

Huy Vo

DESKTOP APPLICATION WITH WINDOWS PRESENTATION FOUNDATION FOR INVENTORY MANAGEMENT

Bachelor's thesis

Information Technology

Bachelor of Engineering

2022



South-Eastern Finland
University of Applied Sciences

Author (authors)	Degree title	Time
Huy Vo	Bachelor of Engineering	May 2022
Thesis title		
Desktop application with Windows Presentation Foundation for Inventory Management		41 pages 0 pages of appendices
Commissioned by		
Not given		
Supervisor		
Timo Mynttinen		
Abstract		
<p>For a long time, individuals and companies have been using pen and paper to record their items in the inventory. To help with this, this thesis aimed to provide a look at developing a desktop application for inventory management. The application was created using Windows Presentation Foundation with Firestore from Google Cloud.</p> <p>The thesis described the tools, language, architectural pattern, framework, and related technologies used to create the application. Moreover, it also discussed the features of the application. Finally, the design and implementation of the application were examined.</p>		
Keywords		
WPF, MVVM, Firebase, Firestore, NoSQL		

CONTENTS

1	INTRODUCTION	4
2	THEORY.....	5
2.1	Desktop application	5
2.2	.NET and .NET Framework	5
2.3	C# programming language	8
2.4	Windows Presentation Foundation	8
2.4.1	Introduction	8
2.4.2	Controls	11
2.4.3	Layout.....	11
2.4.4	Data binding.....	12
2.4.5	Windows	13
2.5	Model-View-ViewModel architectural pattern.....	14
2.5.1	Overview	14
2.5.2	Model-View-ViewModel in WPF.....	16
2.6	Database	16
3	IMPLEMENTATION.....	18
3.1	Design and create data model.....	18
3.2	ViewModel Class Hierarchy	23
3.3	Data and command binding	25
3.4	Background refetching	29
3.5	Sending reorder email	30
3.6	Result	30
4	CONCLUSION.....	37
	REFERENCES	38
	LIST OF FIGURES	40

1 INTRODUCTION

This thesis mainly concentrates on designing, implementing, and developing an inventory management application. Windows Presentation Foundation is used to create desktop user interfaces with .NET Framework. The user interfaces are written in XAML format, and the logic is in C#. Moreover, the Model-View-ViewModel architectural pattern is being applied to the application, which, according to Wikipedia (Model-view-viewmodel 2022), helps separate business logic from graphical interfaces. Every View inside the application will be connected to a ViewModel, which handles binding, for example, to commands and variables.

The application is also directly connected to a cloud database. This architecture will provide a more flexible and straightforward way to develop, deploy and scale “On Demand” horizontally based on what we need (Chandra et al. 2012, 513). It can also help the backup and migration processes become much more manageable.

After discussing the technologies used, it is also essential to see how the application was created. The purpose of this desktop application is to apply those techniques in solving real-life problems, and inventory management is one of them. For a long time, individuals and companies have been using pen and paper to record their items in the inventory. This method is easy, but mistakes are easily made. There are already many products or similar applications on the market; however, most of them cost a lot of money, which is not affordable and efficient for small businesses and users. This application implements some of the core functionalities of those on the market and provides them for free. Moreover, the application includes user interfaces to create and update items and orders. There will be one Window object, and one ContentControl object acts as a homepage for the application so that when a user navigates between applications, the ContentControl will bind with different UserControl.

2 THEORY

This section describes the .NET framework, C# programming language, and Windows Presentation Foundation. They are essential components for creating a desktop application. Information about cloud databases is also studied.

2.1 Desktop application

Nowadays, any service can be accessed using mobile, desktop, or web applications. A mobile application is an application that is designed to run on a phone, tablet or watch. However, publishing and getting the application listed on trusted distribution platforms are also challenging. Publishing mobile applications on Apple's App Store or Google Play Store can cost money. Furthermore, if the applications violate the law or regulation in some countries, they will not be listed on distribution platforms in those countries. A web application is an application that runs remotely on a web server. Still, it can only be accessed using a web browser with an active internet connection. Furthermore, the web applications need to always keep up with the web browsers' technologies; otherwise, some services can be failed and become inaccessible.

A desktop application is a software program which runs stand-alone on a computer and performs a specific task by an end-user. There are many ways to install a desktop application without limitation, including a website, a distributed platform, or a data storage device. If the task that a desktop application performs does not require an internet connection, that desktop application can run offline without any problem. Furthermore, desktop applications can perform complex calculations better compared to web applications.

2.2 .NET and .NET Framework

For Windows Operating System, desktop applications created using .NET, for example, Windows Presentation Foundation, have backward compatibility, which means applications built with previous .NET versions can work without modification on later versions. Compared with native Windows API, which can

also be used to create a desktop application, .NET is a higher-level framework that provides more framework classes and cross-platform features. Moreover, .NET provides a better selection of class libraries.

.NET, created by Microsoft, is a platform with tools and libraries to build applications. .NET used to be a predominant implementation of the Common Language Infrastructure (CLI), running mainly only on Microsoft Windows (.NET Framework 2022). After that, it was replaced by a different .NET project to provide cross-platform features.

Nowadays, .NET is used by developers to create many kinds of applications, including mobile, web, or desktop applications. For example, ASP.NET Core is a cross-platform framework for building modern web application. .NET is widely used because it is free, open-source, trustworthy, easy to use, and applications running on the framework can be implemented with many different programming languages, for example, C#, Visual Basic .NET, or F# (What is .NET? An open-source ... 2022).

The shared library concept for .NET is called class libraries. They allow functionality to be packaged into modules, also called components that can be used by multiple applications. These modules can be used in multiple applications or platforms. Class libraries can be used as a means of loading functionality that is not needed or known at load time or run time. Moreover, .NET class libraries can have access to all the application programming interfaces (APIs) in a specific platform and take significant dependencies on that execution environment. For example, the classic implementation of .NET Framework is a platform-specific class library for Windows Operating Systems. Compared to native Windows API, .NET Framework provides much better libraries with many abstraction levels, so that developers do not need to write low-level and complex code. .NET class libraries can also be portable. Because different execution environments often use a different sub-set of .NET implementation, to enable cross-platform, one of .NET's approaches was to create .NET Portable Class Libraries (PCL). They defined the execution environment as a synthetic platform or platform intersection and take dependencies on it. PCL allows developers to choose a set of platforms that needed support, for example, .NET Framework and Xamarin.iOS.

However, the more platforms it needs to support, the less APIs are available. Finally, to establish uniformity in .NET ecosystem. .NET Standard class libraries are the combination of the advantages of platform-specific and portable libraries. Consequently, *"they are platform-specific in the sense that they expose all functionality from the underlying platform"*. (Lander et al. 2022.)

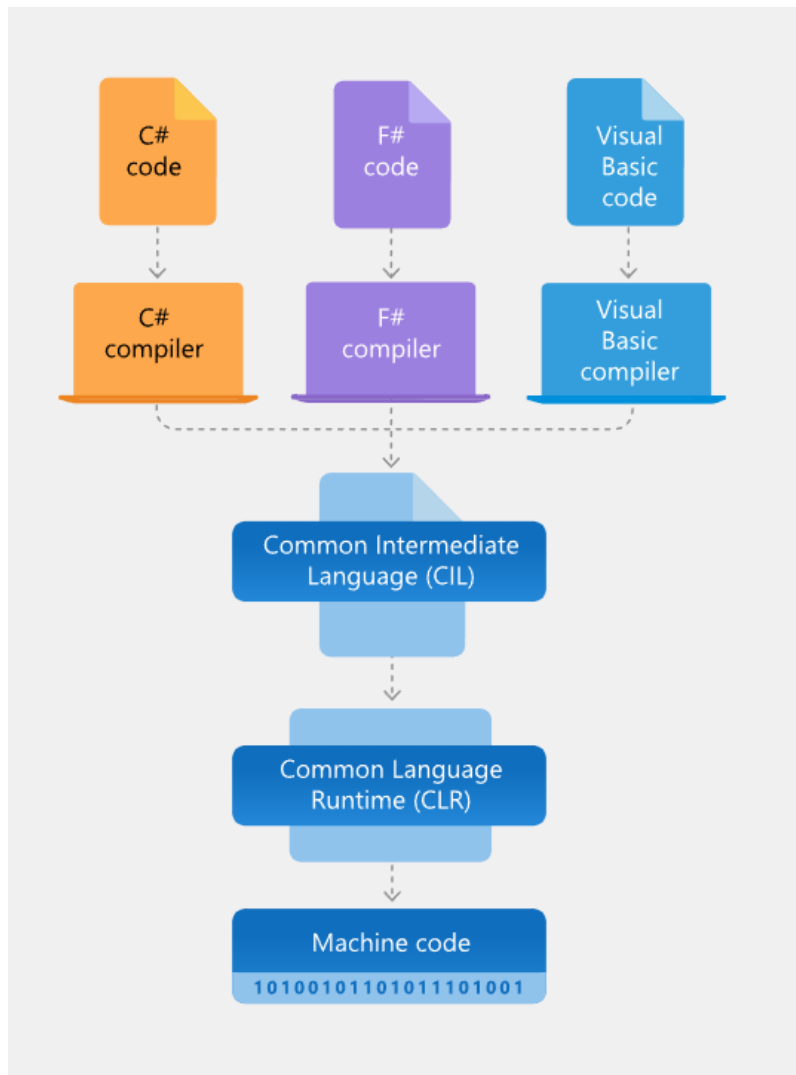


Figure 1. The architecture of the .NET Framework (What is .NET Framework...)

Figure 1 shows how the C#, F# and Visual Basic code are translated to Machine Code by using .NET Framework.

.NET Framework is the original implementation from .NET. Moreover, .NET Framework consists of two major components: the Common Language Runtime (CLR) and Class Library. CLR is an execution engine responsible for running the applications. .NET Framework Class Library (FCL) is an implementation of .NET Standard Library. FCL's APIs are organized into a

hierarchy of namespaces. They will either belong to System.* or Microsoft.* namespaces. FCL includes functions ranging from common including read files or write files, to complex, such as initializing a new instance of SMTP client to send email. (.NET Framework 2022.)

2.3 C# programming language

.NET Framework applications can be written in F#, Visual Basic .NET (VB.NET), C# and even C++. F# is a functional-first, object-oriented programming language, and can be considered as the best superset of C#. However, there are many existing libraries which are not too compatible with F#. Visual Basic .NET is also a high-level, object-oriented programming language. VB .NET syntax use statements similar to English to specify actions which are more familiar with developers who already have experience with BASIC. Meanwhile, C#'s core syntax is similar to C-style languages such as C, C++ and Java (C Sharp (programming... 2022).

