**[Module 8] Portfolio Project: Inventory Database Management System**

Lee Ellington

Colorado State University Global

CSC480: Computer Science Capstone, Design Lab

Instructor: Dr. Lori Farr

Due Date: 4/10/22

**[Module 8] Portfolio Project: Inventory Database Management System**

To better understand what is involved with the development of a software system, a business need for a software system was identified, a solution designed, and a software system implemented for the CSC480 capstone project. This report details the development of an inventory database management system for the Technology and Facilities Department of the School of Management at the University of Texas at Dallas. First, the topic approval process is covered. Next, a description of the project and the system to be designed is explored. UML diagrams of the system’s initial design follow. The system’s fault tolerance was tested during the next phase of the project. Both the testing process and results are provided. Lastly, the details of the final implementation of the software system developed are given.

**Topic Approval**

The decision to develop an inventory database management system was straightforward. Working in the IT department of a university that still relies on a Microsoft Excel spreadsheet to track university assets, the identification of the business need to modernize the management of asset inventory data was obvious. The topic proposal detailed the fact that the IT department is responsible for the tracking of inventory of many assets, in addition to regular auditing. The number of assets currently within the department is consistently over several thousand. With so many assets to track, a software system that allows the management of a database would yield many benefits. A simplified interface could make the addition, deletion, editing, and searching for asset data within the database easy and accessible to even the lowest level technicians. Many problems with the Excel spreadsheet exist. Namely, only one person can have the file open at any time to make changes, searching for asset data can be tedious, and even the unintended loss of asset data. Lastly, initial thoughts on system implementation included simplistic user interface design, Java Swing based desktop application, and a MySQL database to hold the data. Dr. Farr approved the topic as it was both researchable and an interesting application. Therefore, the project proceeded to the next phase - a project proposal where a clear description of the project with an estimate of the implementation timeline was needed.

**Proposal**

A summary of the purpose of the capstone project was needed and no better way to get a grasp for the need of a software then to understand the business need that it addresses. The project proposal consisted of four parts. Specifically, a description of the business problem was given, followed by a description of the information system solution designed to address the problem, a discussion of the major software components that were anticipated to be needed to build the solution, and a tentative implementation timeline.

**Business Problem**

As stated, the IT department of the School of Management at the University of Texas at Dallas is responsible for the tracking of inventory of many assets and regular auditing of those assets. The type of assets that are tracked include things like desktop computers, laptops, server equipment, classroom technology equipment, data closet equipment, and some furniture items. Each one of these assets receives a tag number upon purchase that is used throughout its lifetime for tracking purposes. The current inventory system consists of a singular Microsoft Excel spreadsheet that is saved on an onsite server that the department itself manages. Redundant paper records are also kept as back up if information within the spreadsheet become corrupted or lost. Drawbacks to this system include that the server allows only one user at a time to access and edit the file. There is no way to easily query the data and pull up information about an asset. Furthermore, the process to edit, delete, or add a new asset is time consuming and can lead to issues if user error occurs.

**Information System Solution Description**

Initial thoughts on the information system solution are that the inventory Excel spreadsheet is to be converted to a simple MySQL database. Relational database schemas are ideal for this solution as they allow for easy storing, organizing, and defining of various tables for database information (Murach, 2019). This allows for an easy one-to-one translation from the current system to the new system. To ensure that a completed product can be delivered, the system will only contain one table that will have all the inventory information. This set up might even be the final design as for inventory tracking not many other relations are needed in the database schema. The inventory information that will be pertinent for this implementation will include the fields asset number, device type, serial number, owner name, location, and date added.

Regarding the user interface, a GUI front end is proposed to be developed utilizing Java Swing utilities to provide a user-friendly way to interact with the database. Swing components will be used to build a basic GUI that the technicians can easily exploit for daily inventory tasks. The GUI application will provide ways for the technicians to search, edit, delete, or add an asset to the system. There will be a section that displays search results, a section to input data to be added, and then buttons that allow for the editing or deletion of asset information option.

**Major Software Components**

MySQL Workbench is to be utilized for the initial construction of the inventory database. A sample set of data consisting of data on 30 assets is to be used for testing purposes. The Workbench can also be used to perform test queries to ensure that the desired behavior has been achieved before moving on to the Java side of the project.

