User log in |
| Trigger | User needs to log in to access information on the staff and assets that have been taken out |
| Precondition 1 | User successfully enters the system |
| Precondition 2 | Admin needs to change information based on the information in the table. |
| Actors | The admin |

Basic Flow

|  |  |
| --- | --- |
| Description | This scenario describes the event of a user entering the system |
| 1 | User enters the system |
| 2 | System requires a username and password from the user. |
| 3 | Terms and conditions need to be accepted. |
| 4 | If both the username and password match and terms and conditions have been met, they are now logged in. |
| 5 | Menu reappears, granted to the whole system |
| Termination outcome | Countdown on the main menu to exit if no interaction with the system and button if wish to exit. |

## Case 2 - Adding an employee to the system

|  |  |
| --- | --- |
| Use Case 2 | User wants to add a new member of staff to the system |
| Actor | Admin |
| Use Case Overview | A member of staff needs to be logged into the system for removing assets from the company |
| Subject Area | Adding details |
| Trigger | User needs to log in to be able to add information to the system. |
| Precondition 1 | Admin needs to add a member of staff |
| Precondition 2 | Information then can be requested. |
| Actors | The admin |

Basic Flow:

|  |  |
| --- | --- |
| Description | This scenario describes the event of admin adding both a staff member and asset to the system |
| 1 | User enters the system |
| 2 | Admin enters the following details about the member of staff: |
| 3 | Staff ID |
| 4 | First Name |
| 6 | Surname |
| 7 | They then select the type of asset required from the drop-down box. This could be: Car, Mobile or Laptop. |
| 8 | On selecting the ‘Add Record’ button. |
| Termination outcome | Termination can be made when the information is added to the database and table. |

## Case 3 - Searching for a staff member

|  |  |
| --- | --- |
| Use Case 3 | Admin needs to search for a member of staff |
| Actor | Admin |
| Use Case Overview | Information about a particular member of staff needs to be found |
| Subject Area | Searching |
| Trigger | Details about a member of staff is accurate and up to date |
| Precondition 1 | Admin needs the staff ID. |
| Precondition 2 | Information needs to be up to date if found correctly. |
| Actors | The admin |

Basic flow

|  |  |
| --- | --- |
| Description | This scenario describes the event of admin searching for a member of staff |
| 1 | User enters the system |
| 2 | Admin enters the staff ID of the member they are looking for |
| 3 | Selecting the ‘Search User’ button. |
| 4 | The database then uses a select query using the ID to display information from the table. |
| Termination outcome | After selecting the button, the ID, first name, surname and the type of asset should appear on the screen. Termination of the program can be made after this. |

## Case 4 - Deleting a staff member from the database.

|  |  |
| --- | --- |
| Use Case 4 | Admin needs to delete a member of staff from the database. |
| Actor | Admin |
| Use Case Overview | An asset has expired or been returned and needs to be deleted from the system |
| Subject Area | Deletion |
| Trigger | Asset has expired or been returned |
| Precondition 1 | Admin needs the staff ID. |
| Precondition 2 | Data needs to exist in the database |
| Actors | The admin |

Basic Flow

|  |  |
| --- | --- |
| Description | This scenario describes the event of admin deleting a member of staff from the system |
| 1 | User enters the system |
| 2 | Admin enters the staff ID of the member they are looking to delete |
| 3 | Selecting the ‘Delete User’ button. |
| 4 | The database then uses a select query using the ID to delete the information from the table but keeping the staff member ID known. |
| Termination outcome | After deletion, data should be gone from the table and termination can be done after the countdown or selecting the exit button. |

## Case 5 - Updating a user

|  |  |
| --- | --- |
| Use Case 5 | Updating a user’s information |
| Actor | Admin |
| Use Case Overview | Information about a particular member of staff needs to be updated. This could be the type of asset or the name dependent on the ID. |
| Subject Area | Updating |
| Trigger | Details about a member of staff is accurate and up to date |
| Precondition 1 | Admin needs the staff ID. |
| Precondition 2 | New information needs to be correctly added |
| Actors | The admin |

Basic Flow

|  |  |
| --- | --- |
| Description | This scenario describes the event of admin updating the details of a member of staff |
| 1 | User enters the system |
| 2 | Admin enters the staff ID of the member they are looking to update |
| 3 | After entering the following information |
| 4 | First name |
| 5 | Surname |
| 6 | Type of asset |
| 7 | After selecting the ‘Update User’ Information in the database and table should now be changed. |
| Termination outcome | If data has been updated correctly, user can exit the system through the exit button or process other functions on the interface. |

# Project Blueprint – Chloe

The project blueprint acts like a user guide for software developers which allows them to familiarise themselves with the project’s features. It also provides them with details of the main features’ functionalities, giving the developer an idea of the operability of the project as a whole through each feature.

