Introduction

This documentation aims to provide an insight into the work effort and fundamental knowledge that went into the creation of the requested application. The requested application mentioned, is an e-commerce based executable program called "Finnplus". The scenario presented visualises a company that has just began, and requires some sort of interface for interaction between the user and the goods. Hence, this indicates that various forms of transactions must take place. The task assigns the developer to create data model designs through the use of Hackolade as a blueprint for the interaction between program and databases. Due to use of databases, this means that some form of backend will be required. Frontend will need to be some sort of graphical user interface that the customer can interact with. These menus created must provide the user appropriate queries.

There are various ways to create such an application. This could be done through several languages. For the purpose of this assignment, the application will be done through a Graphic User Interface which will be Python based. The reason for this choice rather than a conventional language such as C#, is because of experience and knowledge in Python. The library Tkinter will be used in this project because the priority set is functionality over aesthetics. Furthermore, it can be more programmer-friendly and due to experience in python, can lead to more complex and efficient code outside the use of the actual library. The choice is further solidified as it would present the user with a more friendly environment.

Looking through the list of menus that are required, each of these will need some sort of information to extract off of. At the core of the application are the products. This is an e-business and the goods are what is important. Hence, there must be a place to store products, so they can be accessed, updated, created and/or deleted. With the need of some sort of cart to store the wanted products, there should be a separate area to store the information. For a cart to be used, a user must be created, therefore some form of login or signup is required which will require its one place to store this information. This way, removing and adding products can be simplified. Shopping cart and checkout will be separate entities in the sense, and will have two different collections. This way, if a user chooses to save their information, a reference can be made to the login collection which can store this information for future purchases. The extensive decision-making and design-based application is described and explained in the design section. Essentially, each of these groups of data will have their own collection in the database.

In regards to the backend of the project, there will be use of a NoSQL database. NoSQL being preferable due to its flexible nature, and ability to scale both vertically and horizontally depending on which databases are used. The main database to be applied and implemented will be MongoDB. In this database, it will store the important information such as the products, the user info and login info. MongoDB is an excellent choice for its higher speeds for querying, while additionally providing benefits of more availability. Furthermore, if the company eventually expands and grows larger, there will not need to be many changes done. Sharding could be introduced, where data will be spread among several nodes to provide constant processing without pause. (Bruce, 2021) With MongoDB it will be relatively simple to setup methods to create a listing for a product, query for existing products, and display product images. This now covering half the required menus with one database.

For the rest, adding products to a cart and removing them, then checking out can still all be connected to the MongoDB database as there would only need to be made two more collections. One collection which stores the products and their information for the shopping cart, and then one more collection that can store the order permanently. However, the details of this will be further gone into in the design section of the documentation. To see the visualisation and data model design see the figure in the design section, where Hackolade was used to create the backend database architecture.

This is the thought process behind the implementation of a document-oriented database for data on the actual products and other components of the application. In essence, the overview in this section aimed to provide a sufficient and basic overview of the solution to the presented problem with starting an e-commerce-based company.

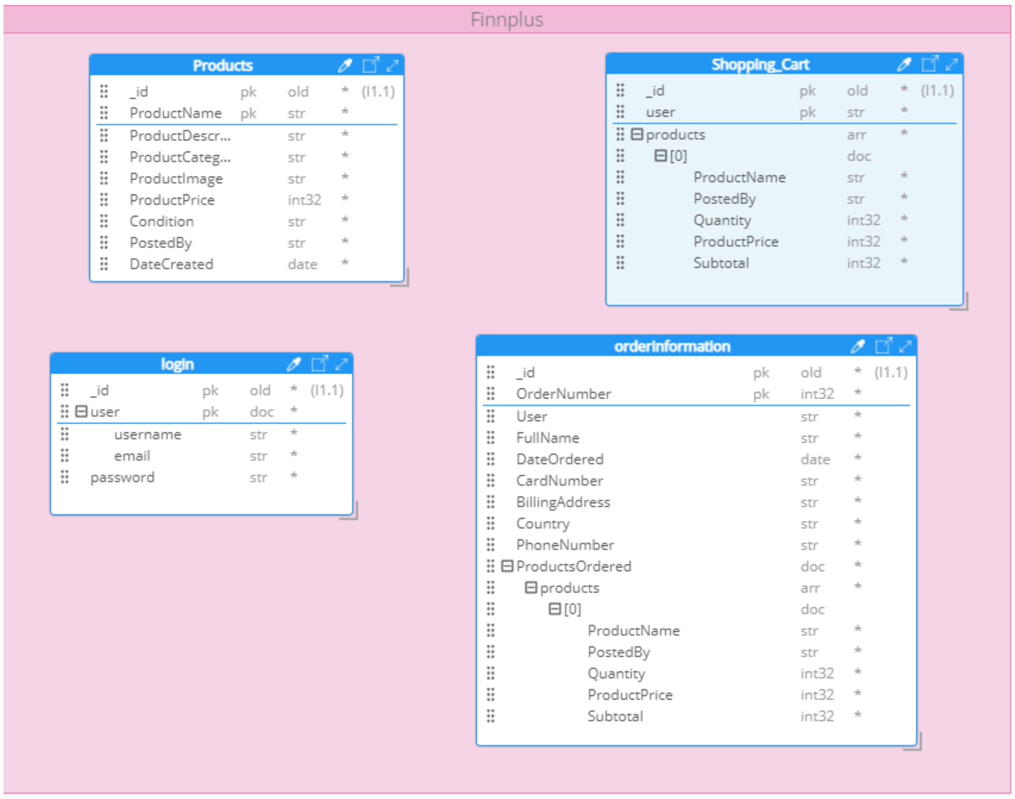
Design

GUI, Tkinter and Python

Designing the application was one of the core components of this project. It needed a suitable and efficient builder that would allow all the various necessities to be completed. Tkinter was chosen for this purpose as mentioned before. There were several reasons behind this decision. The main being that the creation of menus could potentially be done in a simple way. Due to already working with Tkinter in the past, it seemed a good idea to use something that is already familiar and is specifically made for simplicity's sake. This library, available through Python, would be implemented in the code. While Python would be the interpreted language used. By applying Python, it can be possible to create more rigorous and complex features and methods in the application. This is partly due to programming experience with python, over other coding languages. Generally speaking, Python and Tkinter will be used to create a window (Graphic User Interface) with which the customer will be able to interact with. From the main page when the program is loaded there will be several menus that navigate to various parts of the application.

Since this application is e-commerce based, product queries and anything related to the goods of the website are a priority. Hence why the following menus are on the exhaustive list and require most focus: 1. Being able to post a product, 2. These products should be searchable, 3. Products must have an image, 4. Products must be able to be added to cart, 5. Products must also be removable from cart, 6. A checkout section to purchase said items. These are a list of features which the application must have by default. However, the idea behind products being posted, and then bought suggests that some form of user must exist for this to occur. Hence, it was chosen that a login and signup page will be additional menus created. Each of these menus follow a creation process, which differ from each other. The details of these menus are individually explained below in the "Menus, Classes, Frames" section. All of these menus however, must be connected in some way. Because the idea of this design is that the menus will display information which is common between them.

This meant that information must somehow go to all of these menus, or also be created at one menu and then used at another one. This was one of the key database designs that had to be considered. There were several options. How this would be done could be through either client side or server side. This included creating relationships between several collections and then applying them accordingly, or using embedded documents. It would even be possible to transfer data into lists on the client side, where they could be updated, manipulated, deleted etc. For the purpose of this project, the database design choice was to have several collections, but they would not be referenced as relationships or stored as embedded documents. Instead, each collection would keep the relevant and common data, while also containing its own unique data. To see the database structure and design, refer to figure 1. Here the main database, and its collections are visible.



Products

_id	pk	oid	*	(I1.1)
ProductName	pk	str	*	
ProductDescr...		str	*	
ProductCateg...		str	*	
ProductImage		str	*	
ProductPrice		int32	*	
Condition		str	*	
PostedBy		str	*	
DateCreated		date	*	

Shopping_Cart

_id	pk	oid	*	(I1.1)
user	pk	str	*	
⊟products		arr	*	
⊟[0]		doc		
ProductName		str	*	
PostedBy		str	*	
Quantity		int32	*	
ProductPrice		int32	*	
Subtotal		int32	*	

login

_id	pk	oid	*	(I1.1)
⊟user	pk	doc	*	
username		str	*	
email		str	*	
password		str	*	

orderInformation

_id	pk	oid	*	(I1.1)
OrderNumber	pk	int32	*	
User		str	*	
FullName		str	*	
DateOrdered		date	*	
CardNumber		str	*	
BillingAddress		str	*	
Country		str	*	
PhoneNumber		str	*	
⊟ProductsOrdered		doc	*	
⊟products		arr	*	
⊟[0]		doc		
ProductName		str	*	
PostedBy		str	*	
Quantity		int32	*	
ProductPrice		int32	*	
Subtotal		int32	*	

Figure 1: Hackolade – Finnplus Database Design

Regarding the requirements of the program when it comes to execution, there are two factors. Two files must exist in the same directory as the code that will be executed. The first is the images folder, which retrieves and handles images of products. Secondly, the "Finnplus" folder which contains the dump files, which the database will be restored from and launched.