Once the database has been established, the IDE VS Code developed by Microsoft is to be used for the development of the application. The logic of the application will consist of several classes. One class will be responsible for managing the database queries. The Java Database Connectivity API will need to be utilized to define how the client accesses the MySQL database. For purposes of this project, a generic user will be defined for access to be granted to the database. Another class will be responsible for holding the GUI components of the application. The Java Swing components that most likely will need to be used include JFrame, JPanel, JButton, JLabel, JTextField, JMenu, JSlider, and JTable.

**Implementation Timeline**

With four weeks in the course remaining there remains a lot of work to do. Week 5 of the course is to be used for planning of functions and UI design. Week 6 is to see construction of the database and testing with the sample data set completed. Week 7 the logic classes of the Java application should be completed and finally in Week 8 the GUI class, as well as the entirety of the project, should be completed.

**System Design**

The next phase of the capstone project was creation and finalization of a detailed design for the inventory database management system. A system summary and system architecture for the both the initial design and final design as implement are provided.

**Initial Design – System Summary**

The Java GUI application is to be constructed in a modular way so that it can easily be upgraded in the future (Keeling, 2017). The final software delivered at the end of this course will resemble a barebones platform that proves basic functionality and completed in such a way that it can be improved upon in the future easily because of its modular design. Once sufficiently refined in the future, this solution is to be deployed by the IT department at SOM UTD. Every asset entry within the database is to have its own unique asset number. The other fields associated with this unique asset number, or primary key, includes the asset’s device type, serial number, owner name, location, and date added.

The expected functionality of the new inventory system is as follows. The user launches the desktop Java GUI application. A window appears that has fields for entry (asset number, device type, serial number, owner name, location, and date added), buttons for various options (search, add, edit, and delete), an area to display search results, and an area within the window for error message display. Only an asset number, since it is unique, is needed for the search functionality. Once an asset has been found the edit and delete buttons are to be made available for use. Appropriate error messages for situations such as asset not found, input validation, and database issues are to be provided to the user. Furthermore, the state of the system throughout execution is either busy or ready for user input. This is not meant to be a multithreaded application that allows the user to do work while work is being performed. That is, the application is in a responsive state initially, is unresponsive after taking user input (button), and become responsive again after displaying results back to the user.

The inputs to the system are relevant inventory information for a unique asset within the inventory system. For adding new assets every field must be included, but for simple search functionality only an asset number is provided to the system. In addition to textual input, the user will interact with buttons within the GUI to execute the appropriate action. The output will either be an error message or the display of a row from the inventory database. In the future, additional functionalities will be added that can perform a variety of other MySQL queries and display the corresponding tables to the user. It is expected that these improvements would change both the required inputs and the resulting outputs from the system.

**Initial Design – System Architecture**

The new inventory system will consist of several components that operate independently but interact with one another. The modularity of this design facilitates easy implementation of future improvements to the system.

The MySQL relational database is the key component of the inventory system and can be considered the backbone of the system as it is where the asset data is to be stored permanently. Without a properly designed and implemented database the entire system’s functionality is compromised. Utilizing MySQL Workbench, a sample inventory database is to be constructed. For testing purposes, the database is to be hosted locally on the computer used for development of the GUI application. A generic user with simple log in credentials will be used to make integration with the Java GUI application smoother (Murach, 2019). Initially, the database will contain only one table that will hold all the inventory data within its rows, as the code in *Figure 1* below describes.

Text

Description automatically generated

*Figure 1:* MySQL Database Inventory Table Definition of Fields and Data Types

After creation of the database schema and the addition of the inventory table, sample data from 30 example devices will be added to the table using the MySQL Workbench application. The database will then be subjected to various test queries before being deemed ready for integration with the Java GUI application.

The desktop Java application is the GUI front in for the database-based inventory system and is to consist of two main parts. The logic component will contain all the methods and definitions needed to carry out the required logical operations of the application, such as the MySQL queries to add or find an asset within the database. The graphical component will contain all the relevant code to define the graphical user interface of the application. The link between the two components is to be treated as an API so that a standard is created to allow changes to be made easily.

For the logic component, several classes will be utilized to separate functionality. The Main class will be utilized for connecting GUI components with their corresponding logic actions, controlling the flow of the application, and defining all logic operations that correspond to certain action events. The Asset class will be used to encapsulate the data as it moves from the database to the application, or vice versa. It will be a simple class with private data members corresponding to the fields of the database inventory table, in addition to the appropriate set and get methods. These methods will be called within the Main class as needed. It is the only object being utilized within the Java application for storage of data, otherwise all changes are made and read directly from the MySQL inventory database table. Lastly, the SQLQueries class will contain all the code needed to make queries to the inventory table of the database. The types of queries that will need to be executed include SELECT and INSERT INTO. The methods included in this class can be expanded upon later to add more SQL functionality to the application.