According to Wagner, Bill et al. (2022), C# is a modern, object-oriented, and robust programming language. Having roots in the C family, programmers who know Java, JavaScript, C, or C++ can immediately take a grasp of C# code. C# supports strongly typed variables, which is crucial for real-world applications by reducing the risks of data loss from implicit conversions (Corob-Msft et al. 2021).

2.4 Windows Presentation Foundation

Windows Presentation Foundation plays a vital role in creating the thesis product, and the application strictly follows the architectural design.

2.4.1 Introduction

Windows Presentation Foundation (WPF) was initially developed and released by Microsoft as a part of .NET Framework 3.0 in 2006, known as "Avalon." WPF is a UI framework for rendering user interfaces for Windows-based applications and desktop client applications (Windows Presentation Foundation 2022). Generally, WPF uses XAML on XML-based language for creating user interfaces and utilizes data binding.

Figure 2 shows an example of a desktop application in debugging mode, which was made using the WPF framework.

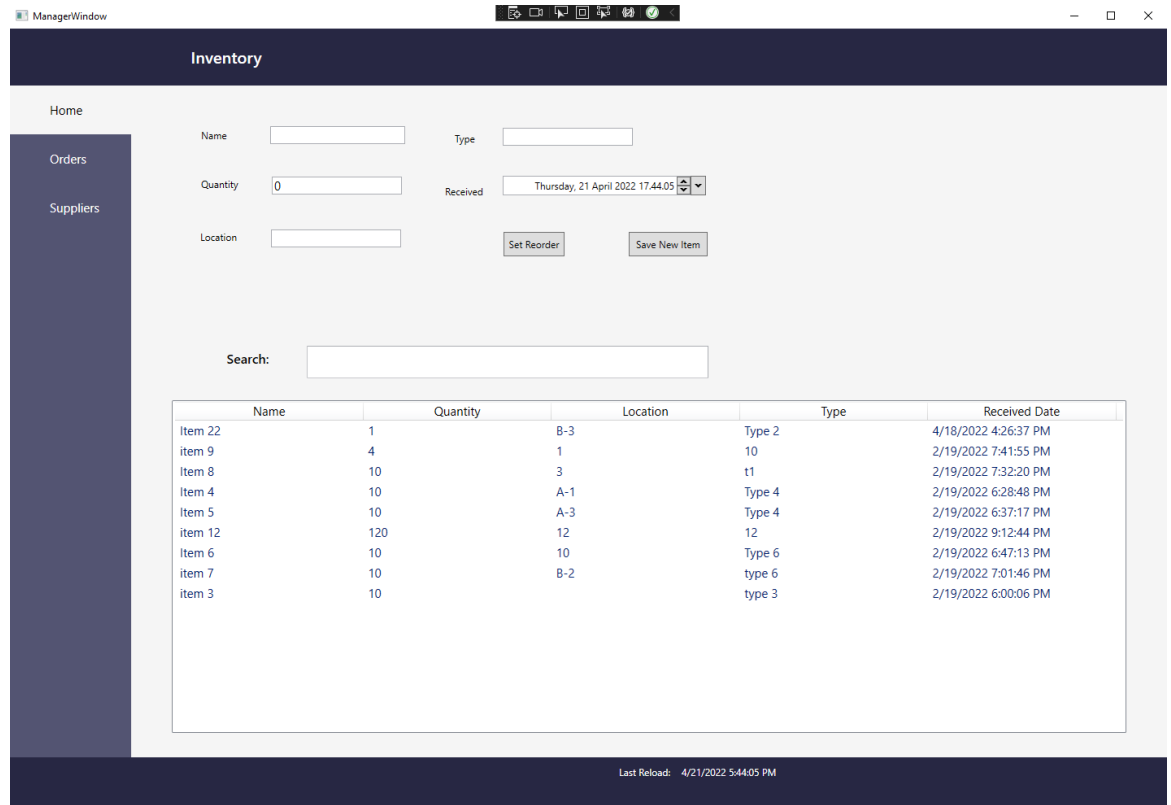


Figure 2. An example of a desktop application

As the latest approach from .NET to create a desktop application, WPF excels its ancestor, WinForms, by providing more features, including integrating UI, 2D, and 3D graphics by utilizing DirectX with XAML support. Moreover, it adapts well to the current trend of separating UI logic from back-end logic. WPF's UI element can be integrated from many different libraries. They can also be created from scratch. As a result, it creates unlimited possibilities for customization. WPF applications can also be hosted on a website. (George 2021; Hammad 2021.)

Mainly, mark-up and code-behind are supported by WPF. For instance, application can create a user interface using XAML and bind it with a controller (or code-behind) to affect its functionalities.

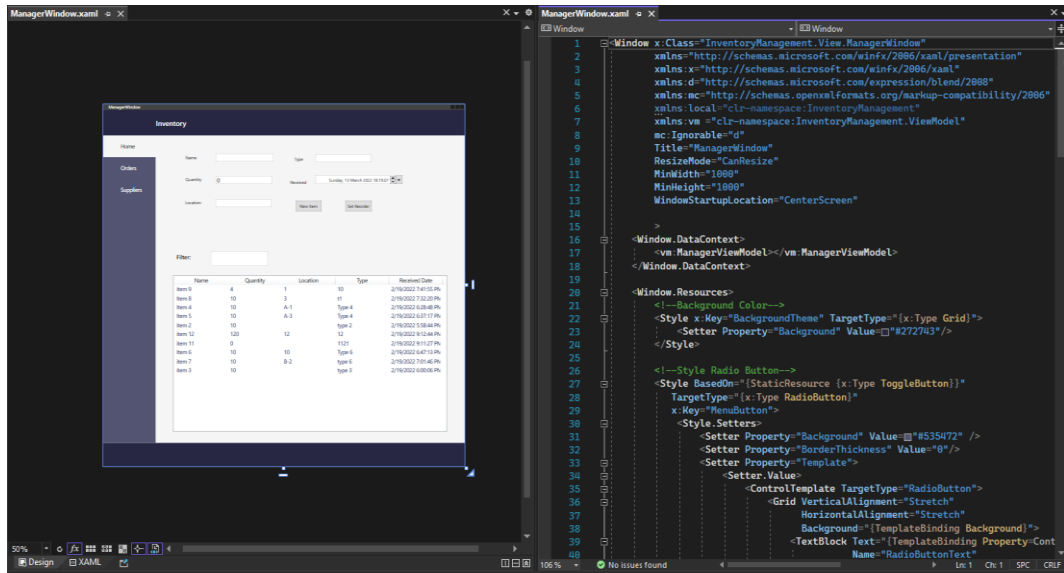


Figure 3. An example of creating user interfaces with WPF using XAML

Figure 3 illustrates the process of creating a UI using XAML, while Figure 4 demonstrates how code-behind files connect to XAML and implement its behavior. In this case, the code-behind file affects the Windows MaxHeight and MaxWidth properties. On the right side of Figure 3 is the design panel which helps developers to visualize how the code from XAML file will look like.

```

1  using System.Windows;
2
3  namespace InventoryManagement.View
4  {
5      /// <summary>
6      /// Interaction logic for ManagerWindow.xaml
7      /// </summary>
8      public partial class ManagerWindow : Window
9      {
10         private string UserName { get; set; }
11         private string Token { get; set; }
12
13         public ManagerWindow()
14         {
15             InitializeComponent();
16
17             MaxHeight = SystemParameters.MaximizedPrimaryScreenHeight;
18             MaxWidth = SystemParameters.MaximizedPrimaryScreenWidth;
19
20         }
21     }
22 }
23
24
25
26
27

```

Figure 4. An example of code-behind file

When creating the application, we can see three essential files: App.config, and App.xaml with the App.xaml.cs code-behind file. App.config is responsible for storing configurations and information about the packages being used inside the application. App.xaml is responsible for declaring the resources for the application. Moreover, we can set the start-up view for the application and

attach some functions inside the OnStartup event in App.xaml.cs code-behind file.

2.4.2 Controls

According to Microsoft Docs *“Control is an umbrella term that applies to a category of WPF classes hosted in either a window or a page, have a user interface, and implement some behavior”* (George & Coulter 2021). Controls are an essential component in a WPF application. It is what users can see and interact with, subsequently providing the user experience.

Many built-in WPF controls are responsible for handling different tasks, from showing data to the user to receiving user inputs. For example, the ListView control allows the application to display a list of items, or Grid control can be used to help layout other controls.

Additionally, WPF also provides a flexible way of creating the application by allowing us to make our custom Controls. A custom control can be created by building a markup file with some built-in Controls and a code-behind file to adjust its behavior. Furthermore, the appearance of a Control can be customized by using inline properties or a Style.

2.4.3 Layout

When creating a user interface, we need to arrange controls into a layout. The main requirement of a layout is that it can be fitted with many different display sizes. WPF provides a layout system that helps create responsive layout.

A layout system is responsible for relative positioning, which increases the controls' ability to adapt to changing window size or display conditions. The layout system acts as a negotiator between controls to decide the layout. The negotiation process has two steps: “first, a control tells its parent what location and size it requires; second, the parent tells the control what space it can have”. (George & Coulter 2021)

Consequently, WPF provides several layout controls, including Canvas, DockPanel, Grid, etc. Each of them has its way of managing its child controls. For example, Grid will position its child controls into columns or rows.

2.4.4 Data binding

Data binding is one of the essential features of WPF. It enables the application to show and allow interaction with data. According to George and Coulter (2022), the process initiates a connection between the user interface and the data it presents. Figure 5 shows the bridge connection between the binding target and the binding source. If settings are appropriately configured, when the property's data change, it will trigger a notification that will update the bound elements automatically.