## Frames

The project is split into three frames, Log In page, Administrator Main page and Employee Main page.

## Log In Page

Graphical user interface, application

Description automatically generated

### Features

|  |  |
| --- | --- |
| *Feature* | *Description* |
| Username text box | The user enters their Username in this text box that will be verified with their password allowing them to log in. |
| Password text box | The user enters their password in the text box which will be verified along with their username allowing them to log in. |
| Log In button | When this button is clicked, the Username and Password that the user entered are verified through the database before logging in successfully. If they are incorrect then the user does not log in. |
| T&C checkbox | This checkbox is used for the user to check provided they agree to the terms and conditions. If they do not agree then they cannot log in to the main page. |
| Policy Document link | The user can click on the link that will direct them to a webpage where they can read the policy document before accepting the T&C. |

### Main Functions

#### Log In Function

The Log In page’s main functionality resides in a single button ‘Log In’. The code executed when the ‘Log In’ button is clicked lies in the def \_login\_btn\_clicked() function.

|  |  |
| --- | --- |
| *Code* | *Description* |
| username = entry\_username.get()  password = entry\_password.get()  check = var1.get() | We first retrieve the users Username and Password along with the state of the check button. |
| search\_user(username, password, check) | The username and password entered are passed through the search\_user function to be validated through the database. |

#### Search Function

The def search\_user(user, password, check) function carries out the remaining functionality of the Log in button verifying the user’s credentials. If the log in is successfully then there are two possible pages the user can be directed to depending on if they are an Administrator or an Employee.

|  |  |
| --- | --- |
| *Code* | *Description* |
| query = "SELECT \* FROM Credentials WHERE Username = '" + user + "' AND Password = '" + password + "'"  result = find\_record(my\_conn, query) | We use SQL to search for entered username and password in the database. The SQL query is passed to the find\_record function and the result is stored in a variable. |
| search\_user(username, password, check) | The username and password entered are passed through the search\_user function to be validated through the database. |
| if not result:  messagebox.showerror("Error ", "Invalid username/password") | If the user’s credentials do not match with any in the database, then an error is thrown. |
| elif result and check == 0:  messagebox.showerror("Login error", "Accept Terms and Conditions to continue") | If the credentials are correct but the T&C have not been agreed to by ticking the checkbox, then an error is thrown with a message box stating the nature of the problem. |
| elif result and check == 1:  subquery = "SELECT \* FROM Credentials WHERE Username = '" + user + "' AND Admin = 'y'"  result2 = find\_record(my\_conn, subquery) | If the credentials are correct and the T&C have been agreed to, then another SQL query is used on a different table in the database to extract their role (Administrator/Employee) and direct them to the correct page. The query is passed to the find\_record function to be executed.  The result from the function is stored in a variable result2. |
| if result2:  raise\_frame(AdminFrame)  root.geometry('1000x400')  countdown(500, AdminFrame)  elif not result2:  raise\_frame(EmployeeFrame)  root.geometry('780x400')  countdown(500, AdminFrame) | If the user is an Administrator, then they will be directed to the corresponding page however, if they are an Employee then they will be directed to The ‘Employee Main page’. |

## Administrator Main page

A screenshot of a computer

Description automatically generated with medium confidenceThe Admin Main page consists of six sections. The data grid displaying the database data, Add Employee, Search Record, Update Asset, Delete Record, and the Exit button.

### Data Grid

A data grid is used to display the data from the database. Each record is taken from the database and placed in a single row in the grid. The data shown in the grid can be edited by adding, updating, or deleting specific records. Each time a change is made, the data grid is updated reflecting the changes made. In order to retrieve and manipulate the data in the database, SQL statements are used.

#### Main Functions

The data grid is controlled by a single function def display\_data().

|  |  |
| --- | --- |
| *Code* | *Description* |
| for index in asset\_tree\_view.get\_children():  asset\_tree\_view.delete(index) | Loop through all records in the data grid.  get\_children() returns a list of all row's id asset\_tree\_view.delete(index) Iterate through all rows, passing the ID to the delete() method to clear the grid ready for population. |
| r\_set = my\_conn.execute('''SELECT \* from User''') | Everything from the User table in the database is selected and stored in a variable. |
| for user in r\_set:  asset\_tree\_view.insert('', 'end', values=user) | For each user record retrieved from the database, insert record into the data grid as a new unique entry. |

### Add Employee

The Add Employee section allows the user to add an employee to the database by entering the ID, name, and surname of the employee in the specified textboxes and by choosing an asset from the drop-down list.