As can be seen from the figure there are 4 collections, but none of them are related. However, if looked at closer it can be seen that many of the collections share similar fields. There are no relationships between the collections, because none are required in this application and database design. The application communicates data by applying the appropriate database. This means that, for example, when a new product is created, it will get added to the products collection. When a customer then searches for this product, and adds it to the shopping cart, it will be added to the users shopping cart. The thought process behind this is that it can simplify the communication between different menus. User login and creation relates to the login collection. In this collection there is a username, their email and password by default. These are required because a user must be able to be identified and signed in. Each user must be unique, in order to create unique product posts and shopping carts. Hence it can be considered a primary key. In essence, what this demonstrates is client sided data transactions.

Then there is the product and shopping cart collections. These two collections mainly communicate with each other. Considering that a product can be created, and that a product can be searched, then these two menus must both relate to the products collection. They can either insert, or query over the data. When a document is created certain fields must be satisfied. Among others are the "postedby" and "datecreated" fields which are automatically generated. The user logged in is gotten and the "datecreated" is put to the day it was made. This relevant information is then used when displaying a product. Furthermore, an image is stored and displayed through this product collection. Hence demonstrating how one collection can be used across several menus through client-side interaction and manipulation. Three main and required menus rely on one collection.

The above process was a solid reason behind the choice of this design. When a document is added or changed in one collection it will not necessarily impact other collections, unless it is meant to. This can be seen as ensuring data consistency and availability to those using the program. The connection between "Products" and "shopping_cart" is done when a product is added to the cart. However, there are several checks, implementations and features done as improvements to make sure this can be done. Shopping cart requires a user, this means that the person viewing the product needs to sign in before adding a product to their cart.

Once this is checked off, the product can be added. The shopping cart stores the name, user who posted the product, the product price and the quality. The subtotal is automatically calculated and added to the document. When a user views their shopping cart, they can now see all the items they've added. This collection will be filled with documents that are unique to the user, hence even if the application is closed, they can be retrieved. Furthermore, the choice for the structure allows for multiple users to interact with the database at the same time, utilising the capabilities of MongoDB through multiple queries occurring simultaneously. However, this does not take into account a scenario where one user is logged in on multiple machines.

The explanations above, and figure presented briefly describe how the database is the core of the application, and is fundamental to the communication between menus. It highlights how information is taken from one collection and put into another. How this is achieved and what these factors are, will be discussed in further detail under the experimentation section, where code is shown and explained. One aspect of such is the design and creation of these user menus.

Menus, Frames, Classes

As mentioned, there is a total of 6 features that the application had to contain. There are additions added to the program in terms of menus that improve the interface. For example, this would be the login and signup feature briefly mentioned. However, in this section each section will be gone over in detail to discuss how the menus are generally connected, what each menu contains and general improvements and checks in place to assure data is valid and available.

The very first design choice was to have a main page menu. This would be what the user would see when they first launch the application. This means that this would be the menu that would be connected to most menus, as that was deemed most ideal for the situation. The frontend design for this was simple, and involved the creation of several labels and buttons. A "label" that would also function as a button on other menus, was the logo. Meaning that usually for a user to return to the main page menu, all they would have to do is click on the logo. The two main and largest buttons on the main menu were the post and search a product. These two buttons would link the PostProduct and the SearchProduct menus. Furthermore, at the top right there was a login and cart button. As the names suggest these would link to the login page and the cart page. However, in this design, a user would need to be logged in, in order to access the cart or the post product menus. Additionally, the username of the person logged in would display on the side of the log in/out button (it would change depending on whether logged in was true or not)

Going to the product pages, as mention the user needs to be logged in to post a product. The reason for this choice is because the username is required for the "PostedBy" in the products database. On the frontend part, there are several entries for the required fields, it follows a basic and simple design as to not confuse the user. One part that needed to be incorporated was the choosing and retrieval of images. An image needed to be retrieved and then displayed to the user, because it would later be resized. This can assure that the user chooses a correct size for the image. Furthermore, another feature in this design is that all entries are gone over and checked to be filled out. This includes making sure that quantity remains an integer. Once all fields are typed in, with no error returned, then product is submitted, and it can be viewed in the product search.

The product search design was an interesting topic. The reason being that there are many ways to accomplish it. One interesting method was a box list. The benefits of this being that all products can be displayed, and if there are too many, the user can automatically scroll. The downside being a limit to the information of products (can only present product name). Therefore, the choice was to create a custom design which would display a number of products when searching for key words. The query would utilise regular expression queries to find products that are similar to the word, but not completely. This was case sensitive. Once the products were collected, they were put into individual rows, where their respective product image, product name, conditioned, the poster and price could be seen. To select a product a user would need to click on a product name button, which would then take them to the product page.

To explain briefly, this application utilizes object-oriented programming, where classes are essentially menus. In this SearchProduct class, there was two menus. The search for products, and then the product page. Once a user clicks on a product, they will be redirected to the product page, where the name of the product is passed on as a value to find it in the database. It takes only the product name because product names are unique. From this, the product information would be collected and the page would be generated (including the product image which is displayed). As mentioned earlier, on these two pages the "Finnplus" logo would be shown on each of these menus to be able to return to the main starting menu. The benefit of this approach while experimenting showed that there is no need for several pages, and thus saves space in code. The downside to this, is that the approach above with the product search may only return a few products before the program goes off the screen. Two solutions were tried, one being with a scrollbar and the other by setting a limit to how many products can be displayed at a time. Ultimately, the products were limited due to avoiding adding redundant code, and further simplifying the design aspects of the project.

To connect with the shopping cart, a quantity would be taken (this quantity is checked to be an integer), and then once a the add to cart button is clicked, the program checks if a user is logged in. If not, they are redirected to the login menu. Otherwise, if logged in then the product undergoes one of three conditions. These conditions are gone over in the experimentations section, with code provided. On the shopping cart front end, the logo button is visible at the top, with the slogan next to it. On the design aspect of the cart, there are 7 labels which show what piece of information is found underneath. The cart includes the name, user that posted the product, the quantity that the user selected, the price and the calculated subtotal. Furthermore, the number of products is returned and beside it the total price amount which adds up all the subtotals. Beside an items information is an x button which deletes the item from the shopping cart. If their cart is empty, the program will detect this and say "This cart is empty! Go find some products at product search", and then beneath it the search for a product button. Once the user is finished shopping, they may proceed to checkout.

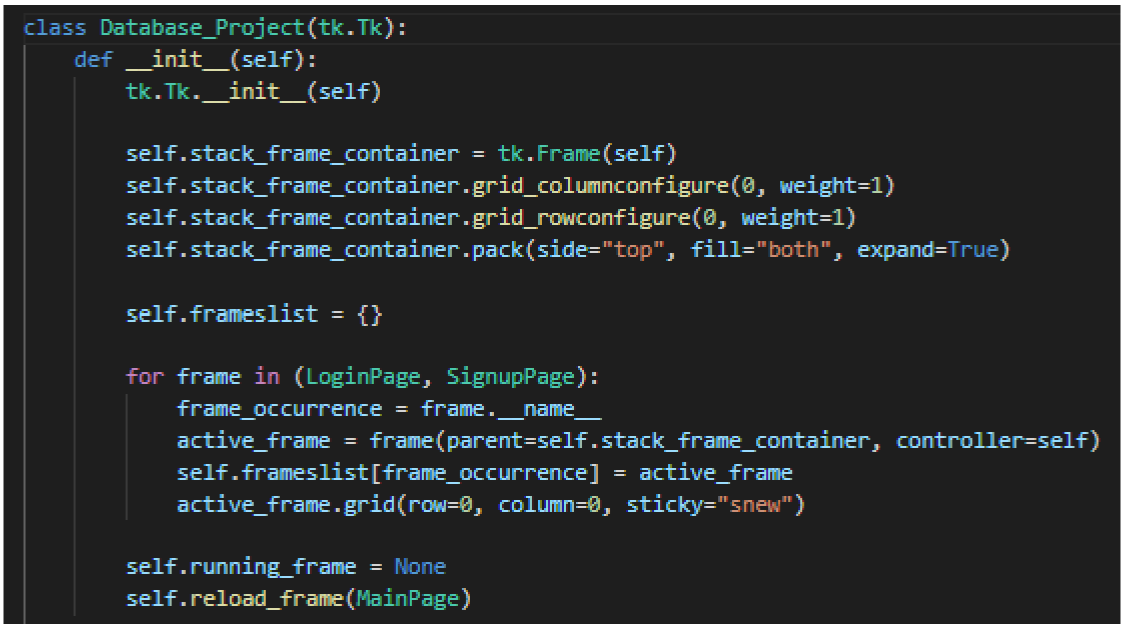
The checkout frontend design remains simple for the reason of simplifying and mainstreaming the process for the user to finish buying items. There is a need for a credit card number, an address, a full name, country and phone. Underneath there are two buttons. A cancel and go back button that returns the user to the shopping cart if they change their mind. If the finalise purchase button is pressed, then the user is notified that the purchase was successful and they are returned to the main menu. This page connects to the shopping cart and orderinformation collections, where it gets information from shopping cart, and the input fields on the application and stores them in orderinformation. A purchase is complete if the entry widgets are all filled, and they are valid values.