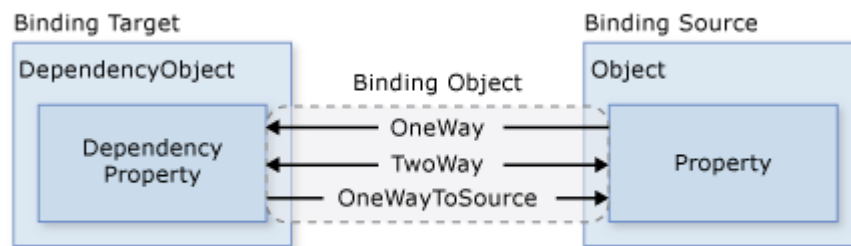


Figure 5. Data binding (George & Aaron 2022)

Figure 5 also shows many different binding connections, or data flows, between the binding target and source. OneWay binding is suitable for read-only contents, where the target will be updated when the property is updated. OneWayToSource is the opposite way of OneWay binding - this binding should be used when the property needs to be updated from the user interface. The most useful binding connection is TwoWay binding. If there is a change in either the binding source or the target, the other end will also get updated automatically. For many controls in WPF, OneWay binding is the default property. However, some dependency properties, such as `TextBox.Text` and `CheckBox.IsChecked`, are TwoWay binding (George & Aaron 2022). For TwoWay and OneWayToSource binding, every time the Dependency Property changes, it will trigger an `UpdateSourceTrigger` event, which will notify the Property of the change, so the Property can also be updated.

When data binding is used in XAML, elements will automatically check the DataContext property to find the binding source. DataContext can also be set using code-behind, but declaring in the XAML file will also help resolve or debug faster in compile time.

2.4.5 Windows

In WPF, windows contain many controls, which help visualize and interact which data. According to George (2022), a window can be divided into client and non-client areas.

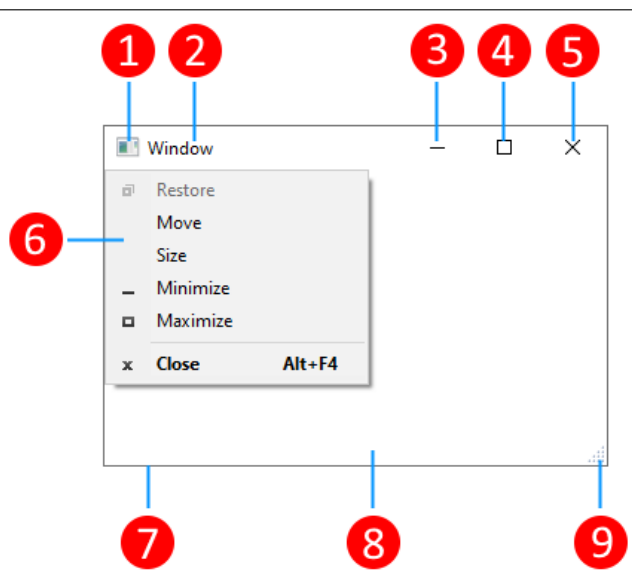


Figure 6 Components of a Window (George 2022)

Figure 6 shows the nine components of a window. The non-client area includes an icon (1), title (2), minimize button (3), maximize button (4), close button (5), system menu (6), and the border (7); these components are common to most windows. Meanwhile, the client area (8) and resize grip (9), is used by developers to add contents or controls the user can see and interact with.

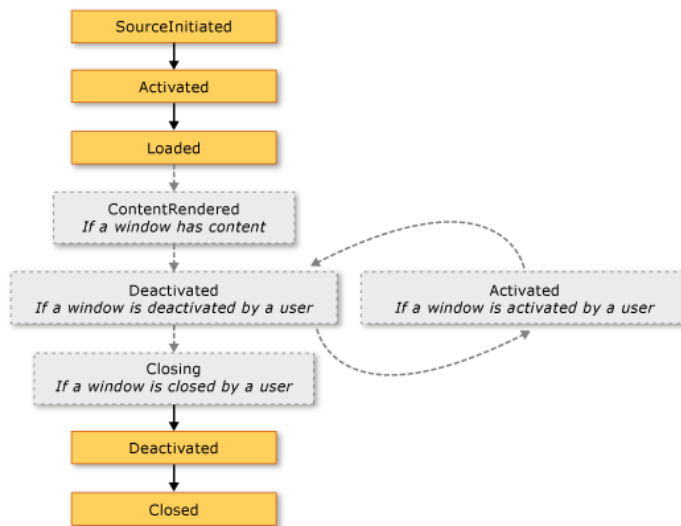


Figure 7. Window life events (George 2022)

Figure 7 shows the life events of a Window. When a Window is first opened, it becomes the activated Window, which is capable of capturing user input. When a user switches from that Window to another, the one that is presently activated is deactivated, and the one that is deactivated is activated.

2.5 Model-View-ViewModel architectural pattern

2.5.1 Overview

Using an architectural pattern is always considered best practice in software engineering, although sometimes it can result an overkill or overcomplicate the application (Architectural pattern 2022). Nevertheless, architectural designs help developers solve and define some essential cohesive elements of software architecture.

Model-View-Controller (MVC) is a famous design pattern, which has been widely used by developers and integrated into many programming languages. The main idea behind MVC is to separate the user interface (View) from business logic (Model) and function to alter the state (Controller). Moreover, it enables one View to switch between Controllers to fit it needs and one Controller can be used by multiple View. As a result, it became harder for unit test from View perspective. Model-View-Presenter is similar to MVC, but it focuses more on presentation logic. Presenter will receive user input from View and process with Model's help before sending data back to View. Yet

View has reference to Presenter, and Presenter aware of View, which will make unit testing become more difficult.

Model-View-ViewModel (MVVM), inspired by MVC and MVP, was invented by Microsoft architects Ken Cooper and Ted Peters to simplify the event-driven programming of user interfaces. Subsequently, it was incorporated into WPF. In MVVM implementation, ViewModel doesn't need to have any prior knowledge about the View, therefore unit testing become much easier with ViewModel. Nowadays, MVVM can be implemented with many frameworks, including Vue.js, Reactjs, or Angular. (Model-view-viewmodel 2022.)

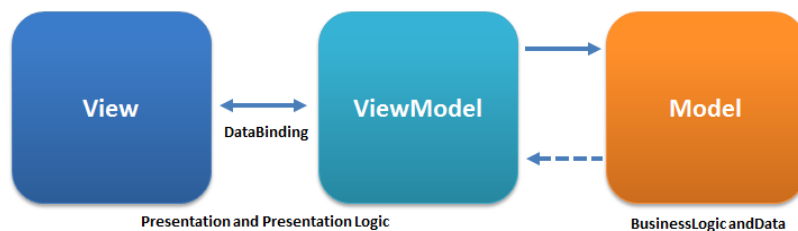


Figure 8. Model-View-ViewModel connection (Model-view-viewmodel 2022)

Model-View-ViewModel contains three essential components which are Model, View, and ViewModel. Figure 8 shows how these three components communicate with each other. Model and ViewModel are usually written in C#. Firstly, the Model component represents the content of the data. The data's classes and properties are declared inside the application in this component. Secondly, the View component is what users can see on the screen. It shows the applications' content, including models, and users can interact with the View by using mouse clicks or keyboard events. Figure 9 shows how a double-click mouse event is handled using a code-behind file.

```

private void listViewDoubleClick_Handler( object sender, MouseButtonEventArgs e )
{
    Item selectedItem = (Item)((FrameworkElement)e.OriginalSource).DataContext;
    HomeViewModel currentDataContext = (HomeViewModel)DataContext;

    if (selectedItem != null)
    {
        currentDataContext.ItemSelected.Execute(selectedItem);
    }
}
  
```

Figure 9. An example of handling mouse clicks event.

Finally, the essential component is ViewModel. ViewModel is responsible for handling data binding between View and Model. Specifically, in WPF, markup files contain one property, DataContext, which we can use to connect to ViewModel.

2.5.2 Model-View-ViewModel in WPF

Sharing the same similarity with any other programming language or framework, everything is simple when a WPF application is first created. As the application grows, more features are added to the code base and the complexity increases. As a result, the WPF community adapted the MVVM pattern to help organize the codebase.

According to Smith (2009), data binding is the essential characteristic of WPF to enable the potential of the MVVM pattern. WPF encouraged many aspects of MVVM, such as utilizing the strong separation of displaying the data from its state and behavior. When properties from a View get bound with a ViewModel, bridge connections are automatically generated. Moreover, command binding is an essential aspect of WPF, which empowers the MVVM pattern. A function from a ViewModel can be exposed to a View and allowed to be consumed by controls. Furthermore, ViewModel classes are easy to unit test. By separating View and ViewModel, all states, behavior, and functionality stay in a different set, making unit testing faster.

2.6 Database

According to Oracle (What is a database...), data or information, typically saved electronically in a computer system, is organized into a collection called a database. Moreover, on the market, many different databases are widely used by developers. Each of the databases has its benefits and drawbacks. However, Relational Databases and NoSQL are the most well-known ones.

In relational database, data is organized into tables, and the data inside a table may have connections or relationships to a different table. An instance of data is uniquely represented in a table row with the corresponding data type. As a result, all the instances in one table have the same structure and make it easier to add, remove, or edit the data. However, this will limit the flexibility for

data and generate possibility for errors whenever a column is added or removed from the table. (Jatana et al. 2012, 2.)

According to Schaefer (What is NoSQL...), NoSQL, known as “non-SQL” or “not only SQL” is preferred to databases that store data in different but more modern formats than traditional relational tables. Consequently, it provides flexible schemas for different usage and better horizontal scaling. NoSQL is recommended for scenarios in which databases need to handle vast volumes of data. As a result, NoSQL is selected for this thesis product because of its flexibility in our case.

There are two options for choosing the place to host our database. The first way is to host on the same local computer with the application, called an on-premises database. This method has proven secure and reliable because we will have access to the same machine, but it also means an increase in the complexity of scaling or installing it.

Meanwhile, cloud databases have taken the world and become ubiquitous in recent years. According to IBM (What is a cloud...), companies will provide the database as a service, and users can access them through their cloud platform. This method will eliminate all the complexity and hard work of implementing, maintaining, and scaling the database. Furthermore, cloud databases usually come with a pay-as-you-go subscription, which is beneficial since we can pay for what we are using. Despite the benefits of cloud databases, it also comes with a cost. Firstly, there will be a security risk since the physical databases are in the cloud’s platform provider’s possession. Additionally, connecting to the cloud database is usually through the internet, exposing the database to cybersecurity threats. However, the disadvantages of cloud databases cannot overwhelm the benefits it brings back.

Firebase, developed by Google, is a cloud platform for developing web and mobile applications. It helps developers or companies by providing a wide range of services to build, improve and grow their applications’ infrastructure, such as Authentication, Database, Storage, and Hosting. Firestore is Firebase’s NoSQL cloud database. It has flexible hierarchical data structures and impressive scalability capabilities.

3 IMPLEMENTATION

The practical part of this thesis shows how the application was structured and developed. All the explained concepts from chapter 2 will be used in this implementation.