To connect to the inventory database, the Main class will utilize the appropriate Java Database Connectivity driver while utilizing the resources imported with the java.sql.\* libraries. The process is relatively straightforward for connecting to a MySQL database when the username and password or set to generic credentials. A connection via the DriverManager class is to be created using the Connection Class and the database connection URL. Every interaction with the database is to be within a try/catch clause so that errors can be handled gracefully. ActionListeners are to be utilized for each button and when pressed the corresponding action is to be performed as defined with the Main class. This means that if the add button was pushed, all the text fields will be read, validated, and then thee correct SQLQueries method call made to perform the addition to the inventory table of the database.

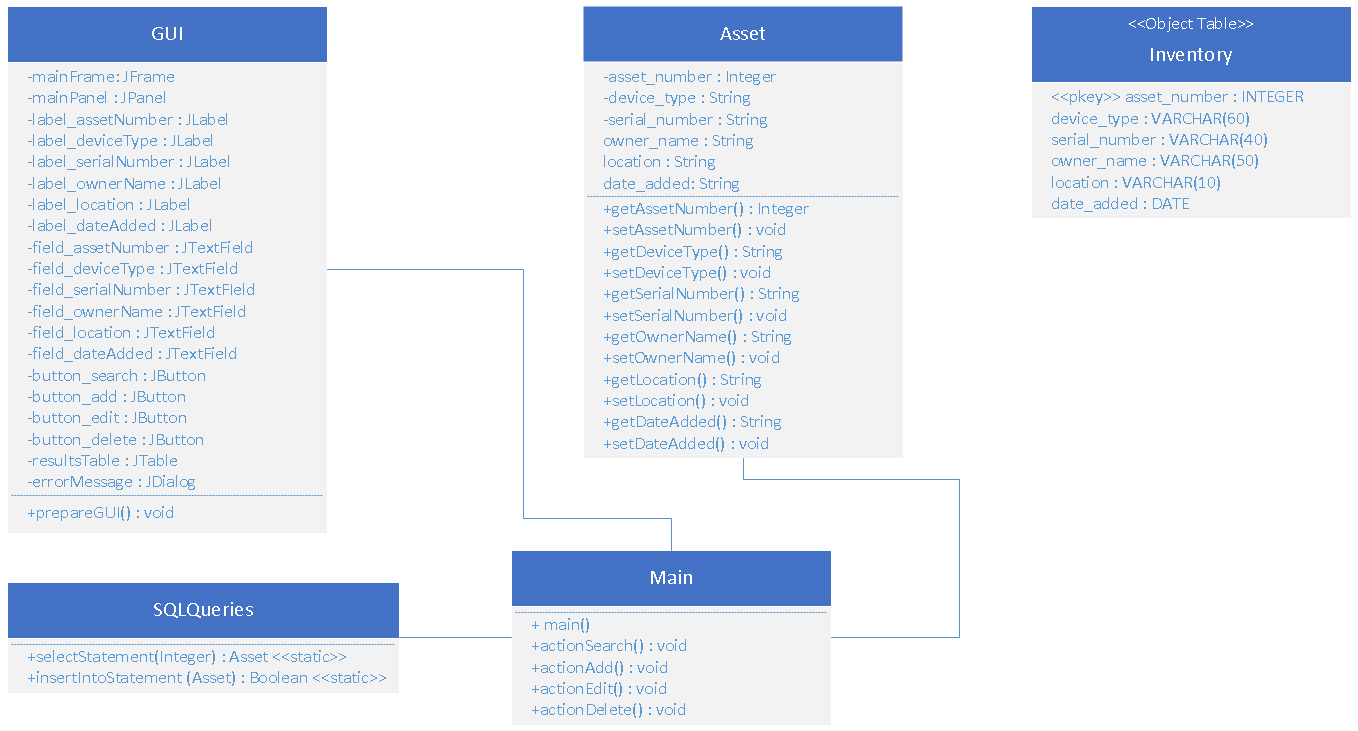
The GUI class defines the graphical layout of the application and will utilize the resources available through the Swing library. The JButton component will be used for user input on what action to perform and JTextField to get textual user input. JTable will be used to display result data back to the user, JLabel used to label GUI components, and JDialog to display error messages to the user. JFrame will be utilized to hold the entire application in a window and JPanel to organize components within the frame. These various components will be arranged in such a way as seen below in *Figure 2*.