This wouldn't be possible without the login and signup pages, as they allow for the selection of the user that is signed in and for the creation of said user. When a user logs in, or is redirected to the login page, they are met with a login section containing two entry fields. There isn't anything complicated about the login or signup page, as they are designed for simplicity's sake. If login is successful, the user signed in is updated. If the user doesn’t have an account, they can click on the "Create New Account". Here they can create an account with a user and an email, both can be used to login. Hence, these two values need to be unique to avoid complications. At any time, the user may go back by clicking "Cancel and go back" which will redirect them to the main starting menu.

In essence, each page is connected through some button that causes an action, which does something and then redirects the user. How these redirections work, how each page is created specifically, and how menus stay updated is another aspect to discuss. To sumarisee, this application is object oriented. Communication occurs through classes, and objects. Further discussion of this can be found in the experimentation section below, with code included for better understanding.

Experimentation

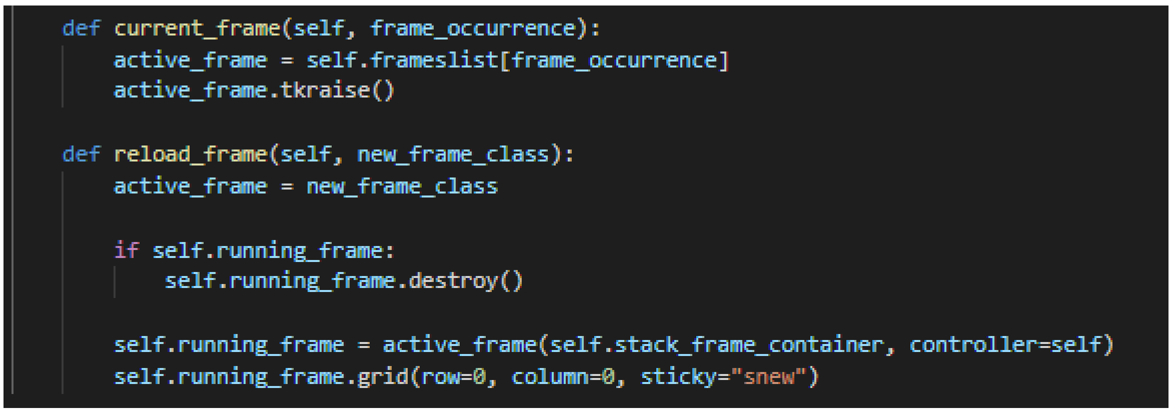
When originally starting with this project, functions were used because of the simplicity behind them. However, it was quickly realised that moving between frames would not be such an easy task. Hence, there were other ways of exploring how to allow these different frames to go between each other. The main problem was, one frame could lead to many, but then to go back would require several functions for each frame individually. Eventually, it was decided that classes would be used for the entirety of the project. The app launches into the initiating function, here is where all the classes were first created before the actual application was visible. This was accomplished by creating a container which would be used to essentially stack frames on top of each other, and then to display the current one, the frame would be raised to the top. A for loop is utilized, and takes the classes as arguments. A frame name is generated from the actual class name, then it is passed two values. Self.stack_frame_container and controller, all classes require this argument as it is what communicates between this main function and the rest. (Oakley, 2011)



```
class Database_Project(tk.Tk):
    def __init__(self):
        tk.Tk.__init__(self)

        self.stack_frame_container = tk.Frame(self)
        self.stack_frame_container.grid_columnconfigure(0, weight=1)
        self.stack_frame_container.grid_rowconfigure(0, weight=1)
        self.stack_frame_container.pack(side="top", fill="both", expand=True)

        self.frameslist = {}

        for frame in (LoginPage, SignupPage):
            frame_occurrence = frame.__name__
            active_frame = frame(parent=self.stack_frame_container, controller=self)
            self.frameslist[frame_occurrence] = active_frame
            active_frame.grid(row=0, column=0, sticky="snew")

        self.running_frame = None
        self.reload_frame(MainPage)
```

Figure 2 – How the frames are created from classes

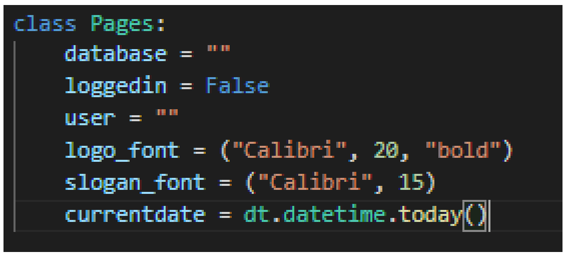
This was the first version of how the program would be created and launched, however this had its negative factors. Whenever a page required updating, this method would not work. Because it's basically putting frames above each other, not creating new instances. That is why two methods were created. One called "current_frame" and another called "reload_frame". The current frame method would raise a frame, without reloading it. This is done by calling the dictionary with the frame name. On the other hand, the reload frame checks if the frame is running, and then destroys is. Similar to a process later in the code, an aspect is taken and destroyed, and then created again in order to be uploaded. See two methods below. (Oakley, 2012)



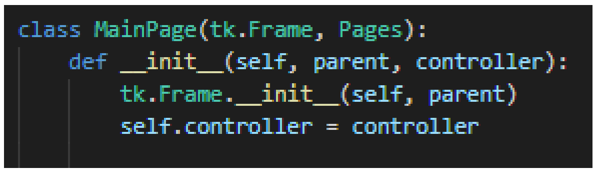
```
def current_frame(self, frame_occurrence):
    active_frame = self.frameslist[frame_occurrence]
    active_frame.tkraise()

def reload_frame(self, new_frame_class):
    active_frame = new_frame_class

    if self.running_frame:
        self.running_frame.destroy()

    self.running_frame = active_frame(self.stack_frame_container, controller=self)
    self.running_frame.grid(row=0, column=0, sticky="snew")
```

Figure 3 – current_frame and reload_frame methods

Once the creation of frames began, and there were a few existing there was another issue. The main page was created, and it would lead to several different pages. Some of which would share the same data between them. For a simple example, there is a login button that changes whether a user is logged in or not. This means that the program and each class must know somehow, whether this is true or not. One way this was tried was to pass these values to each class from "Database_Project", however this did not work. The conclusion to this problem, was to use inherited values. An entire separate class, with no relation to tkinter would be created, where whenever they are required, the class can retrieve them because they are inherited.



```
class Pages:
    database = ""
    loggedin = False
    user = ""
    logo_font = ("Calibri", 20, "bold")
    slogan_font = ("Calibri", 15)
    currentdate = dt.datetime.today()
```

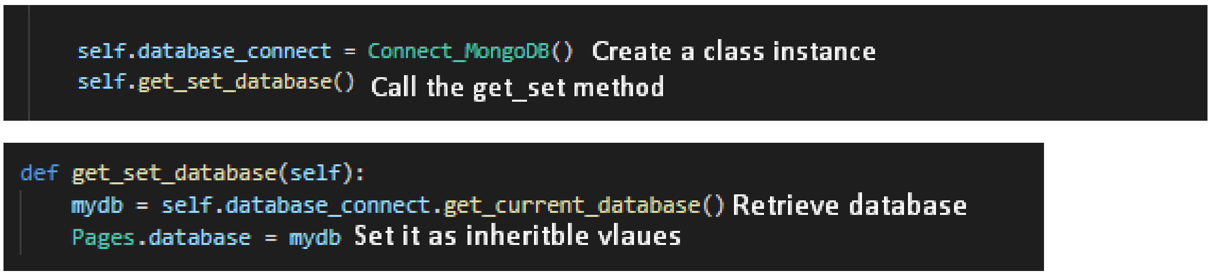


```
class MainPage(tk.Frame, Pages):
    def __init__(self, parent, controller):
        tk.Frame.__init__(self, parent)
        self.controller = controller
```

Figure 4 – Pages class, inherited by all frames

Figure 5 – How the Pages class is inherited

Next was the implementation and connection to the database. There was experimentation with connection to the database before initiating the loop, but then the database could not be retrieved. Another was to try to call the class whenever the database was to be retrieved, this done by getters and setters, and through the use of properties. However, this would cause the entire window to stall because the database would be constantly running. It was not optimal. The solution was to have the database only called once, set it at the main page and then set it inside the class Pages so it can be accessed by other classes as well. See below.