3.1 Design and create data model

In C# programming language, a solid object-oriented programming language, a class is a reference type. It is a prototype or blueprint for a variable. It also contains the business logic, which determines how the data is processed inside the application.

The purpose of the application is to manage items inside the inventory as well as the orders and supplier for reordering. As a result, the focus classes are the Item, Order, and Supplier for this application. Each will contain a unique **Id** field generated from the Firestore database.

The Item model defines how the general item stored inside the application. The class has seven fields. As shown in Figure 10, the **Name** and **Type** are the item's name and which type can be identified. **Location** is where the item is currently stored. **Quantity** is the current quantity of the item in stock, and **ReceivedDate** is the date it arrived at the inventory. Finally, **Reorder** field is a **Reorder** rule for the **Item**. If it is set, when the Item quantity gets below that threshold, the application will automatically send an email to the **Supplier** with a quantity.

```
1  using System;
2
3  namespace InventoryManagement.Model
4  {
5
6      public class Item
7      {
8
9          public string Id { get; set; }
10
11         public string Name { get; set; }
12
13         public string Type { get; set; }
14
15         public string Location { get; set; }
16
17         public int Quantity { get; set; }
18
19         public DateTime ReceivedDate { get; set; }
20
21         public Reorder Reorder { get; set; }
22
23     }
24 }
25
26
27
28
29
30
31
32
```

Figure 10. Item class

As shown in Figure 11, the **Order** class is the prototype for orders created inside the application. This class has two properties: **Status** and **Type**, which have an enum data type for strict data value. The other properties are **Id**, **Name**, **SupplierId**, and **Items**. As the name implies, they are responsible for describing the character, supplier, and list of items from that order.

```

public enum OrderStatus
{
    DELIVERING,
    FINISHED,
    CANCELED,
}

public enum OrderType
{
    NEW,
    REORDER
}

public class Order
{
    public string Id { get; set; }

    public string Name { get; set; }

    public string SupplierId { get; set; }

    public DateTime OrderDate { get; set; }

    public OrderStatus Status { get; set; }

    public OrderType Type { get; set; }

    public List<ItemOrdered> Items { get; set; }
}

```

Figure 11. Order class

The **Order** class from Figure 11 is a blueprint for order data taken from Firestore. Figure 12 shows the populated Order class used for ViewModel to make it easier to query and access the data. In this class, all the **Quantity** and **ItemId** inside **List<ItemOrdered>** are populated with accurate data when it is queried.

```

public class OrderViewModelClass
{
    public string Id { get; set; }

    public string Name { get; set; }

    public Supplier Supplier { get; set; }

    public DateTime OrderDate { get; set; }

    public OrderStatus Status { get; set; }

    public OrderType Type { get; set; }

    public List<ItemOrderedDetail> Items { get; set; }
}

```

Figure 12. Populated Order class

Supplier class describes some basics information for a supplier, as shown in Figure 13. The email field in this class is necessary for reordering.

```
1 namespace InventoryManagement.Model
2 {
3     public class Supplier
4     {
5         public string SupplierId { get; set; }
6         public string Name { get; set; }
7         public string Email { get; set; }
8         public string Phone { get; set; }
9         public string Address { get; set; }
10    }
11 }
12
13
14
15
16
```

Figure 13. Supplier class

Lastly, another essential class in this application is the **Global** static class. This class stores data and contains all the business logic for fetching, updating, and deleting data from Firebase. Figure 14 shows the data stored inside the Global class. These are **ObservableCollection** type for **Order**, **Reorder**, **Item**, and **Supplier**. **ObservableCollection** type is used because it has a built-in notify function to the View, which is convenient when an item inside the array is add, deleted or the array is refreshed.

```
public static class Globals
{
    public static ObservableCollection<Order> Orders = new ObservableCollection<Order>();
    public static ObservableCollection<Reorder> Reorders = new ObservableCollection<Reorder>();
    public static ObservableCollection<Item> Items = new ObservableCollection<Item>();
    public static ObservableCollection<Supplier> Suppliers = new ObservableCollection<Supplier>();
}
```

Figure 14. Data stored in the Global class

```

#region Load Database
public static async Task<ObservableCollection<Order>> Load_Orders()
{
    FirestoreDb db = GetDbRef();
    CollectionReference ordersRef = db.Collection("order");
    QuerySnapshot snapshot = await ordersRef.GetSnapshotAsync();

    ObservableCollection<Order> orders = new ObservableCollection<Order>();

    foreach (DocumentSnapshot doc in snapshot.Documents)
    {
        try
        {
            Order order = new Order();
            order.Id = doc.Id;

            Dictionary<string, object> document = doc.ToDictionary();
            order.Name = (string)document["name"];
            order.SupplierId = (string)document["supplierId"];
            order.OrderDate = ((Timestamp)document["orderDate"]).ToDateTime();
            order.Items = parseObjectToListString(document["items"]);
            order.Type = (OrderType)Enum.Parse(typeof(OrderType), (string)document["type"]);
            order.Status = (OrderStatus)Enum.Parse(typeof(OrderStatus), (string)document["status"]);
            orders.Add(order);
        }
        catch (Exception e)
        {
            MessageBox.Show(e.Message);
        }
    }

    Orders = orders;
    return Orders;
}

```

Figure 15. Method for fetching Order from Firestore

Figure 15 shows an example of fetching data from Firestore. Firstly, for all actions, in addition to fetching, the application also needs to get a reference to the database. The connection string to the database is stored in a JSON file. Next, the **CollectionReference** is needed for the query. Now the application can make the query into the Firestore database and store it in a snapshot. The rest of the process includes parsing the data to be used in the application.

```
namespace InventoryManagement.Model
{
    public enum LogTarget
    {
        ITEM,
        ORDER,
        SUPPLIER
    }

    public enum LogAction
    {
        ADD,
        DELETE,
        UPDATE
    }

    public class Log
    {
        public string Name { get; set; }

        public LogTarget Target { get; set; }

        public LogAction Action { get; set; }

        public DateTime Date { get; set; }

        public string Description { get; set; }
    }
}
```

Figure 16. Log class

Besides the previous **Model**, there is also a **Log** class for logging all the actions including **ADD**, **UPDATE** or **DELETE**. The class has four fields, including **Name**, **Action**, **Target**, and **Description**, as shown in Figure 16.

3.2 ViewModel Class Hierarchy

The same features appear in most of the ViewModel functionality, especially **INotifyPropertyChanged**. This is one of the most important features implemented in WPF to help empower the MVVM pattern. If a property in a ViewModel gets updated, the property can trigger **PropertyChangedEventHandler** to notify WPF binding connection with the View, as shown in Figure 17. Consequently, we can separate the logic into a class base and let new ViewModel classes inherit it.

```

using System.ComponentModel;

namespace InventoryManagement.ViewModel
{
    public abstract class ViewModelBase : INotifyPropertyChanged
    {
        #region Constructor
        protected ViewModelBase()
        {
        }
        #endregion

        #region INotifyPropertyChanged

        public event PropertyChangedEventHandler PropertyChanged;

        public void OnPropertyChanged( string propertyName )
        {
            PropertyChangedEventHandler handler = PropertyChanged;

            if (handler != null)
            {
                handler(this, new PropertyChangedEventArgs(propertyName));
            }
        }

        #endregion
    }
}

```

Figure 17. INotifyPropertyChanged base class

Figure 18 shows how **OnPropertyChanged** is used. When the ViewModel gets a new value for the public property **Name**, it will update the private member **_name**. Moreover, it will also trigger the **OnPropertyChanged** function and inform the connected View.

```

private string _name;
public string Name
{
    get { return _name; }
    set { _name = value; OnPropertyChanged("Name"); }
}

```

Figure 18. OnPropertyChanged implementation

Figure 19 shows an example of a class hierarchy for complex scenarios. The same logic in many ViewModel classes can be extracted and placed in many small ViewModelBase types when the application grows.

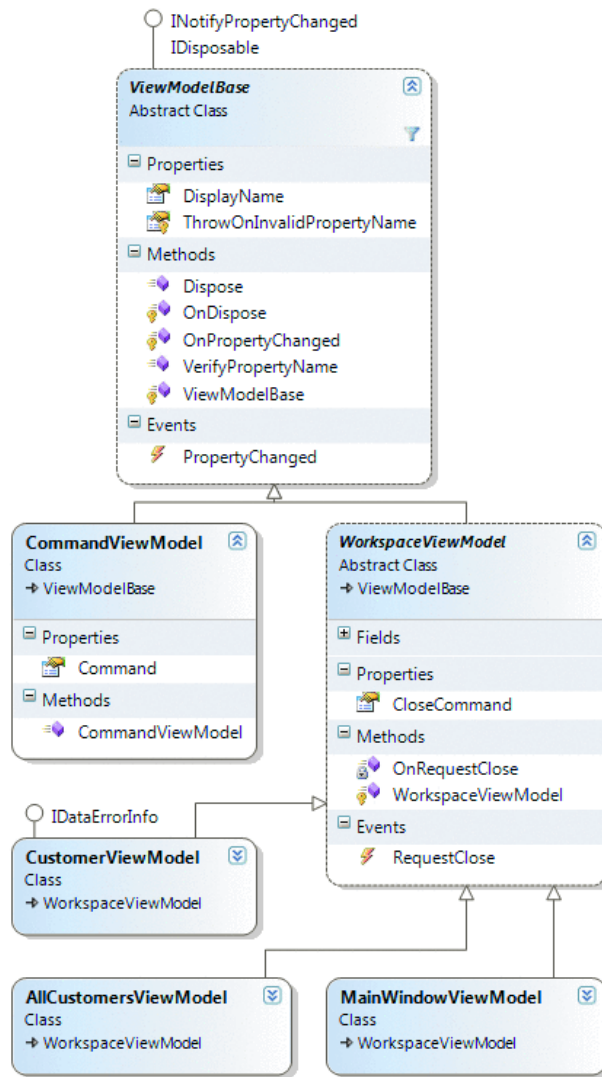


Figure 19. Inheritance Hierarchy (Smith 2009)

3.3 Data and command binding

Introduced in Section 2.2.4, data binding is a strong characteristic of WPF. Firstly, a **View** needs to bind with a **ViewModel**. This can be done by assigning the **DataContext** property from a View with a **ViewModel** class. As shown in Figure 20, a **DataContext** is declared in **ManagerViewModel** class.

```

<Window x:Class="InventoryManagement.View.ManagerWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:local="clr-namespace:InventoryManagement"
  xmlns:vm="clr-namespace:InventoryManagement.ViewModel"
  mc:Ignorable="d"
  Title="ManagerWindow"
  ResizeMode="CanResize"
  MinWidth="1000"
  MinHeight="1000"
  WindowStartupLocation="CenterScreen"

  >
  <Window.DataContext>
    <vm:ManagerViewModel></vm:ManagerViewModel>
  </Window.DataContext>