Graphical user interface, application

Description automatically generated

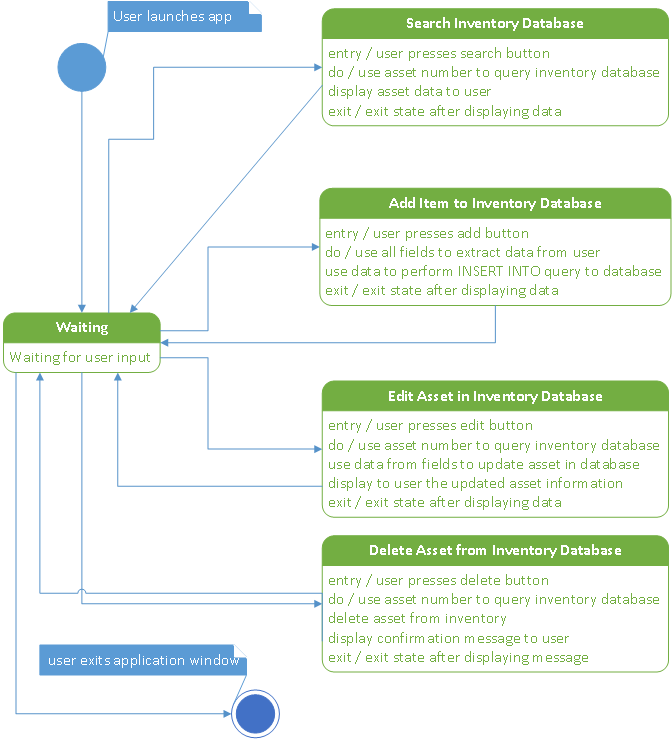
*Figure 2:* Example of Java Application GUI Layout

**Initial Design - UML Class Diagram**



*Figure 3*: Inventory Management System UML Class Diagram

**Initial Design – State Machine Diagram**



*Figure 4*: Inventory Management System State Machine Diagram

**Final Design – System Summary**

While the initially proposed system’s design was sound, certain improvements were made to the version of the software system solution implemented. Instead of utilizing Java Swing, JavaFX was chosen as the graphics utility to be used. A successor to Swing, JavaFX offers an ability to bind properties and has a built-in separation of logic (Controller) and visual components (FXML) that aid in development (Kruk et al., 2017). With the change to the graphical framework being used, so too did the architecture of software system.

The GUI of the application was also reimagined to allow for more functionality for the user. The primary scene contains a UTD logo, a FXML Label title of the application., and five FXML Buttons that provide the user with options on how to manage the assets within the database. The initial search scene presents the user with a labeled FXML TextField for the user to input the asset number of the asset they are looking for. Two buttons that allow the user to choose between searching or returning to the main menu (primary scene) are provided. If the user inputs an incorrectly formatted asset number, or the asset is not in the database, they are given an error message. A results scene is loaded that uses a FXML TableView to present the data of the asset searched for. The same FXML Buttons and FXML TextField remains for the user to perform additional searches if desired. The add action load the add scene where there is an FXML Label and corresponding FXML TextField for each of the data points for an Asset object. Underneath, the FXML Buttons for executing the add action or returning to the main menu are presented to the user. The add scene provides input validation with helpful error messages to the user. The edit action first shows an asset confirmation scene where the user is presented with a FXML Label and FXML TextField to input the asset number of the asset they wish to edit. The FXML Buttons below allow for the user to choose the confirm asset action or return to the main menu. Input validation and appropriate error messages are presented to the user within this scene. Once an appropriate asset number has been entered, the supplementary edit results scene is loaded. This scene has a FXML TableView that displays the data currently associated with the asset number in the database. Below that, FXML Labels and corresponding FXML TextFields to enter edits for the device type, serial number, owner name, location, and date added attributes of the asset are provided. Lastly for this scene, two buttons that allow the user to choose between executing the edit action or to return to the main menu are given. As with other scenes, input validation and descriptive error messages are included. The delete scene follows a similar format to other scenes. An FXML Label and corresponding FXML TextField for asset number user input is first. Two FXML Buttons that allow the user to choose between executing the delete action or to return to the main menu are provided. Appropriate input validation and error message display to the user are supplied. Lastly, the display all scene retrieves all the asset data from the database and displays it in an FXML TableView. Below the table an FXML Button gives the user the ability to return to the main menu. Luckily the design of the MySQL database was sound and did not need to be altered for the final implementation.