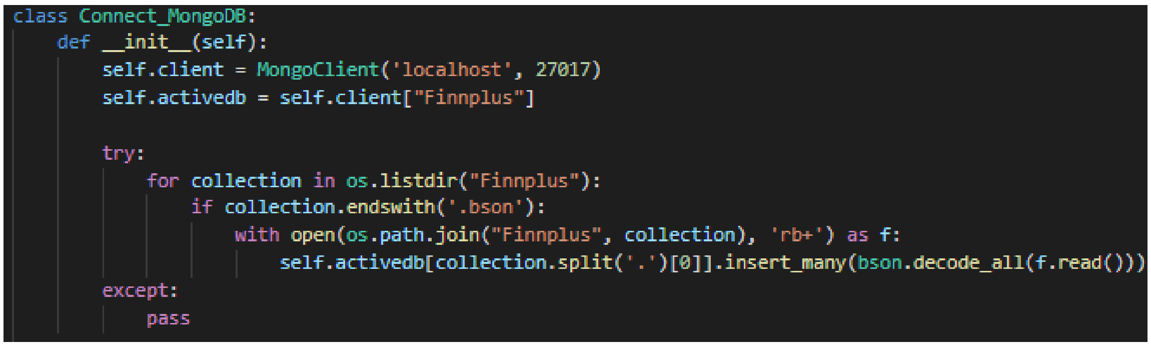


```
self.database_connect = Connect_MongoDB()  Create a class instance
self.get_set_database()  Call the get_set method
```

```
def get_set_database(self):
    mydb = self.database_connect.get_current_database() Retrieve database
    Pages.database = mydb  Set it as inheritble vlaues
```

Figure 6 - Inside MainPage class

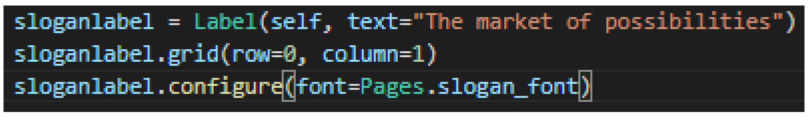
However, because this application must run completely, without any further input, the database requires to be somehow operable, without the need of manual importing. The way this was done was through the use of the os, bson libraries. A try statement was used to determine whether the database already exists or not. If not, then go through all files in the folder, return those that end with ".bson". Proceed to open the file, and create a buffer instance to be able to read and write the given files. Then splice the string, to only include the actual collection name with "collection.split('.')[0]". Read over the file, and insert all data into the collections. (Lh4cKg, 2019) If the database does exist, then the program should carry on as usually, and simply pass this step.



```
class Connect_MongoDB:
    def __init__(self):
        self.client = MongoClient('localhost', 27017)
        self.activedb = self.client["Finnplus"]

        try:
            for collection in os.listdir("Finnplus"):
                if collection.endswith('.bson'):
                    with open(os.path.join("Finnplus", collection), 'rb+') as f:
                        self.activedb[collection.split('.')[0]].insert_many(bson.decode_all(f.read()))
        except:
            pass
```

Figure 7 - Inside the database class:

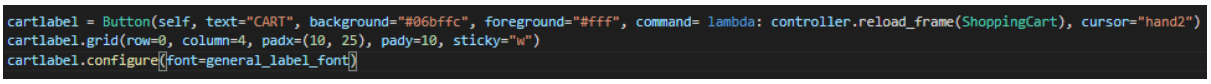
This was all mostly done to connect to the backend of the database, while also ensuring the front end's basic design and background structure. What the user would see was now started to develop. A large portion of the actual code is labels, buttons and entries. The reason for this is that each label would take three lines. Originally, this wasn't the case because the ".grid" was attached onto the label creation. However, in order to updated and configure a labels text with a different font, size and weight it needs to be separate or else it will recognize the label as a "NoneType object" and will not detect what configure means. The way in which methods and different classes are called is through the "command" argument.



```
sloganlabel = Label(self, text="The market of possibilities")
sloganlabel.grid(row=0, column=1)
sloganlabel.configure(font=Pages.slogan_font)
```

Figure 8 – Typical label and its configuration

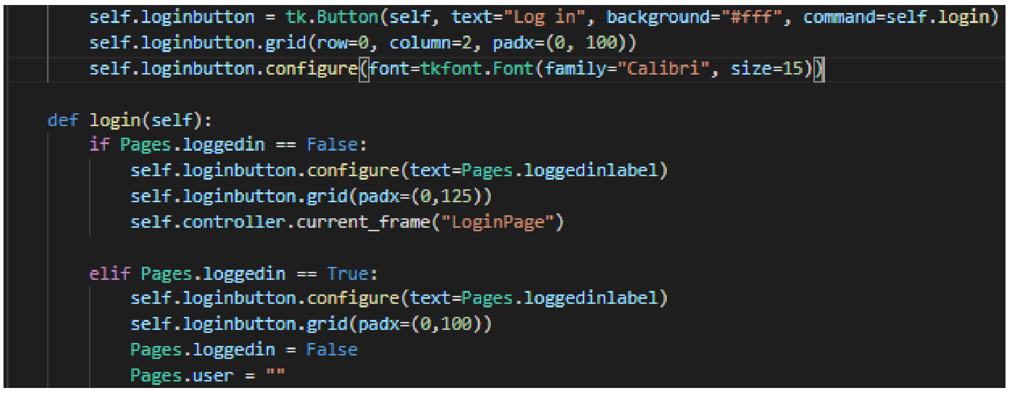
Through this argument, what needed to be done was that a function had to be called, and then parameters passed through this function. At some points, a method within the class was called to carry out additional things before the redirection occurred. Other times, the redirection was within the command clause. Keeping in mind that the in the initiating class controller was equal to self. Meaning that controller references that class, and its methods. Therefore, to get the change frame or reload the frame, "controller.reload_Frame()" is used with the class as the parameter. One interesting thing to note here was that this did not work alone, it was not completely functional. The reason for this being that it would go to that frame, without the click of a button. This is corrected through the use of a lambda, usually only required when referring to the parent frame. See figure 9 for code.



```
cartlabel = Button(self, text="CART", background="#06bffc", foreground="#fff", command= lambda: controller.reload_frame(ShoppingCart), cursor="hand2")
cartlabel.grid(row=0, column=4, padx=(10, 25), pady=10, sticky="w")
cartlabel.configure(font=general_label_font)
```

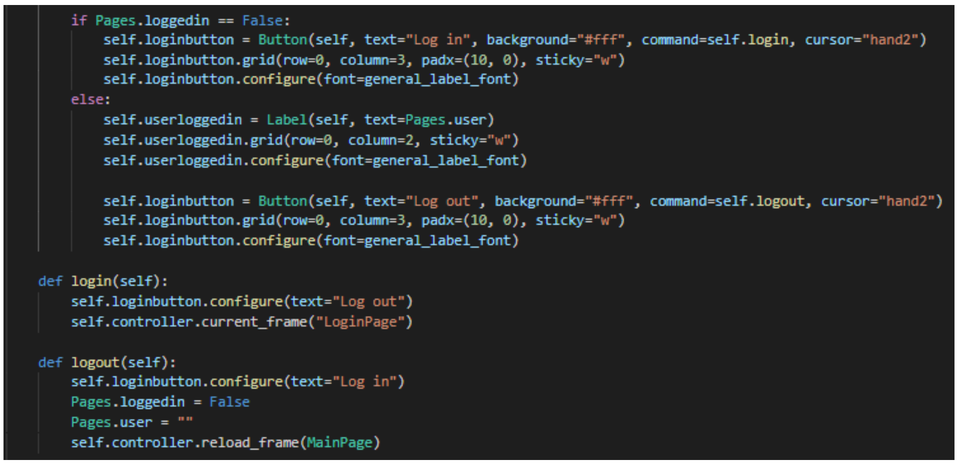
Figure 9 – Calling methods in the initiating Database_Project class