```

Figure 20. DataContext binding

In Figure 21, in **ManagerWindow** there is one **ContentControl**, which can be used to show the content of any type, and we can see its **Content** has a binding with **CurrentView**. And inside our **ViewModel** class, this **CurrentView** can be switched to any content using buttons.

```

<ContentControl
  Content="{Binding CurrentView}"
  >
</ContentControl>

```

Figure 21. ManagerWindow.xaml

Figure 22 shows a command bound with a button to change the **Content**. As we can see, the **CurrentView** is updated every time we press a button.

```

RelayCommand _homeViewCommand;
public ICommand homeViewCommand
{
    get
    {
        if (_homeViewCommand == null)
        {
            _homeViewCommand = new RelayCommand(async o =>
            {
                await Global.Load_Items();
                CurrentView = homeViewModel;
            });
        }
        return _homeViewCommand;
    }
}

RelayCommand _ordersViewCommand;
public ICommand ordersViewCommand
{
    get
    {
        if (_ordersViewCommand == null)
        {
            _ordersViewCommand = new RelayCommand(async o =>
            {
                await Global.Load_Orders();
                ordersViewModel = new OrdersViewModel();
                CurrentView = ordersViewModel;
            });
        }
        return _ordersViewCommand;
    }
}

```

Figure 22. ManagerViewModel class

As shown in Figure 23, when **CurrentView** is updated, **OnPropertyChanged** is triggered, which notifies **ManagerWindow** View about the change.

```

private object _currentView { get; set; }
public object CurrentView
{
    get
    {
        return _currentView;
    }
    set
    {
        _currentView = value;
        OnPropertyChanged("CurrentView");
    }
}

```

Figure 23. CurrentView property

When we want to bind a collection of objects to a control, for example, **ListView** or **ListBox**, we also need to implement **INotifyPropertyChanged**. However, WPF provides us with **ObservableCollection<T>** class, which can

automatically notify controls when an item gets added or removed, or the whole collection is refreshed.

Every markup file for View will have a code-behind file for handling events. However, we can also expose a function from ViewModel and use it as a command. This approach allows the function to access variables from the ViewModel directly. It can be done by creating a nested class inherited **ICommand** interface. (Smith 2009)

```
namespace InventoryManagement
{
    class RelayCommand : ICommand
    {
        #region Fields
        readonly Action<object> _execute;
        readonly Predicate<object> _canExecute;
        #endregion

        #region Constructors
        public RelayCommand( Action<object> execute ) : this(execute, null) { }
        public RelayCommand( Action<object> execute, Predicate<object> canExecute )
        {
            if (execute == null)
            {
                throw new ArgumentNullException("execute");
            }
            _execute = execute; _canExecute = canExecute;
        }
        #endregion

        #region ICommand Members
        public bool CanExecute( object parameter )
        {
            return _canExecute == null ? true : _canExecute(parameter);
        }
        public event EventHandler CanExecuteChanged
        {
            add
            {
                CommandManager.RequerySuggested += value;
            }
            remove
            {
                CommandManager.RequerySuggested -= value;
            }
        }
        public void Execute( object parameter )
        {
            _execute(parameter);
        }
        #endregion
    }
}
```

Figure 24. RelayCommand class inherits from ICommand

As shown in Figure 24, this nested class has two properties **_execute** and **_canExecute**. **_execute** is implemented with type **Action<object>**, which can be used to pass a method as a parameter, and **_canExecute** with type **Predicate<object>** which “represents the method that defines a set of criteria and determines whether the specified object meets those criteria” (Predicate delegate (system)). **Action<object>** and **Predicate<object>** are two

important delegates from the .NET framework. We can see how the RelayCommand is implemented in Figure 25.

```
#region RelayCommand
RelayCommand _ItemSelected;
public ICommand ItemSelected
{
    get
    {
        if (_ItemSelected == null)
        {
            _ItemSelected = new RelayCommand(o => OpenItemDialog(o));
        }
        return _ItemSelected;
    }
}
}

```

Figure 25. RelayCommand implementation

3.4 Background refetching

To ensure the data inside the application is always synchronized with the cloud database, **DispatcherTimer** is implemented inside the application. This will create a timer that is integrated into a prioritized queue of work items for a specific thread. From that, a function can be added to the **Tick** event of **DispatcherTimer**, and this **Tick** event will occur when the time elapsed.

```
#region DispatcherTimer
public static DispatcherTimer dispatcherTimer = new System.Windows.Threading.DispatcherTimer();
public static void InitializeDispatchTimer( EventHandler eventHandler )
{
    if (dispatcherTimer.IsEnabled)
    {
        dispatcherTimer.Stop();
        var eventField = dispatcherTimer.GetType().GetField("Tick", BindingFlags.NonPublic | BindingFlags.Instance);
        var eventDelegate = (Delegate)eventField.GetValue(dispatcherTimer);
        var invocationList = eventDelegate.GetInvocationList();
        foreach (var handler in invocationList)
            dispatcherTimer.Tick -= ((EventHandler)handler);
    }

    dispatcherTimer.Tick += new EventHandler(eventHandler);
    dispatcherTimer.Interval = new TimeSpan(0, 1, 0);
    dispatcherTimer.Start();
}
}
#endregion

```

Figure 26. DispatcherTimer

A **Dispatcher** is created on a thread when user starts the application. Moreover, that dispatcher will become the only **Dispatcher** associated with that thread. The responsibility of **InitializeDispatchTimer** in Figure 26 is to find and remove any event handler in **Tick** before adding a new one, so there will only be one event handler running at a time. As a result, every time user

switches between tabs, the application will pass a function to refetch data from cloud database.

3.5 Sending reorder email

If there is a reorder rule for an **Item**, when the **Quantity** of the **Item** meets the threshold, the application will send reorder email to a **Supplier**.

```
#region Send Email
public static void SendEmail( string SupplierId, int quantity, string itemName )
{
    Supplier supplier = Suppliers.Single(x => x.SupplierId == SupplierId);
    if (supplier == null)
    {
        MessageBox.Show("Unable to send Reorder email. Supplier didn't exist!");
    }
    try
    {
        string Body = $"<h3>Hello {supplier.Name},</h3>" + $"<div>I need {quantity} for {itemName}</div>! Please deliver it as soon as possible</div>" + $"<div>Best regards!</div><div>Huy</div>";

        var smtpClient = new SmtplibClient("smtp.gmail.com")
        {
            Port = 587,
            Credentials = new NetworkCredential("huyvothisemail@gmail.com", "wXvWzRp1n25F2"),
            EnableSsl = true,
        };
        var mailMessage = new MailMessage
        {
            From = new MailAddress("huyvothisemail@gmail.com"),
            Subject = "Reorder",
            Body = Body,
            IsBodyHtml = true,
        };
        mailMessage.To.Add(supplier.Email);
        smtpClient.Send(mailMessage);
    }
    catch (Exception e)
    {
        MessageBox.Show(e.Message);
    }
}
#endregion
```

Figure 27. Reorder function

As shown in Figure 27, the application itself will become a **SmtplibClient** and open port **587** to send email.

3.6 Result

This sub chapter shows the final resulting application. The startup position of the application is set to the middle of the screen where the mouse is at. The application is separated into four different tabs: Home, Orders, Suppliers, and History.

```

App.xaml
Application
1 <Application x:Class="InventoryManagement.App"
2   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4   xmlns:local="clr-namespace:InventoryManagement"
5   xmlns:vm="clr-namespace:InventoryManagement.ViewModel"
6   xmlns:view="clr-namespace:InventoryManagement.View"
7   StartupUri="View/Shared/ManagerWindow.xaml"
8   >
9   <Application.Resources>
10    <DataTemplate DataType="{x:Type vm:HomeViewModel}">
11      <view:HomeView/>
12    </DataTemplate>
13
14    <DataTemplate DataType="{x:Type vm:OrdersViewModel}">
15      <view:OrdersView/>
16    </DataTemplate>
17
18    <DataTemplate DataType="{x:Type vm:SuppliersViewModel}">
19      <view:SuppliersView/>
20    </DataTemplate>
21
22    <DataTemplate DataType="{x:Type vm:LogsViewModel}">
23      <view:LogView/>
24    </DataTemplate>
25  </Application.Resources>
</Application>

```

Figure 28. App.xaml

As we can see in Figure 28, when a user first opens the application, they are prompted to ManagerWindow.xaml, which is the Home Tab. This tab has a list of items in inventory and a **UserControl** to add a new item, as shown in Figure 29.

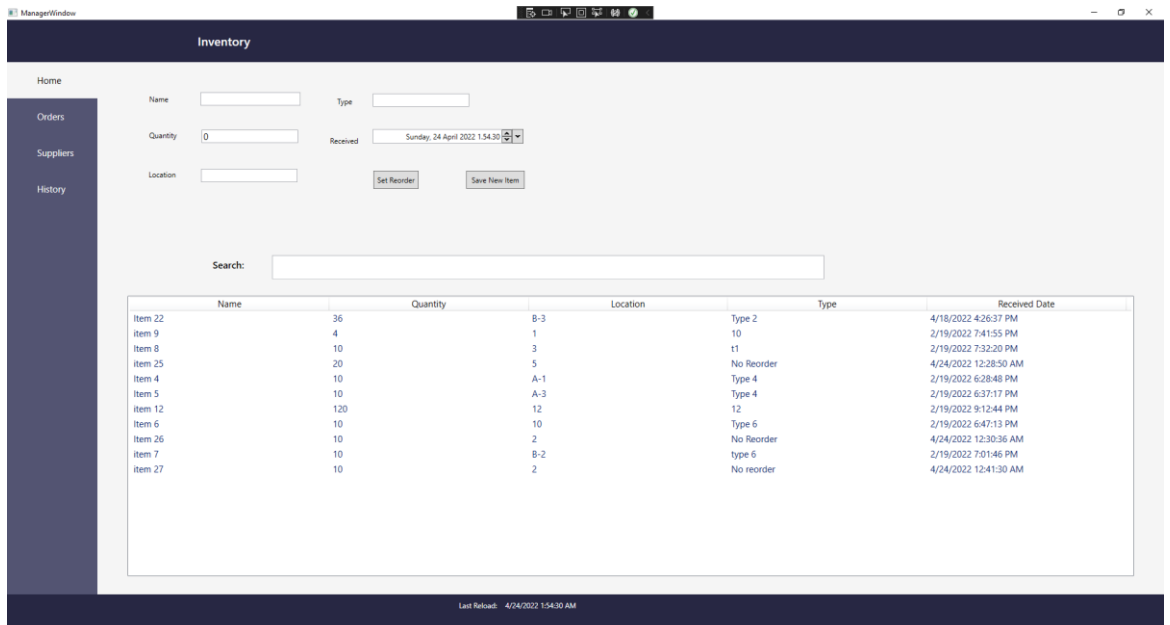


Figure 29. Home Tab

Moreover, the user can see and edit the item by double-clicking on the list, shown in Figure 30. On the top of the search bar, there are many fields for

user to add to create a new item. At the end of any tab, there is a **TextField** to indicate when the data was reloaded.