**Final Design – System Architecture**

***Java Application Classes***

A main class App is the driving class of the application. It not only initializes and loads the initial JavaFX scene to present to the user, but it also contains two utility methods that are used throughout the application by other classes to load appropriate scenes to be displayed. Within the main() method of the App class, an Asset and SQLQueries object are initializes before launching the application window. These objects are persistent throughout the runtime of the application. The workingAsset object is used for the passing of data while the sqlQueries object is used to make various SQL query method calls.

The Asset class is a simple class that is used for data encapsulation. It consists of private data members, a default and parameterized constructor, and the needed set and get methods for the data members.

The SearchController class is used to control what happens on the search.fxml scene. It also defines and attaches shadow effects to several FXML components within it’s initialize() method. Within the search() method the user is guided through input validation and when valid input is received the class calls the setRoot() method to chance scenes to the results.fxml scene.

The ResultsController class is used to control what happens on the results.fxml scene. Namely, an asset number is entered by the user and a search is performed within the inventory database for that asset. This class defines a DropShadow effect for FXML Labels, TextFields, and TableViews, and then assigns them to the appropriately linked FXML components. The TableView component is set up with the appropriate columns, height properties, and initialized. The call to the sqlQueries searchDatabase\_assetNumber() query. If the asset was found then the information is added to the TableView, otherwise an error message is presented to the user. Lastly, in the resultsSearch() method the user input is validated and then stored within the workingAsset object. The results.fxml scene is then reloaded to display the data corresponding to the new search attempt.

The AddController class is used to control what happens in the add.fxml scene. Appropriate FXML links are established and then DropShadow effects are defined within the initialize() method. DropShadow effects are then added to the FXML TextFields within the scene. The add() method takes in the input validated data from the user and attempts to add it to the MySQL database. It utilizes a SimpleDateFormat object for confirming appropriate date input and then a series of if statements that throw exceptions for the other field’s input validation. If the call to the sqlQueries addAsset() method is successful, the user is notified and the text fields and workingAsset fields are reset to allow for another asset to be added to the database.

The EditController class is used to control what happens in the edit.fxml scene. Much like the search scene, the user must first confirm that an asset exists within the database before it can be edited. Appropriate FXML links are made with the TextField and Label. The initialize() method is used to define two different DropShadow effects and assign them to the FXML linked components. The confirimAsset\_edit() method performs input validation on the asset number input and stores the appropriate input in to the workingAsset’s asset number field. The results\_edit.fxml scene is then loaded.

The ResultsEditController is used to control what happens in the results\_edit.fxml scene. First, links to the needed FXML components are made. Within the initialize() method, DropShadows for TextFields, Labels, and a TableView are defined and assigned to the appropriate components. The TableView then is initializes with the appropriate columns and attributes. The database is queried using the sqlQueries searchDatabase\_assetNumber call. If the asset is found then the asset data is loaded to the TableView and presented to the user, if not an error message is provided. The editAsset() method allows the user to edit an asset that has already been found to exist within the database. A SimpleDateFormat object is used to confirm proper date input from the user. If statements that throw exceptions are used to validate the other input fields. A sqlQueries call to editAsset() is made. Whether the asset was successfully edited in the database or not, the user is notified.

The DeleteController class is used to control what happens in the delete.fxml scene. A simple scene, a FXML Label and corresponding TextField for asset number input is provided to the user. These objects are first linked within the class declaration. In the initialize() method a DropShadow effect for the FXML Label and TextField is defined and assigned appropriately. The deleteAsset() method deletes an asset from the inventory database through taking in the asset number from the user and using it to call the deleteAsset() method from SQLQueries. The user is notified for both success and failure to delete an asset.

The DisplayAllController class is used to control what happens within the display\_all.fxml scene. First, a link to the FXML TableView is established. Within the initialize() method a DropShadow effect for the TableView is defined and assigned to the linked FXML TableView. The TableColumns are initialized and assigned to the TableView with the appropriate display attributes. The data base is queried for all assets currently within the database by a call to the sqlQueries getAllAssets() method. The asset data returned from that method call is used to set the items of the FXML TableView.