The final piece of code that is shared among some classes is the login feature. The login button can be found in more than one class. Originally, what was attempted was to update the value of the button once it was clicked. Meaning if logged in, then change to log out. If logged out then log in. The only prominent issue with this, if the user clicked to login, but changed their mind then the button would still be updated. The theory behind this was to use the inherited "Pages.loggedin" value to determine this. Experimentation with this occurred, and a way to fix this problem was to check at the start of the frame if the user is logged in or out, and then determine the appropriate action if clicked.



```
        self.loginbutton = tk.Button(self, text="Log in", background="#fff", command=self.login)
        self.loginbutton.grid(row=0, column=2, padx=(0, 100))
        self.loginbutton.configure(font=tkfont.Font(family="Calibri", size=15))

    def login(self):
        if Pages.loggedin == False:
            self.loginbutton.configure(text=Pages.loggedinlabel)
            self.loginbutton.grid(padx=(0,125))
            self.controller.current_frame("LoginPage")

        elif Pages.loggedin == True:
            self.loginbutton.configure(text=Pages.loggedinlabel)
            self.loginbutton.grid(padx=(0,100))
            Pages.loggedin = False
            Pages.user = ""
```

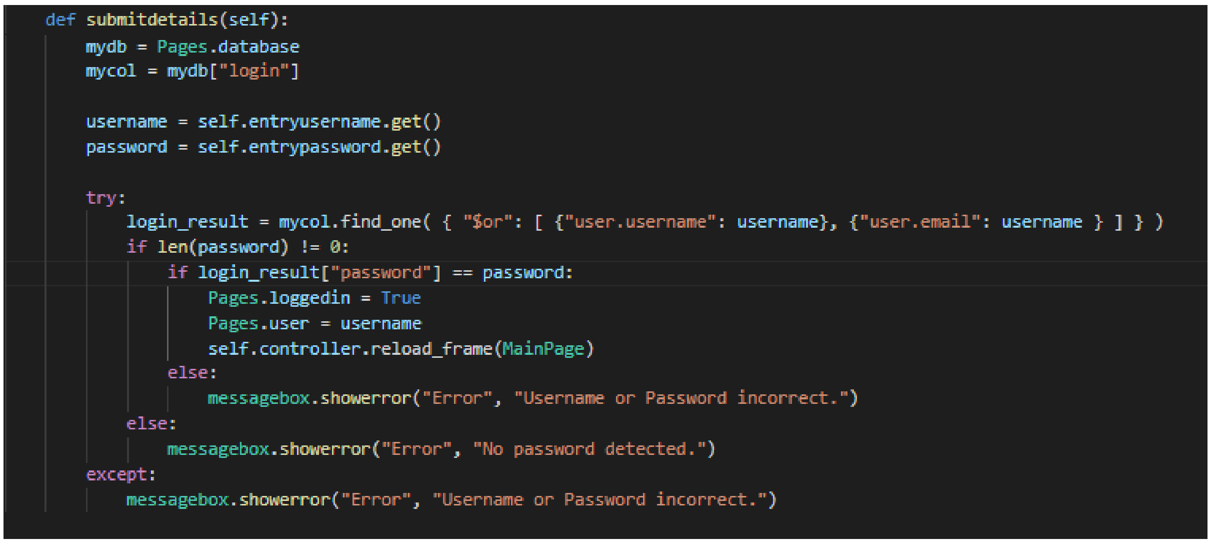
Figure 10 – First implementation of a login/logout button



```python
        if Pages.loggedin == False:
            self.loginbutton = Button(self, text="Log in", background="#fff", command=self.login, cursor="hand2")
            self.loginbutton.grid(row=0, column=3, padx=(10, 0), sticky="w")
            self.loginbutton.configure(font=general_label_font)
        else:
            self.userloggedin = Label(self, text=Pages.user)
            self.userloggedin.grid(row=0, column=2, sticky="w")
            self.userloggedin.configure(font=general_label_font)

            self.loginbutton = Button(self, text="Log out", background="#fff", command=self.logout, cursor="hand2")
            self.loginbutton.grid(row=0, column=3, padx=(10, 0), sticky="w")
            self.loginbutton.configure(font=general_label_font)

    def login(self):
        self.loginbutton.configure(text="Log out")
        self.controller.current_frame("LoginPage")

    def logout(self):
        self.loginbutton.configure(text="Log in")
        Pages.loggedin = False
        Pages.user = ""
        self.controller.reload_frame(MainPage)
```

Figure 11 – Final implementation of a login/logout button

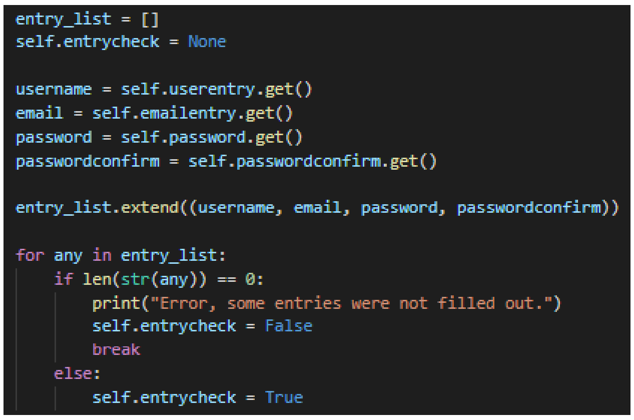
In the login class, there would be two entry fields. Followed by buttons. The entries related to the username and password. Once the button "Login" was clicked, the submitdetails method was called. The code would use a try and except to use the values entered to connect to the database collection "login", where it would check for existing user (by username or email) and then additionally check if the password was not empty. If the password is correct, update Pages.user and Pages.loggedin values and then redirect to MainPage. Otherwise, return an error to notify the user what was wrong.



```python
    def submitdetails(self):
        mydb = Pages.database
        mycol = mydb["login"]

        username = self.entryusername.get()
        password = self.entrypassword.get()

        try:
            login_result = mycol.find_one( { "$or": [ {"user.username": username}, {"user.email": username } ] } )
            if len(password) != 0:
                if login_result["password"] == password:
                    Pages.loggedin = True
                    Pages.user = username
                    self.controller.reload_frame(MainPage)
                else:
                    messagebox.showerror("Error", "Username or Password incorrect.")
            else:
                messagebox.showerror("Error", "No password detected.")
        except:
            messagebox.showerror("Error", "Username or Password incorrect.")
```

Figure 12 – User login

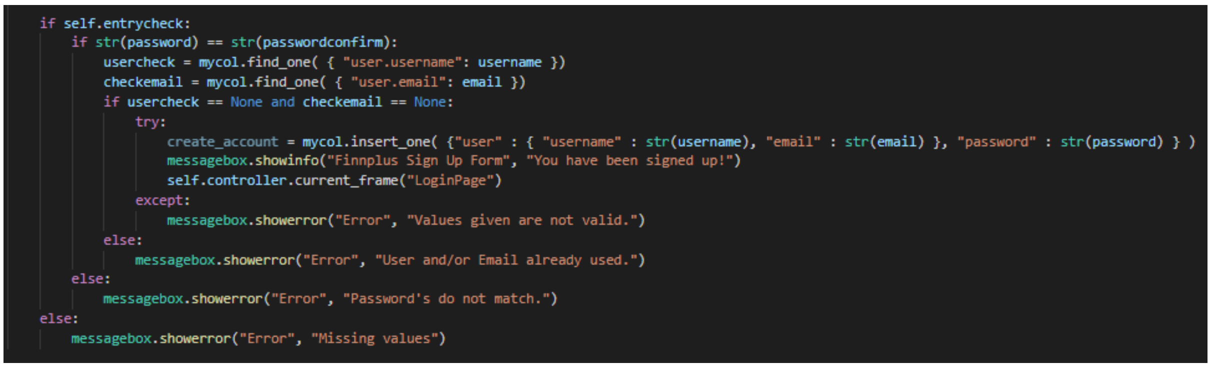
When a user attempts to sign up, they will be prompted to enter a user, email and password, to make sure the fields are not empty, all these entry widget values will be put into a list, and a Boolean value created, so that they can be checked to not be empty. This is done by checking the length of the string, if its 0 return False, if everything is True then carry on.



```python
entry_list = []
self.entrycheck = None

username = self.userentry.get()
email = self.emailentry.get()
password = self.password.get()
passwordconfirm = self.passwordconfirm.get()

entry_list.extend((username, email, password, passwordconfirm))

for any in entry_list:
    if len(str(any)) == 0:
        print("Error, some entries were not filled out.")
        self.entrycheck = False
        break
    else:
        self.entrycheck = True
```