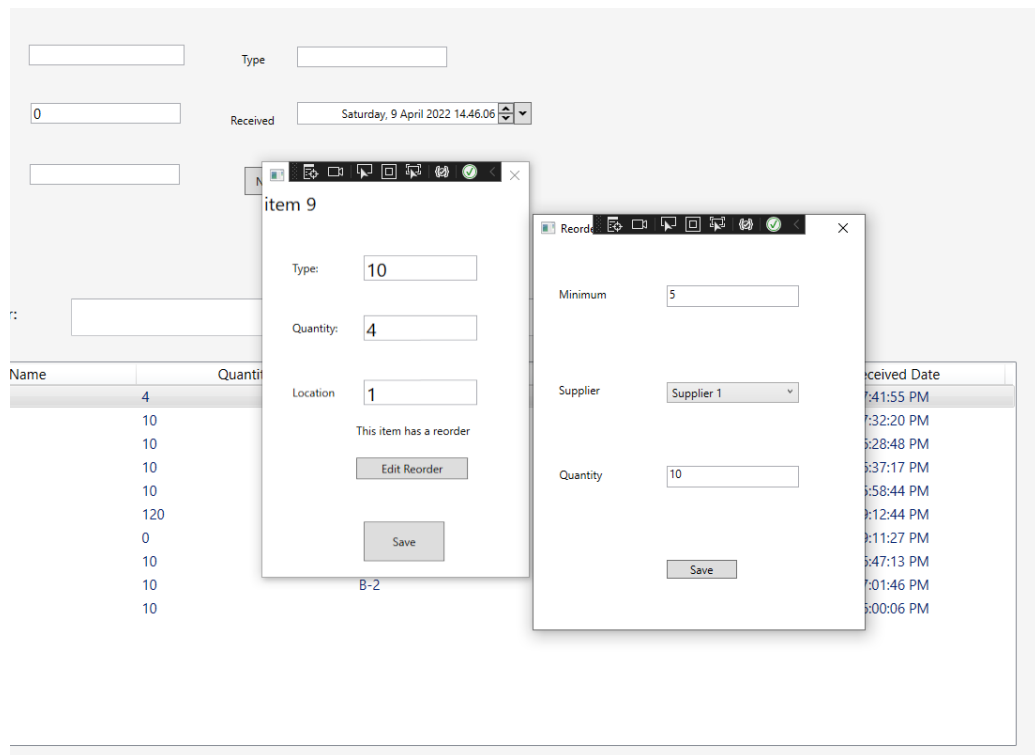


Figure 30. Item and Reorder rule

If the **Quantity** of **Item** goes below the **Minimum**, the application will send an email to the **Supplier**. Figure 31 shows **Order's** tab with a window to create a new order.

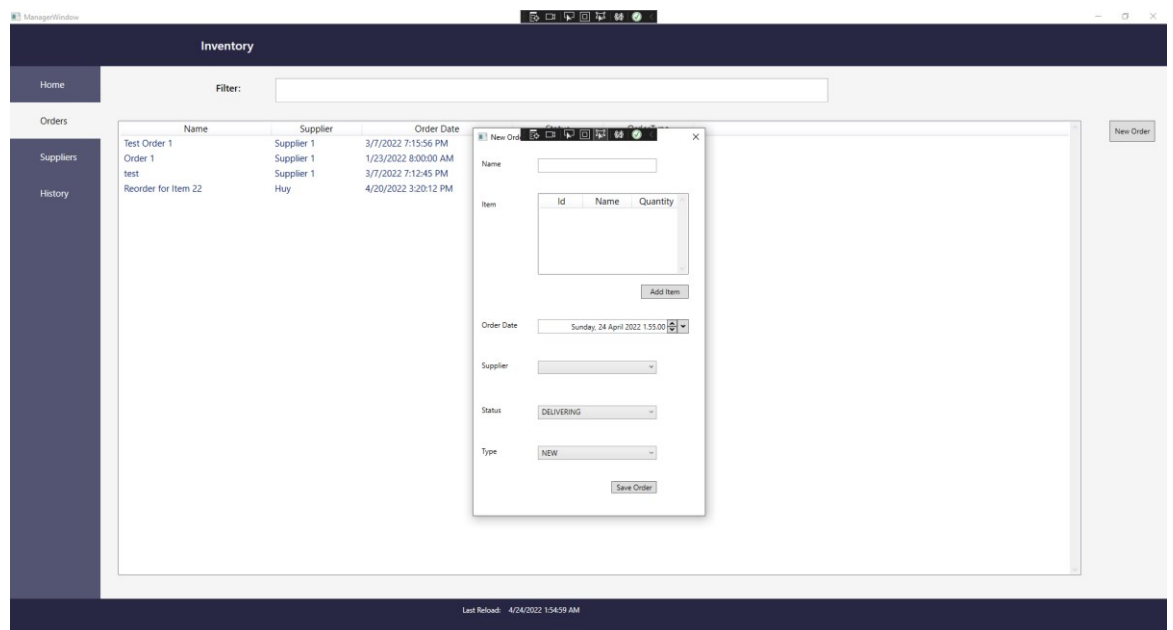
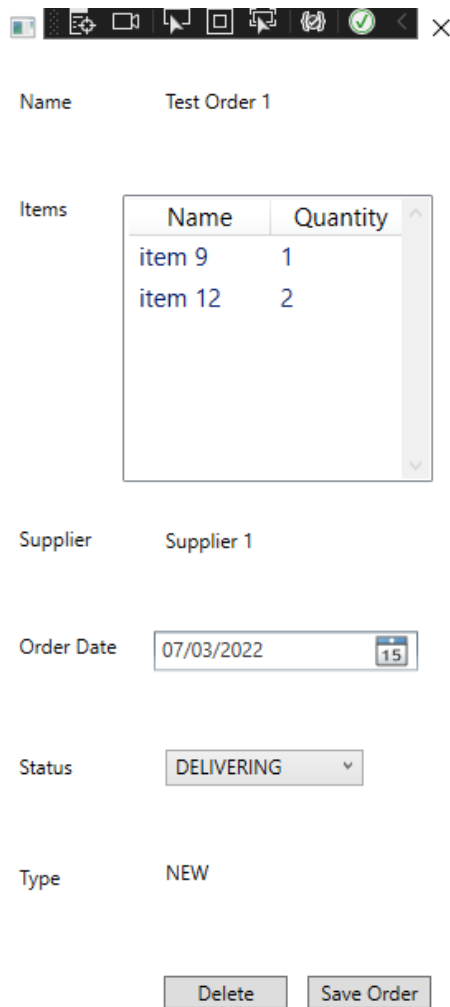


Figure 31. Order Tab with New Order window

Figure 32 shows detail about an order, which includes **Name**, **OrderedItem**, **Supplier**, **OrderDate**, **Status** and **Type**. Figure 31 shows



The screenshot shows a web application interface for viewing an order. At the top, there is a browser toolbar. Below it, the order details are displayed as follows:

- Name:** Test Order 1
- Items:** A table with two columns: Name and Quantity.

Name	Quantity
item 9	1
item 12	2
- Supplier:** Supplier 1
- Order Date:** 07/03/2022 (with a calendar icon showing the 15th)
- Status:** DELIVERING (dropdown menu)
- Type:** NEW

At the bottom, there are two buttons: "Delete" and "Save Order".

Figure 32. View Order

Figure 33 shows an example email for reordering. The supplier's name and contact information are taken from **Suppliers** tab.

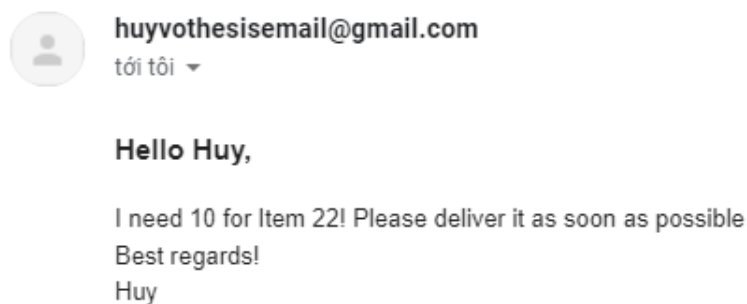


Figure 33. Reorder email to supplier

The Order tab and Supplier Tab share the command layout, containing a list of Orders or Suppliers and a button to add a new one, as shown in Figure 34.

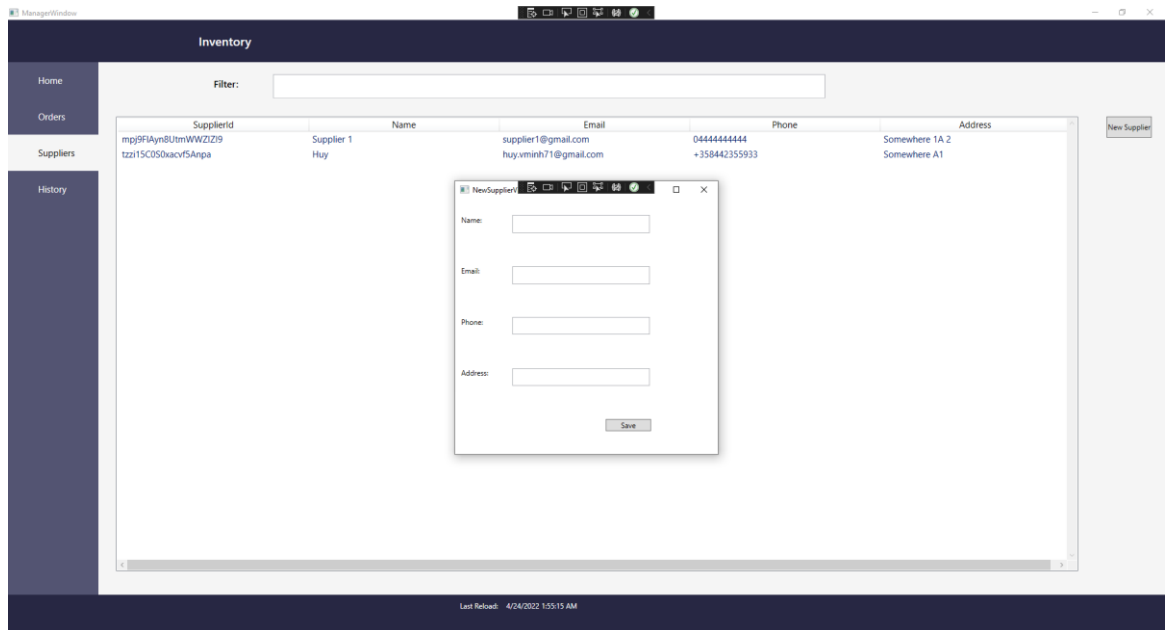


Figure 34. Supplier Tab

In Home, Orders or Suppliers tabs, there is a search bar for searching items in the list, as shown in Figure 35.