The SQLQueries class contains methods that allow for the connection to and queries of the hardcoded url MySQL inventory database. It first defines a private static final String for the database url, username, and password. The searchDatabase\_assetNumber() method takes in an Integer number and uses it to search the database for an asset entry with a matching asset number. If the returned result set from the statement execution is empty, the method returns false, but if there is data in the returns result set then the data is added to the workingAsset object, and the value of the return Boolean is changed to true. The addAsset() method uses the data in a passed in Asset object to add an asset to the MySQL database inventory table. After making connection to the database, it attempts to make the prepared update statement to the database. If one ore more rows were affected by the updated, the asset added Boolean is changed to true and then it is returned. The deleteAsset() method uses the asset number from a passed in Asset object to delete an asset from the MySQL database inventory table. After establishing a connection to the database, the prepared delete statement is executed. If one or more rows were affected by the query, then the asset deleted Boolean is updated to true and returned. The editAsset() method uses the data from a passed in Asset object to update the columns of an asset that already exists within the database table. After making a connection to the database the prepared update statement is executed. If at least one row in the table was affected by the query, then the asset edited Boolean is changed to true and returned. Lastly, the getAllAssets() returns a ObservableList<Assets> that contains data for all the assets currently within the database table. After making a connection to the database the prepared query statement is executed. While there are objects within the returned result set a new Asset object is created, updated with the result set data, and added to the ObservableList<Assets> that are to be returned out off the method.

***Java Application FXML***

As implemented, the .fxml files that are the GUI piece of the JavaFX framework define the style and layout details for the application. Full use of FXML GridPanes, HBoxes and attributes such as style, onAction, and alignment made creating an aesthetically pleasing (*Figure 5*) and easy to use UI for the application simple.



*Figure 5*: Main Splash Page of Application – primary.fxml

***MySQL Database***

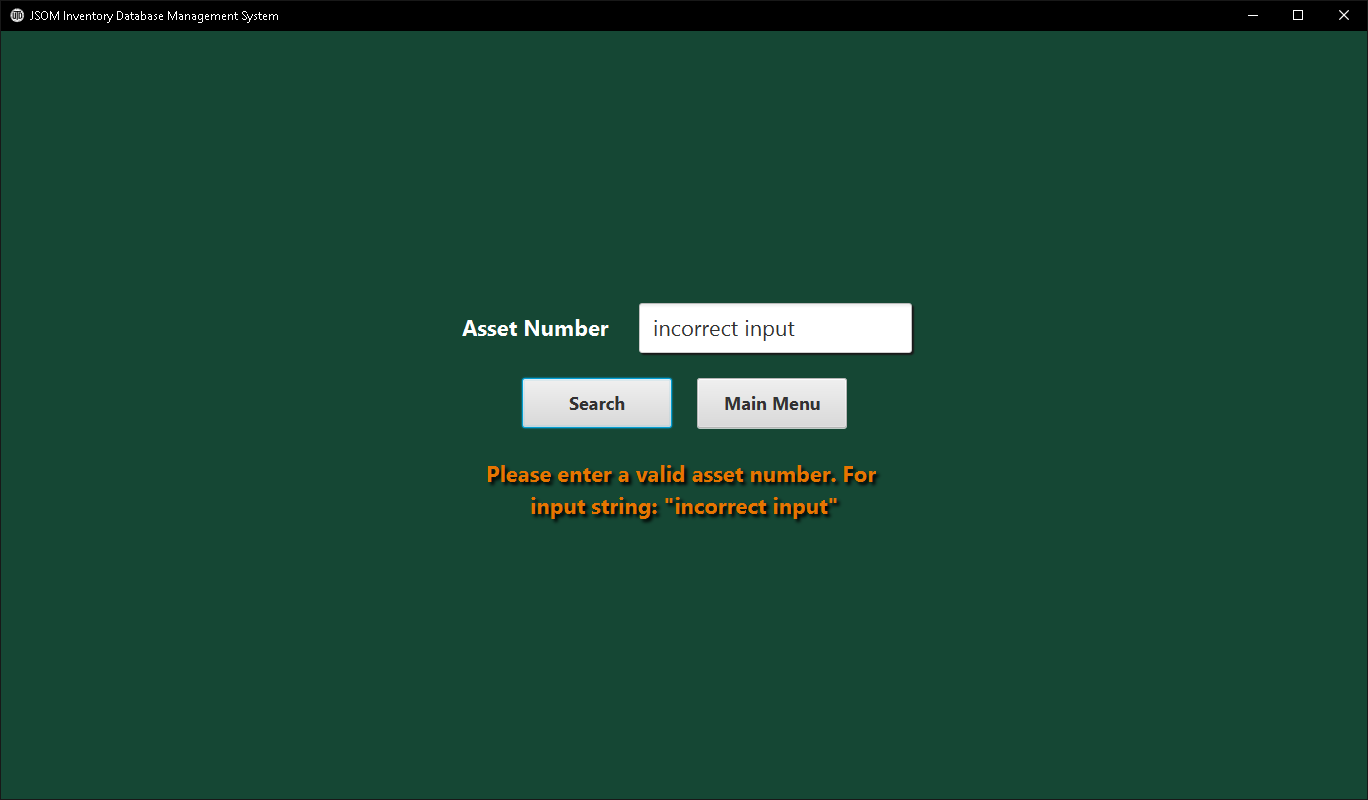
The MySQL database remained the same except for how the test data was uploaded. Instead of manually entering test data, an .csv file was made, and MySQL Workbench was used to import asset data from that .csv file into the inventory table. 431 assets in total were added to the database as a test set.