Figure 13 – Common way across classes to check for empty fields

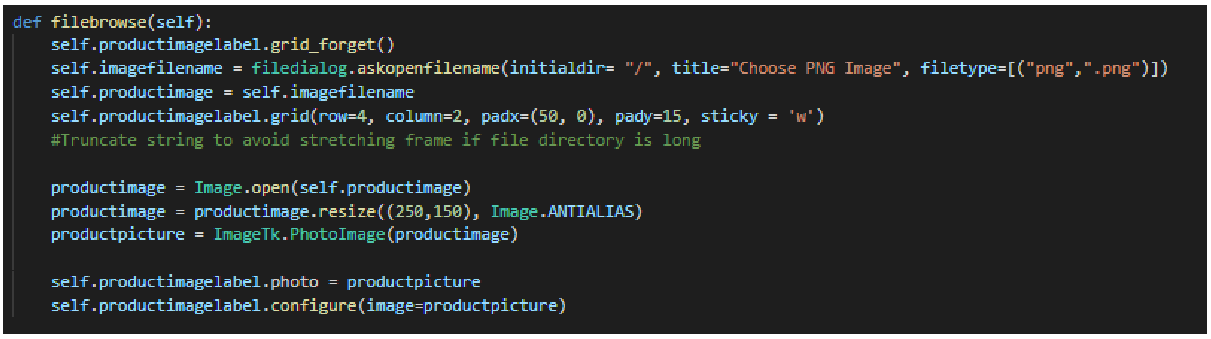
Once the conditional is passed, the collection is used to check whether the user or email values exist, because these need to be unique. It is crucial for the rest of the program, and hence all these checks must be put into place. If it is unique the query will return None, and therefore can be created. Errors and information are display through messageboxes. If successful the user will be notified through a "showinfo" messagebox, otherwise it will display a "showerror" messagebox. The figure below is a common procedure that many of the classes follow, with little alteration (mostly just to the database queries). These differences are highlighted, but generally follow the same theory for error detection.



```python
if self.entrycheck:
    if str(password) == str(passwordconfirm):
        usercheck = mycol.find_one( { "user.username": username })
        checkemail = mycol.find_one( { "user.email": email })
        if usercheck == None and checkemail == None:
            try:
                create_account = mycol.insert_one( {"user" : { "username" : str(username), "email" : str(email) }, "password" : str(password) } )
                messagebox.showinfo("Finnplus Sign Up Form", "You have been signed up!")
                self.controller.current_frame("LoginPage")
            except:
                messagebox.showerror("Error", "Values given are not valid.")
        else:
            messagebox.showerror("Error", "User and/or Email already used.")
    else:
        messagebox.showerror("Error", "Password's do not match.")
else:
    messagebox.showerror("Error", "Missing values")
```

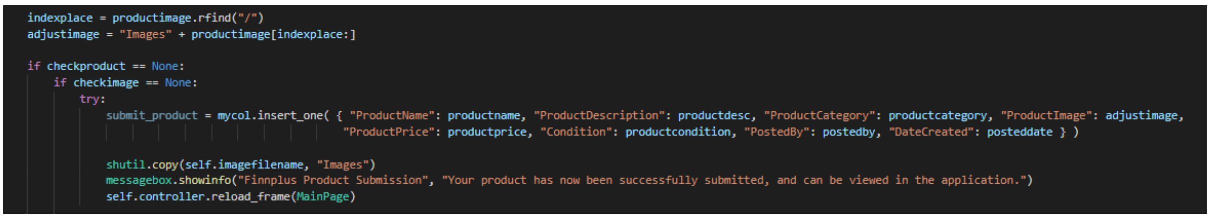
Figure 14 – Signup page error detection, and read and write MongoDB queries

One of the more complicated aspects of this project was images. Although several pages and videos showed how to implement images fairly simply, it was challenging to get them adjusted and working with the program. It was creating a label, and then setting the image to it. (Codemy.com, 2020) Understanding how an image is implemented was important, but understanding why sometimes it didn't display was crucial. Resizing an image was required, and needed additional steps to be taken, this involved importing the library pillow and then using it to resize images in order to make the images all be able to fit on the program and keep it organised. (Tutorialspoint) However, this would still cause issues. An image needed to be saved in the database, and an image was found through a directory. This meant that a user had to choose an image, and the program needed to know where that image was. Although some thought went into it, it was decided in the end to let the user browse for a file, and then move that file into the project's own directory. This was done by importing shutil. (Hussain, 2020)



```
def filebrowse(self):
    self.productimagelabel.grid_forget()
    self.imagefilename = filedialog.askopenfilename(initialdir= "/", title="Choose PNG Image", filetype=[("png",".png")])
    self.productimage = self.imagefilename
    self.productimagelabel.grid(row=4, column=2, padx=(50, 0), pady=15, sticky = 'w')
    #Truncate string to avoid stretching frame if file directory is long

    productimage = Image.open(self.productimage)
    productimage = productimage.resize((250,150), Image.ANTIALIAS)
    productpicture = ImageTk.PhotoImage(productimage)

    self.productimagelabel.photo = productpicture
    self.productimagelabel.configure(image=productpicture)
```

Figure 15 – Letting the user browse the file – "filedialog.askopenfilename()"

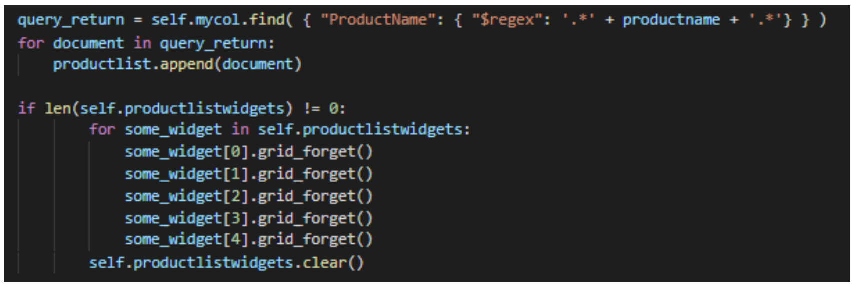


```
    indexplace = productimage.rfind("/")
    adjustimage = "Images" + productimage[indexplace:]

    if checkproduct == None:
        if checkimage == None:
            try:
                submit_product = mycol.insert_one( { "ProductName": productname, "ProductDescription": productdesc, "ProductCategory": productcategory, "ProductImage": adjustimage,
                                                    "ProductPrice": productprice, "Condition": productcondition, "PostedBy": postedby, "DateCreated": posteddate } )

                shutil.copy(self.imagefilename, "Images")
                messagebox.showinfo("Finnplus Product Submission", "Your product has now been successfully submitted, and can be viewed in the application.")
                self.controller.reload_frame(MainPage)
```

Figure 16 – Moving image file to Images folder – "shutil.copy()"

This way, regardless of where the image was taken from, once the product was submitted, the file would be moved into the image's directory. The additional benefit of this is that regardless of where the code of this program is located, as long as there is an images folder in the same directory, it will be able to store and retrieve images. Looking closely at Figure 16, indexplace returns the productimage string, with the last /. This collects only the image file name, and then adds Images, to make sure it is set to the images directory to be later retrieved.

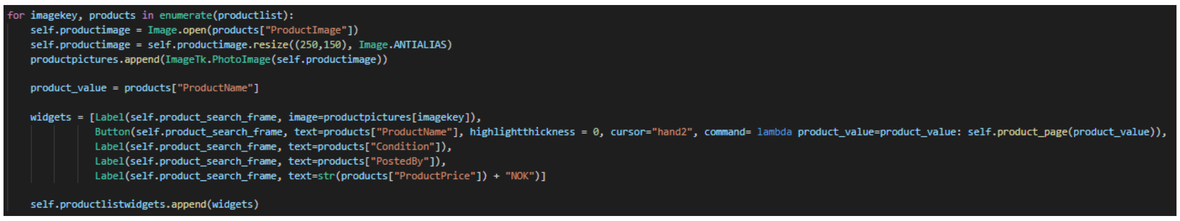
When it came to the product search, various information needed to be retrieved including the images mentioned above. However, the design behind this required for there to be two frames. One which would search through the products and list them, and another to function as a page and display information on that product. This meant that the two frames would need to interact, while also be erased. When a product is clicked on, then the search frame should disappear, and vice-versa for clicking on the search feature. This was done through the grid_forget() at the start of a method. Regarding actual queries, it was decided that when a user searches for a product, they should be able to search by a word that is like the product, but not exact. This was implemented through a regular expression in the database query. Furthermore, if a search was redone then the last list should disappear. This was done through a for loop and checking if there are any widgets in a list.