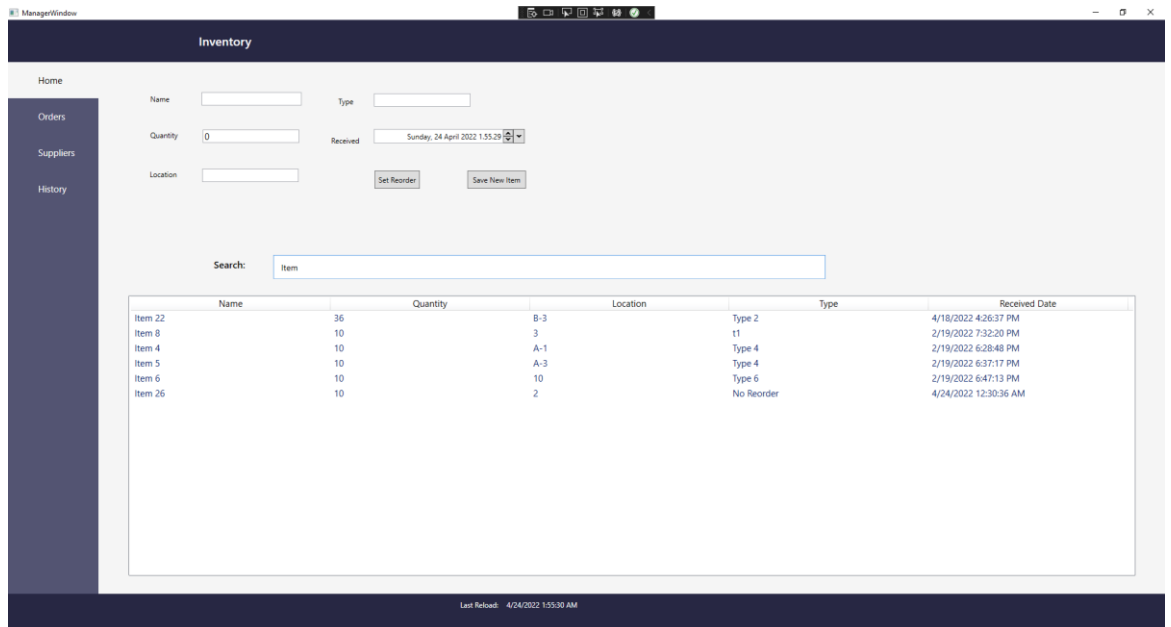


Figure 35. Searching for Item

Figure 36 shows how data are organized inside Firestore cloud database.

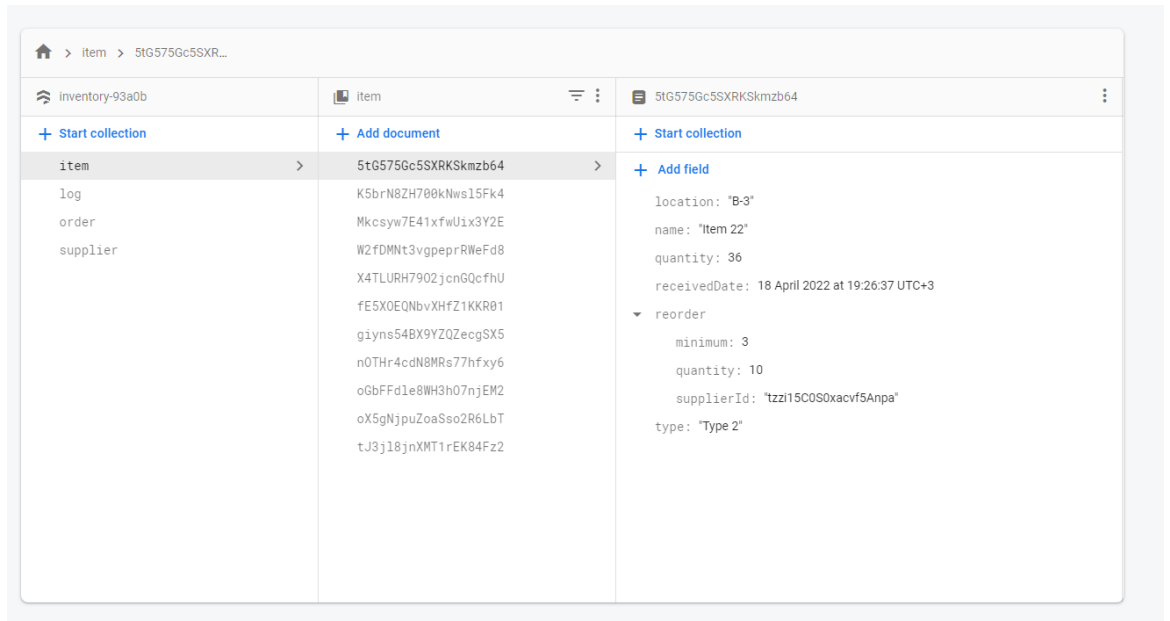


Figure 36. Firestore database

Figure 37 shows an example of unit testing in practice. As we can see, the `ManagerViewModel` is selected for testing.

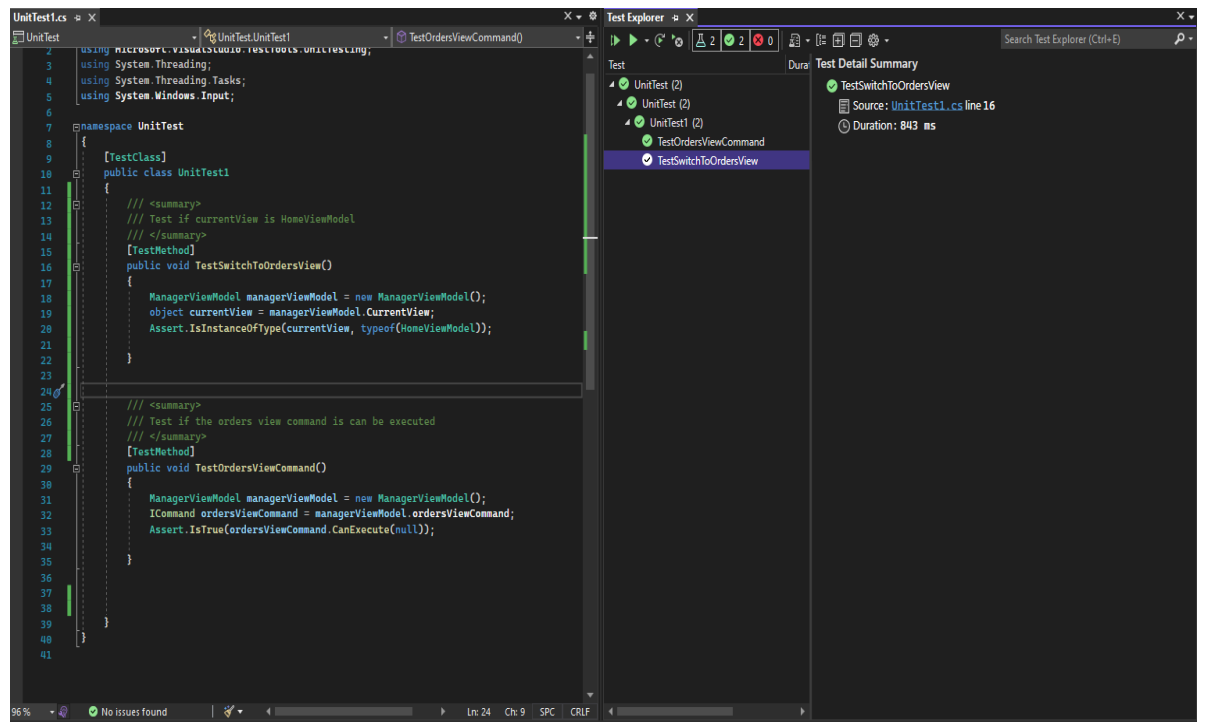


Figure 37. Unit testing

Figure 38 shows the History tab which contains all the logs in the application.