**System Test**

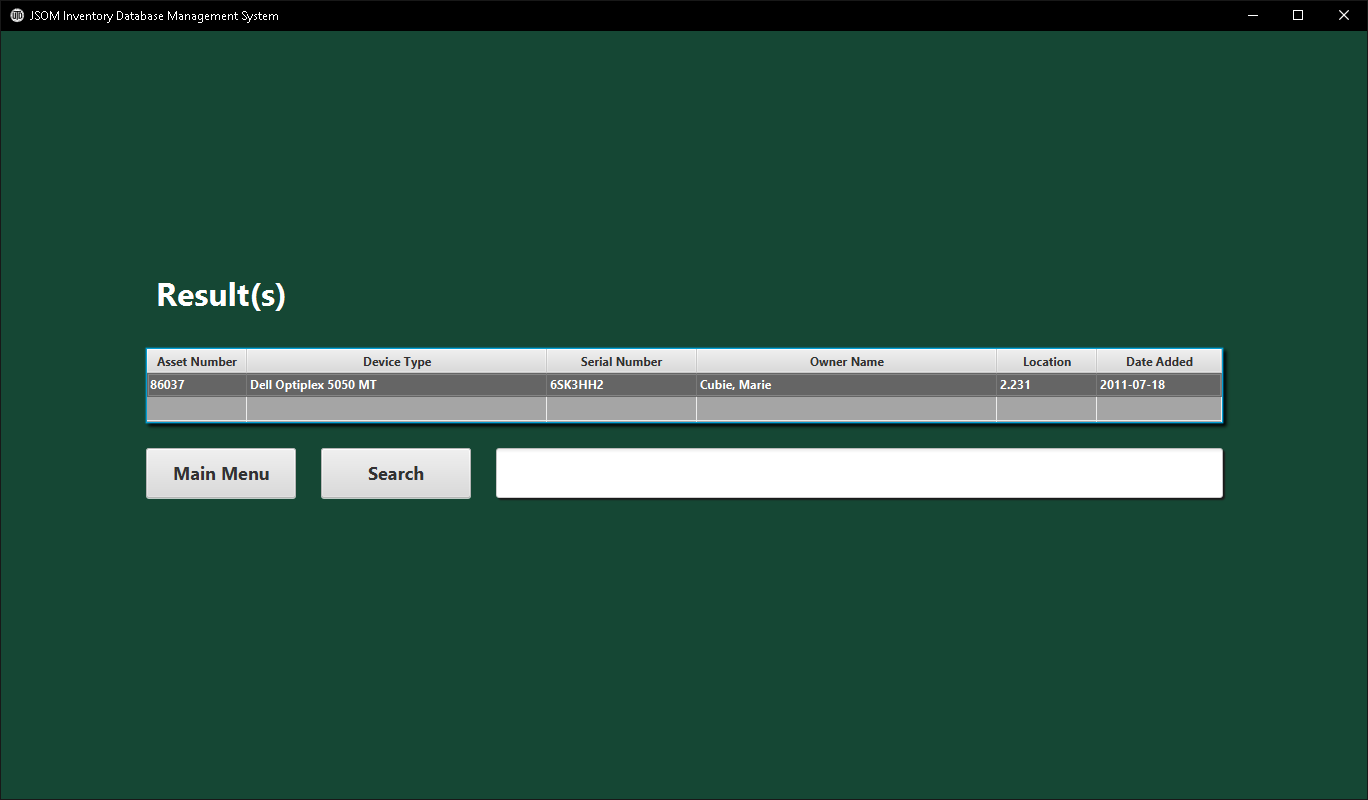
A software is fault-tolerant when a set of software facilities have been put in place to detect and recover from faults that are not handled by underlying hardware or operating system mechanism (Huang & Kintala, 1993). To ensure that the inventory database asset management system was fault-tolerant, input validation and exception handling was integrated so that the application could recover from issues gracefully (Baresi & Pezzè, 2006). To test these fault-tolerant measures testing was conducted. Following the testing plan proposed in week 6 of the course, the capstone project would implement the testing methodologies of unit testing, integration testing, system testing, and performance testing.