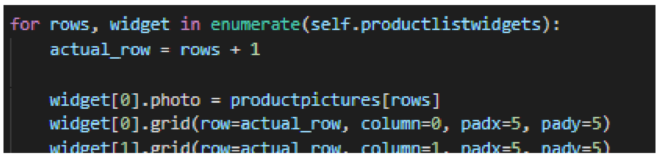
```
query_return = self.mycol.find( { "ProductName": { "$regex": '.*' + productname + '.*'} } )
for document in query_return:
    productlist.append(document)

if len(self.productlistwidgets) != 0:
        for some_widget in self.productlistwidgets:
            some_widget[0].grid_forget()
            some_widget[1].grid_forget()
            some_widget[2].grid_forget()
            some_widget[3].grid_forget()
            some_widget[4].grid_forget()
        self.productlistwidgets.clear()
```

Figure 17 – Displaying how searches work, and how they are rewritten

One of the crucial parts in this, is when an image was implemented in a for loop, only the latest one would be called, or it would not be displayed at all. This meant that each image must be defined separately. This was achieved through a for loop and append them to a list. So, they can later be called through an image key in an enumerate statement. The images needed to be displayed and this was achieved by anchoring the image to the object. (Avión, 2017; Vermeulen, 2019)



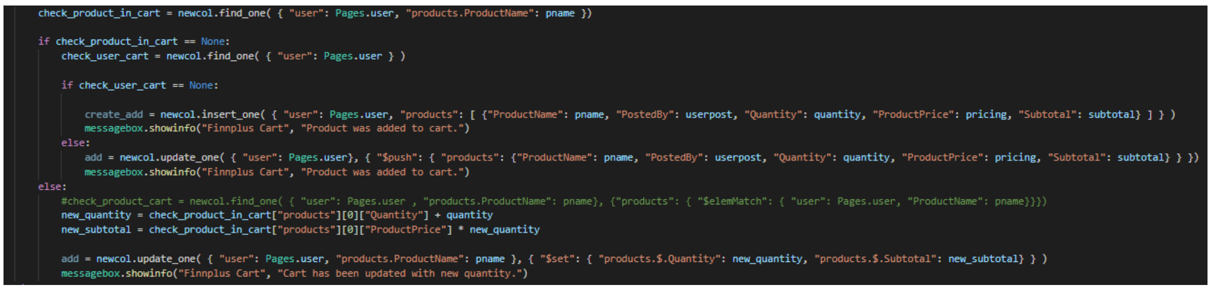
```
for imagekey, products in enumerate(productlist):
    self.productimage = Image.open(products["ProductImage"])
    self.productimage = self.productimage.resize((250,150), Image.ANTIALIAS)
    productpictures.append(ImageTk.PhotoImage(self.productimage))

    product_value = products["ProductName"]

    widgets = [Label(self.product_search_frame, image=productpictures[imagekey]),
               Button(self.product_search_frame, text=products["ProductName"], highlightthickness = 0, cursor="hand2", command= lambda product_value=product_value: self.product_page(product_value)),
               Label(self.product_search_frame, text=products["Condition"]),
               Label(self.product_search_frame, text=products["PostedBy"]),
               Label(self.product_search_frame, text=str(products["ProductPrice"]) + "NOK")]

    self.productlistwidgets.append(widgets)
```



```
for rows, widget in enumerate(self.productlistwidgets):
    actual_row = rows + 1

    widget[0].photo = productpictures[rows]
    widget[0].grid(row=actual_row, column=0, padx=5, pady=5)
    widget[1].grid(row=actual_row, column=1, padx=5, pady=5)
```

Figure 18 – Product list creation, image creation, image display and anchoring

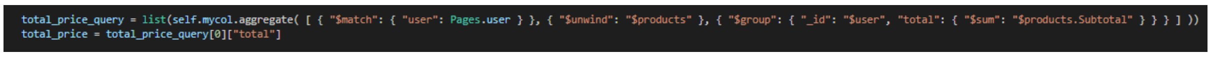
A very difficult component to implement was when a product was added to the cart. Due to having an array of embedded documents, this made the query and database theory more difficult, but more organised and individual to each user. There were three premises; Create a shopping cart for user if there isn't one, update the shopping cart if product isn't there and if it is, update its quantity. When updating the quantity, it would always remove the rest of the products. It was figured out that a "$" must be placed between products field and the embedded document's field in order to retrieve that occurrence and then update. (Murakami, 2021) See code below.



```
    check_product_in_cart = newcol.find_one( { "user": Pages.user, "products.ProductName": pname })

    if check_product_in_cart == None:
        check_user_cart = newcol.find_one( { "user": Pages.user } )

        if check_user_cart == None:

            create_add = newcol.insert_one( { "user": Pages.user, "products": [ {"ProductName": pname, "PostedBy": userpost, "Quantity": quantity, "ProductPrice": pricing, "Subtotal": subtotal} ] } )
            messagebox.showinfo("Finnplus Cart", "Product was added to cart.")
        else:
            add = newcol.update_one( { "user": Pages.user}, { "$push": { "products": {"ProductName": pname, "PostedBy": userpost, "Quantity": quantity, "ProductPrice": pricing, "Subtotal": subtotal} } })
            messagebox.showinfo("Finnplus Cart", "Product was added to cart.")
    else:
        #check_product_cart = newcol.find_one( { "user": Pages.user , "products.ProductName": pname}, {"products": { "$elemMatch": { "user": Pages.user, "ProductName": pname}}})
        new_quantity = check_product_in_cart["products"][0]["Quantity"] + quantity
        new_subtotal = check_product_in_cart["products"][0]["ProductPrice"] * new_quantity

        add = newcol.update_one( { "user": Pages.user, "products.ProductName": pname }, { "$set": { "products.$.Quantity": new_quantity, "products.$.Subtotal": new_subtotal} } )
        messagebox.showinfo("Finnplus Cart", "Cart has been updated with new quantity.")
```

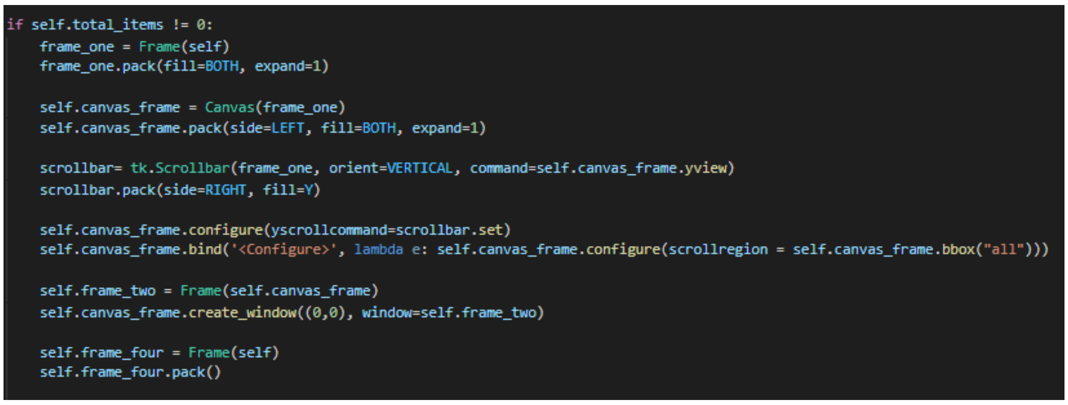
Figure 19 – Code for creating shopping carts, adding and updating products

Some of the last features to highlight was the shopping cart frame. Similarly, to other frames, there are various labels created through the use of a for loop. In this page, all the products that were added to the cart are shown by iterating through the shopping cart collection of the user. To note, use of aggregation allowed for a return of the total price of all the products in that cart combined. This meant iterating through the array of embedded documents, and summing them. First a user was matched against the collection's documents, and the products field was unwound to allow aggregating through it. The query was grouped by the user and then also grouped the "total" by the sum of all product subtotals. Since it returns a user and the product list, "[0]" is used because the query returns a list of results. See code below.



```
total_price_query = list(self.mycol.aggregate( [ { "$match": { "user": Pages.user } }, { "$unwind": "$products" }, { "$group": { "_id": "$user", "total": { "$sum": "$products.Subtotal" } } } ] ))
total_price = total_price_query[0]["total"]
```

One thing considered was that if too many products are added then the frame would be too large for the screen. A scrollbar was considered for this section and experimented with. The scrollbar was functional, but was opted to be removed because of the clunky functionality and aesthetic. Additionally, it overcomplicated the design of the frame. More than four frames were required and mixed with the pack() placement method, it did not return the wanted results. See experimented code below.



```
if self.total_items != 0:
    frame_one = Frame(self)
    frame_one.pack(fill=BOTH, expand=1)

    self.canvas_frame = Canvas(frame_one)
    self.canvas_frame.pack(side=LEFT, fill=BOTH, expand=1)

    scrollbar= tk.Scrollbar(frame_one, orient=VERTICAL, command=self.canvas_frame.yview)
    scrollbar.pack(side=RIGHT, fill=Y)

    self.canvas_frame.configure(yscrollcommand=scrollbar.set)
    self.canvas_frame.bind('<Configure>', lambda e: self.canvas_frame.configure(scrollregion = self.canvas_frame.bbox("all")))

    self.frame_two = Frame(self.canvas_frame)
    self.canvas_frame.create_window((0,0), window=self.frame_two)

    self.frame_four = Frame(self)
    self.frame_four.pack()
```

Figure 20 – Experimenting with a scrollbar feature

Additionally, what was wanted is that if the cart is empty then the user that is logged in should not be met with a blank space. There had to be a way to tell them that the cart is empty. Hence an if statement is used that checks the length of the cart. If cart doesn't exist, redirect user to different frame with label to search for products and the search products button. If the products document is empty, then the same thing should happen.