#### Features

|  |  |
| --- | --- |
| *Feature* | *Description* |
| ID text box | The user enters the User ID in this text box for the employee they want to add to the database. |
| Name text box | The user enters the Name in this text box for the employee they want to add to the database. |
| Surname text box | The user enters the Surname in this text box for the employee they want to add to the database. |
| Asset drop-down list | The user can choose the Asset they want for the employee that they are adding to the database. They can choose one of three assets: Car, Mobile, Laptop. |
| Add button | When the user clicks this button, all the details of the employee are added to the database and will be shown in the data grid provided that all fields have been filled and the User ID is not already present in the database. |

#### Main Function

The code executed when the ‘Add’ button is clicked lies in the def add\_data() function.

|  |  |
| --- | --- |
| *Code* | *Description* |
| my\_id = tid2.get('1.0', END)  my\_asset = options.get()  my\_name = tName.get('1.0', END)  my\_surname = tsname.get('1.0', END) | All the employee’s details to be added are retrieved from the text boxes and are stored in variables. |
| if int(my\_id):  my\_conn.execute("""  INSERT INTO User(ID, Name,  Surname, Asset)  VALUES (?,?,?,?)""", (my\_id,  my\_name, my\_surname, my\_asset))  my\_conn.commit()  display\_data() | SQL is used in order to add the details to the database provided that the user ID entered is valid.  The display\_data() function is then called to refresh the data grid and display the data added. |
| clear\_boxes() | All the text boxes and the drop-down list are reset ready for the next employee to be added calling this function. |
| else:  messagebox.showerror("Error", "Invalid input")  except:  messagebox.showerror("Error", "Invalid input") | If any errors occur, then they are caught through the else and except statements displaying a message box if any errors do occur. |

### Search Record

If the user wants to view the details for a specific employee, then they use the Search Record section as a quick alternative to scrolling through the data grid trying to find that particular employee.

#### Features

|  |  |
| --- | --- |
| *Feature* | *Description* |
| User ID text box | The user enters the User ID in this text box for the employee they want to search in the database. |
| Search button | When the user clicks this button, all the details of the employee they searched for will be shown in a message box provided that the search was successful and an employee with the specified details was found. |

#### Main Function

The code in the def search(id) function is executed when the ‘Search’ button is clicked.

|  |  |
| --- | --- |
| *Code* | *Description* |
| try:  int(id) | We check that the ID value past to the function is valid using a try- catch statement. |
| q = "SELECT \* FROM User WHERE ID= "+id | Using an SQL statement, we select the employee with the specified ID. |
| data\_row = my\_cursor.fetchone()  messagebox.showinfo("Employee Details", f"ID: {data\_row[0]}\nName: {data\_row[1]}\nSurname: {data\_row[2]}\nAsset: {data\_row[3]}" ) | The row found is stored in a variable and then by taking each field of the row, the data is displayed in a message box. |
| except sqlite3.Error as my\_error:  print("error: ", my\_error)  except:  messagebox.showerror("Error","Check input") | If there was an error with the database or if there was invalid input, throw an error. |

### Update Asset

This section is used when the user wants to update the asset of one of the employees.

#### Features

|  |  |
| --- | --- |
| *Feature* | *Description* |
| ID text box | The user enters the User ID in this text box for the employee they want to update the asset of in the database. |
| Asset drop-down list | The user can choose the Asset they want for the employee that they are updating. They can choose one of three assets: Car, Mobile, Laptop. |
| Update button | When the user clicks this button, the asset of the employee they specified will be updated and the data grid will be refreshed reflecting this change. |

#### Main Function

The code in the def update() function is executed when the ‘Update’ button is clicked.

|  |  |
| --- | --- |
| *Code* | *Description* |
| my\_asset2 = options2.get()  my\_id2 = tid3.get('1.0', END) | The ID and asset that the user specified are stored in variables. |
| r\_set = my\_conn.execute("""UPDATE User SET Asset= ? WHERE ID= ?""", (my\_asset2, my\_id2))  my\_conn.commit() | The asset of the employee is updated using an SQL statement to manipulate the database. |
| messagebox.showinfo("Updated", "Record updated") | A message box appears to inform the user that the record was updated. |
| display\_data()  options2.set("") | The display\_data() function is called to refresh the data grid to show the changes made and the drop-down menu is reset for the next update. |

### Delete Record

This section allows the user to delete an employee of their choosing. They enter the employee’s ID and then click the Delete button to delete the record.