Suppliers	Name	Action	Target	Date	Description
History	Item item 24 update	UPDATE	ITEM	4/24/2022 1:35:23 AM	Reorder removed with quantity 10 a
	Item item 24 update	UPDATE	ITEM	4/24/2022 1:35:32 AM	Reorder removed with quantity 10 a
	New Item: item 24	ADD	ITEM	4/24/2022 12:28:49 AM	Quantity: 10, Location: 3, Type: Reor
	New Supplier Random	ADD	SUPPLIER	4/24/2022 1:16:39 AM	New supplier Random
	Item item 24 update	UPDATE	ITEM	4/24/2022 1:38:52 AM	Reorder removed with quantity 10 a
	Item item 24 update	UPDATE	ITEM	4/24/2022 1:26:56 AM	Reorder removed with quantity 2 an
	Order pitL0TY5nGy5i	UPDATE	ORDER	4/24/2022 1:39:19 AM	Order status is updated to FINISHEC
	New Item: item 26	ADD	ITEM	4/24/2022 12:30:47 AM	Quantity: 10, Location: 2, Type: No R
	Order: 0nqCj36nbbR	DELETE	ORDER	4/24/2022 1:47:37 AM	Order is deleted!
	Item item 24 update	UPDATE	ITEM	4/24/2022 1:26:53 AM	Reorder removed with quantity 2 an
	Supplier H1q6GaaF7l	UPDATE	SUPPLIER	4/24/2022 1:43:09 AM	Supplier Info is updated!
	Item item 22 update	UPDATE	ITEM	4/24/2022 1:17:39 AM	Quantity updated from 36 to 26
	New Item: item 25	ADD	ITEM	4/24/2022 12:29:34 AM	Quantity: 20, Location: 5, Type: No R
	New Item: item 27	ADD	ITEM	4/24/2022 12:41:45 AM	Quantity: 10, Location: 2, Type: No R
	Item item 24 update	UPDATE	ITEM	4/24/2022 1:34:31 AM	Reorder removed with quantity 10 a
	Item item 24 update	UPDATE	ITEM	4/24/2022 1:28:46 AM	Reorder removed with quantity 10 a
	Item item 24 update	UPDATE	ITEM	4/24/2022 1:27:16 AM	Reorder removed with quantity 10 a
	Item item 24 update	UPDATE	ITEM	4/24/2022 1:35:31 AM	Reorder removed with quantity 10 a
	Item item 24 update	UPDATE	ITEM	4/24/2022 1:34:27 AM	Reorder removed with quantity 10 a
	Supplier: H1q6GaaF7l	DELETE	SUPPLIER	4/24/2022 1:47:52 AM	Supplier is deleted!
	Item item 24 update	UPDATE	ITEM	4/24/2022 1:31:01 AM	Reorder removed with quantity 10 a
	New Order Test	ADD	ORDER	4/24/2022 12:44:06 AM	New order with id: 0nqCj36nbbR5ou
	Item item 24 update	UPDATE	ITEM	4/24/2022 1:28:50 AM	Reorder removed with quantity 10 a
	Item item 24 update	UPDATE	ITEM	4/24/2022 1:27:21 AM	Reorder removed with quantity 10 a
	Item item 24 update	UPDATE	ITEM	4/24/2022 1:28:44 AM	Reorder removed with quantity 10 a
	Item: y4w9ipJ73XEXl	DELETE	ITEM	4/24/2022 1:47:22 AM	Item is deleted!
	Updated Item: item 2	UPDATE	ITEM	4/24/2022 12:28:49 AM	Quantity: 10, MinQuantity: 5, Suppl
Item item 24 update	UPDATE	ITEM	4/24/2022 1:27:07 AM	Reorder removed with quantity 2 an	

Last Reload: 4/24/2022 1:48:43 AM

Figure 38. History tab

4 CONCLUSION

The target of this thesis was to build a simple desktop application for managing storage. Finally, the goal was met, and the application can run on the desktop. The code is available to the public at <https://github.com/vominhhuy71/Thesis>, so anyone can use it freely and customize it based on one's needs. Although this is a simple application for managing inventory, several improvements can be considered including:

- Adding more eye-candy and animation to the user interfaces. This will help creating a better user experience.
- Creating a redundant database inside the application in case the application isn't connected to the Internet.
- Adding more unit testing to increase the reliability of the application.
- Creating a dedicated API server for the application. This API server can be used for sending email to the supplier, solving some edge cases, such when two users modify the data at the same time, and adding authentication system to the application. This API server will undoubtedly reduce the workload of the application, but it will also increase the complexity of the application's architecture.
- The application can be enhanced to manage workload or documents, based on users' needs.
- The records of items imported and exported from inventory can be implemented with blockchain technology to maintain data integrity.

Throughout building the application, I have had the opportunities to apply everything I have learnt, including complicated concepts and techniques, into practice. To create this application, knowledge about desktop application, object-oriented programming, WPF's controls and data binding were required. Besides, architectural patterns like Model-View-ViewModel were necessary to ensure the scalability of the application. Additionally, I learned a lot about the advantages and disadvantages of Windows Presentation Foundation. It is a good framework for building a desktop application, but it can only work in Windows operating system.

REFERENCES

- Corob-Msft et al., 2021. Type conversions and type safety. WWW document. Available at: <https://docs.microsoft.com/en-us/cpp/cpp/type-conversions-and-type-safety-modern-cpp?view=msvc-170> [Accessed 8 March 2022]
- D. G. Chandra, R. Prakash & S. Lamdharia, 2021. "A Study on Cloud Database," *2012 Fourth International Conference on Computational Intelligence and Communication Networks*, 2012, pp. 513-519, DOI: 10.1109/CICN.2012.35. PDF Document. Available at: <https://ieeexplore.ieee.org/abstract/document/6375167> [Accessed 6 March 2022]
- George, A. D., & Coulter, D., 2021. Desktop Guide (WPF .NET). WWW document. Available at: <https://docs.microsoft.com/en-us/dotnet/desktop/wpf/overview/?view=netdesktop-6.0> [Accessed 13 March 2022]
- George, A.D., 2021. Introduction to WPF | Microsoft Docs. WWW document. Available at: <https://docs.microsoft.com/en-us/dotnet/desktop/wpf/introduction-to-wpf?view=netframeworkdesktop-4.8> [Accessed 2 April 2022].
- George, A.D. & Aaron, 2022. Data binding overview (WPF .NET) | Microsoft Docs. WWW document. Available at: <https://docs.microsoft.com/en-us/dotnet/desktop/wpf/data/?view=netdesktop-6.0> [Accessed 4 April 2022].
- George, A.D., 2022. Overview of WPF windows (WPF .NET) | Microsoft Docs. WWW document. Available at: <https://docs.microsoft.com/en-us/dotnet/desktop/wpf/windows/?view=netdesktop-6.0> [Accessed 4 April 2022].
- Hammad, M., 2021. Difference between WPF and Winforms. Web site. Available at: <https://www.geeksforgeeks.org/difference-between-wpf-and-winforms/> [Accessed 13 March 2022]
- Ibm.com n.d. What is a cloud database?. Web site. Available at: <https://www.ibm.com/cloud/learn/what-is-cloud-database> [Accessed 2 April 2022].
- Lander, R. et al., 2022. .NET class libraries. Microsoft Docs. WWW document. Available at: <https://docs.microsoft.com/en-us/dotnet/standard/class-libraries> [Accessed 21 April 2022].
- Jatana, N. et al., 2012. A Survey and Comparison of Relational and Non-Relational Database. PDF Document. Available at: <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.678.9352&rep=rep1&type=pdf> [Accessed 28 March 2022].
- Microsoft n.d. What is .NET? An open-source developer platform. WWW document. Available at: <https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet> [Accessed 8 March 2022]

Predicate delegate (system), n.d. WWW document. Available at: <https://docs.microsoft.com/en-us/dotnet/api/system.predicate-1?view=net-6.0> [Accessed 9 April 2022].

Schaefer, L., n.d. What is NoSQL? NoSQL databases explained. MongoDB. Web site. Available at: <https://www.mongodb.com/nosql-explained> [Accessed 16 April 2022].

Smith, J., 2009. Patterns - WPF apps with the model-view-viewmodel design pattern. E-magazine article. MSDN Magazine Issues. Available at: <https://docs.microsoft.com/en-us/archive/msdn-magazine/2009/february/patterns-wpf-apps-with-the-model-view-viewmodel-design-pattern> [Accessed 5 April 2022].

Wagner, B. et al., 2022. A Tour of C# - C# Guide | Microsoft Docs. WWW document. Available at: <https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/> [Accessed 8 March 2022]

What is a database? Oracle, n.d. Web site. Available at: <https://www.oracle.com/database/what-is-database/> [Accessed 28 March 2022].

What is .NET framework? A software development framework, n.d. WWW document. Available at: <https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet-framework> [Accessed 10 April 2022].

Wikipedia contributors, February 2022. .NET Framework. WWW document. Available at: https://en.wikipedia.org/w/index.php?title=.NET_Framework&oldid=1071023204 [Accessed 8 March 2022]

Wikipedia contributors, March 2022. C Sharp (programming language). WWW document. Available at: [https://en.wikipedia.org/w/index.php?title=C_Sharp_\(programming_language\)&oldid=1075778688](https://en.wikipedia.org/w/index.php?title=C_Sharp_(programming_language)&oldid=1075778688) [Accessed 8 March 2022]

Wikipedia contributors, February 2022. Windows Presentation Foundation. WWW document. Available at: https://en.wikipedia.org/w/index.php?title=Windows_Presentation_Foundation&oldid=1072647910 [Accessed 13 March 2022]

Wikipedia contributors, February 2022. Architectural pattern. WWW document. Available at: https://en.wikipedia.org/w/index.php?title=Architectural_pattern&oldid=1069477266 [Accessed 23 March 2022]

Wikipedia contributors, February 2022. Model-view-viewmodel. WWW document. Available at: <https://en.wikipedia.org/w/index.php?title=Model%E2%80%93view%E2%80%93viewmodel&oldid=1074534894> [Accessed 27 March 2022]

LIST OF FIGURES

Figure 1. The architecture of the .NET Framework (What is .NET Framework...)	7
Figure 2. An example of a desktop application	9
Figure 3. An example of creating user interfaces with WPF using XAML	10
Figure 4. An example of code-behind file	10
Figure 5. Data binding (George & Aaron 2022)	12
Figure 6 Components of a Window (George 2022)	13
Figure 7. Window life events (George 2022)	14
Figure 8. Model-View-ViewModel connection (Model-view-viewmodel 2022)	15
Figure 9. An example of handling mouse clicks event.	15
Figure 10. Item class	19
Figure 11. Order class	20
Figure 12. Populated Order class	20
Figure 13. Supplier class	21
Figure 14. Data stored in the Global class.	21
Figure 15. Method for fetching Order from Firestore	22
Figure 16. Log class	23
Figure 17. INotifyPropertyChanged base class	24
Figure 18. OnPropertyChanged implementation	24
Figure 19. Inheritance Hierarchy (Smith 2009).	25
Figure 20. DataContext binding	26
Figure 21. ManagerWindow.xaml	26
Figure 22. ManagerViewModel class	27
Figure 23. CurrentView property	27
Figure 24. RelayCommand class inherits from ICommand	28
Figure 25. RelayCommand implementation	29
Figure 26. DispatcherTimer	29
Figure 27. Reorder function	30
Figure 28. App.xaml	31
Figure 29. Home Tab	31
Figure 30. Item and Reorder rule	32
Figure 31. Order Tab with New Order window	32
Figure 32. View Order	33
Figure 33. Reorder email to supplier	33

Figure 34. Supplier Tab	34
Figure 35. Searching for Item	34
Figure 36. Firestore database	35
Figure 37. Unit testing	35
Figure 38. History tab	36