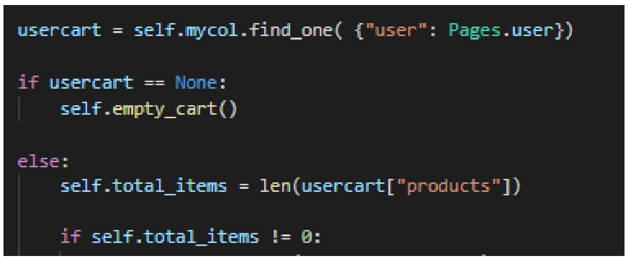

```
usercart = self.mycol.find_one( {"user": Pages.user})

if usercart == None:
    self.empty_cart()

else:
    self.total_items = len(usercart["products"])

    if self.total_items != 0:
```

Figure 21 – Determining if the cart is empty, filled or does not exist

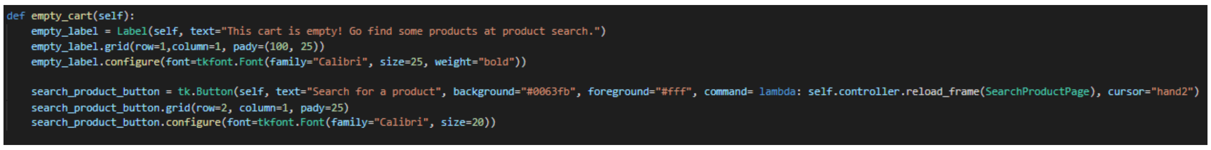

```
def empty_cart(self):
    empty_label = Label(self, text="This cart is empty! Go find some products at product search.")
    empty_label.grid(row=1,column=1, pady=(100, 25))
    empty_label.configure(font=tkfont.Font(family="Calibri", size=25, weight="bold"))

    search_product_button = tk.Button(self, text="Search for a product", background="#0063fb", foreground="#fff", command= lambda: self.controller.reload_frame(SearchProductPage), cursor="hand2")
    search_product_button.grid(row=2, column=1, pady=25)
    search_product_button.configure(font=tkfont.Font(family="Calibri", size=20))
```

Figure 22 – Showing what method is called if cart doesn't exist, or is empty

Lastly, one feature of the cart was how to implement a way to remove products. A needed aspect of the program was to have a way to remove products from the cart. This is where the benefit of creating the complex collection design is most prominent. By having the shopping cart tied to the user, a product's name can be passed as a parameter to a method and then using that, query through the database to find that exact value and pull it from the document. Each product would have its own button to be removed. As mentioned, a user shouldn't see an empty cart, therefore if there are no products, then the entire cart should be removed, and the frame reloaded to show the correct labels.

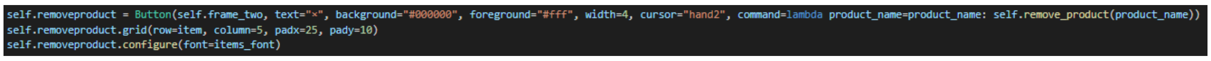

```
self.removeproduct = Button(self.frame_two, text="×", background="#000000", foreground="#fff", width=4, cursor="hand2", command=lambda product_name=product_name: self.remove_product(product_name))
self.removeproduct.grid(row=item, column=5, padx=25, pady=10)
self.removeproduct.configure(font=items_font)
```

Figure 23 – Passing the product name as a parameter to be removed through the click of a button

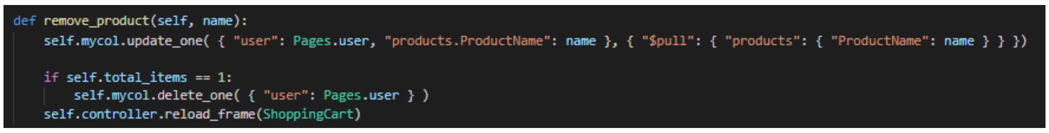

```
def remove_product(self, name):
    self.mycol.update_one( { "user": Pages.user, "products.ProductName": name }, { "$pull": { "products": { "ProductName": name } } })

    if self.total_items == 1:
        self.mycol.delete_one( { "user": Pages.user } )
    self.controller.reload_frame(ShoppingCart)
```

Figure 24 – Method to removing the product by using an update query

This was the summary of all the important features, and experimentation that went into writing the code of this e-commerce application. There can still be several improvements and additional features, however the main purpose of this section was to highlight how the application functioned through code and the use of classes to carry over information and connect different frames together.

Conclusion

To summarise, the task of this project was to create some form of e-commerce application. This was to create a system for a start-up business. This application was to work with and imitate a portal in which customers are able to interact with and order several products and goods. The core was to have the application handle online transactions. This was done through the designers own choosing, in regards to both frontend and backend development. Furthermore, a form of model was required to show the data model design. This mostly being, for the backend database implementation. To begin with, a Hackolade database design model was presented to summarise the various collections, and how documents within them are structured and organised. The Finnplus database had four collections, the login info of users, the products information, data on the shopping cart and the order information when someone checks out.

The collections share common values between themselves, which differs from a traditional approach of embedded documents or referencing relationships. The underlying foundation of the program, and utilizing client-side interaction allows for collection to share these common fields and values. Through this method of storing and retrieving information, communication between these collections is dependent on the design of the frontend and how individual frames inside the program are created. Introducing the frontend aspect of the project. The frontend part of the e-commerce application was put together through the use of Python and the library Tkinter. There were 6 characteristics of the application that were mandatory. These being posting a product, searching for a product, displaying an image of a product, adding products to a cart, removing products from a cart and a checkout section. Each of these mandatory features were split into individual frames in order to ensure data consistency.

However, each of these frames were created separately through the use of classes and objects. The application was centred around object-oriented programming. Through this, each frame could have its own instance and area to allow for greater capabilities in organisation and structure of the frontend part. Two methods were used to create these frames; either the frame will put into a stack where the running frame will be raised to the top of the stack to be viewed, or the frame will be reloaded in order to allow for modification and updating of labels, buttons and entries. Each class/frame had its own process to follow that allowed for the necessary features to be carried out, furthermore providing additional features such as a login/signup page and individual product pages. In essence, all these components mentioned above, and detail-oriented thought behind coding allowed for the successful completion of the project; presenting an e-commerce-based application to allow and simulate online transactions.

References

Avión. (2017, November 6). Tkinter image is blank. Stack Overflow. Retrieved October 23, 2021, from https://stackoverflow.com/questions/47138691/tkinter-image-is-blank

Bruce, D. (2021, May 28). Understanding the pros and cons of MongoDB. knowledgenile. Retrieved September 16, 2021, from https://www.knowledgenile.com/blogs/pros-and-cons-of-mongodb/

Codemy.com. (2020, November 24). How to reload the frame in tkinter everytime switching the frame? YouTube. Retrieved October 23, 2021, from https://www.youtube.com/watch?v=WurCpmHtQc4

Hussain, M. (2020, June 10). How do I upload an image on Python using Tkinter? Stack Overflow. Retrieved October 23, 2021, from https://stackoverflow.com/questions/42600739/how-do-i-upload-an-image-on-python-using-tkinter

Lh4cKg. (2019). MongoDB dump and restore database with Python PyMongo driver. GitHub Gist. Retrieved October 24, 2021, from https://gist.github.com/Lh4cKg/939ce683e2876b314a205b3f8c6e8e9d

Murakami, G. (2012, June 8). Remove embedded document in a nested array of documents. Stack Overflow. Retrieved October 23, 2021, from https://stackoverflow.com/questions/10950451/remove-embedded-document-in-a-nested-array-of-documents

Oakley, B. (2011, September 26). Switch between two frames in tkinter. Stack Overflow. Retrieved October 23, 2021, from https://stackoverflow.com/questions/7546050/switch-between-two-frames-in-tkinter

Oakley, B. (2012, March 12). How to reload the frame in tkinter everytime switching the frame? Stack Overflow. Retrieved October 23, 2021, from https://stackoverflow.com/questions/66600226/how-to-reload-the-frame-in-tkinter-everytime-switching-the-frame

TutorialsPoint. (n.d.). How to resize an image using Tkinter? Tutorialspoint - Simplify Early Learning. https://www.tutorialspoint.com/how-to-resize-an-image-using-tkinter

Vermeulen, B. (2019, October 25). Cannot display more than one images in the canvas of tkinter. Stack Overflow. Retrieved October 23, 2021, from https://stackoverflow.com/questions/58559150/cannot-display-more-than-one-images-in-the-canvas-of-tkinter