#### Features

|  |  |
| --- | --- |
| *Feature* | *Description* |
| ID text box | The user enters the User ID in this text box for the employee they want to delete. |
| Delete button | When the user clicks this button, the asset of the employee they specified will be deleted and the data grid will be refreshed reflecting this change. |

#### Main Function

The code in the def delete() function is executed when the ‘Delete’ button is clicked.

|  |  |
| --- | --- |
| *Code* | *Description* |
| my\_id3 = tid4.get('1.0', END) | Get the ID specified by the user and store it in a variable. |
| if len(tid4.get("1.0", END)) <= 1 :  my\_id3 = tid2.get('1.0', END)  result = find\_record(my\_conn, """SELECT \* FROM User WHERE ID=""" + my\_id3) | The record selected to be deleted is highlighted and if the length of the inputted data in the delete text box is less than 0 (box is empty), then take the ID value from the 'add ID' textbox which was populated by user selection and use an SQL statement in the find\_record function. |
| if not result:  messagebox.showerror("Error ", "Record does not exist") | If the ID is incorrect or does not exist in the database, then an error message appears and deletion is unsuccessful. |
| r\_set = my\_conn.execute("""DELETE FROM User WHERE ID=""" + my\_id3) my\_conn.commit() | Using SQL, the record with the specified ID is deleted. |
| messagebox.showerror("Deleted ", "Record deleted")  clear\_boxes()  display\_data() | A message box appears informing the user that the record has been deleted and the data grid is refreshed to depict the change by calling the display\_data()function and the text box is cleared using the clear\_boxes()  function. |

### Exit Button

The Exit button when clicked terminates the program. However, the button has a timer attached to it so if the user is still on the page when the time has run out then the page will terminate automatically. This is a safety feature that has been implemented that allows an ample amount of time for the user to complete their tasks before the program terminates, requiring them to log in again to continue if it does. This prevents exposure to attacks that steal or reuse user's session identifiers allowing multiple users accessing the same account.

#### Main Code

This code is attached to the Exit button terminating the program when clicked.

bExit = tk.Button(my\_w\_1, text='Exit', width=8, bg='light grey', command=lambda: destroy\_frames())

The code in the def countdown(time) function is executed when the time runs out on the timer.

|  |  |
| --- | --- |
| *Code* | *Description* |
| if time == -1:  AdminFrame.destroy()  LoginFrame.destroy()  EmployeeFrame.destroy()  root.destroy() | If the time reaches -1, terminate the program and destroy all the frames. |
| else:  l4 = tk.Label(my\_w\_1, text="time remaining: %d seconds" % time) # show with a label how much time is left | Otherwise, display the time remaining with a label. |
| my\_w\_1.after(1000, countdown, time-1) | Every 1000 ms countdown by 1 second each time. |

## Employee Main Page

The Employee Main page has three of the features present in the Administrator Main page. Data Grid, Search Record and Exit Button. Therefore, the Employee page has limited functionality allowing the Employee user permission to only search for a record and view it in the data grid. They are not allowed to manipulate the data as that is only to be done by an Administrator.

Graphical user interface, application

Description automatically generated

## Other Functions

### Clear Function

The clear\_boxes() function is used to clear and reset all text boxes ready for next user input.

def clear\_boxes():

options.set("Select Asset")

tid2.delete('1.0', END)

tName.delete('1.0', END)

tsname.delete('1.0', END)

tid4.delete('1.0', END)

### Select Function

The def select\_user(event) function is used when a user clicks on an entry in the data grid table the selected record is inputted into the 'add employee' boxes

|  |  |
| --- | --- |
| *Code* | *Description* |
| clear\_boxes() | Resets all the textboxes by calling this function. |
| index = asset\_tree\_view.selection()[0] | Sets index to the returned row index selected by the user. |
| selected\_item = asset\_tree\_view.item(index)['values'] | Sets variable selected\_item to the string of values that are at that grid index/row. |
| tid2.insert(END, selected\_item[0])  tName.insert(END, selected\_item[1])  tsname.insert(END, selected\_item[2]) | Inserts value at specified column index to the corresponding textbox. |
| options.set(selected\_item[3]) | Sets the value of the drop-down menu, options, to the value at column index [3]. |

### Find Function

The def find\_record(connection, query) is used to execute the specific query that is passed to it and then returns the result.

|  |  |
| --- | --- |
| *Code* | *Description* |
| cursor = connection.cursor()  result = None | Establish a connection with the cursor and initialise the result to ‘None’ |
| cursor.execute(query)  result = cursor.fetchall()  return result | If the record was found, then assign it to the result variable and return it otherwise return ‘None’. |