***User Cases – Test Case 1: Search***

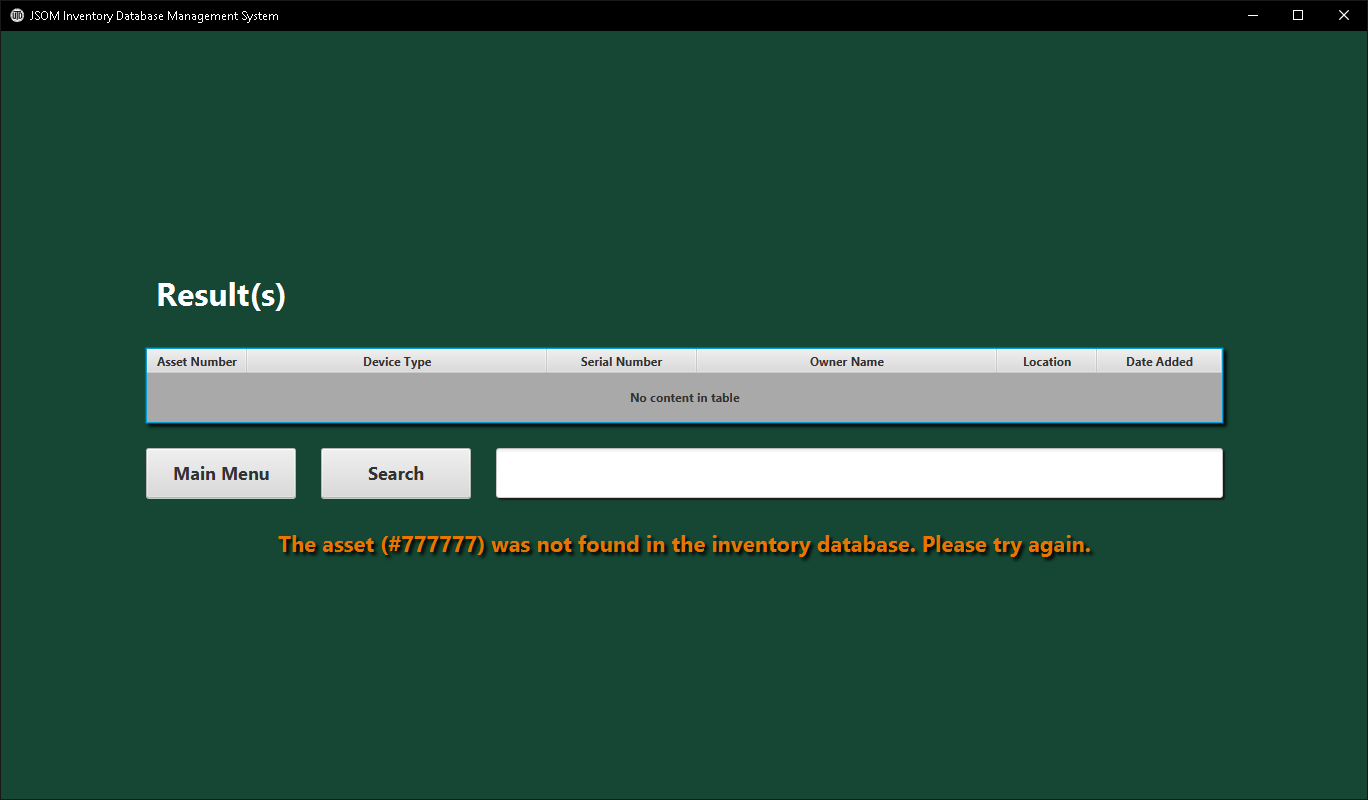
User inputs an asset number for an asset they wish to find within the database. An error message is presented if the asset number is in an incorrect format. If accepted, a results scene loads that has a table to display the asset data to the user. An appropriate error message is displayed if the asset is not in the data base. The user can then search for another asset or return to the main menu.



*Figure 6*: User Input Validation – search.fxml



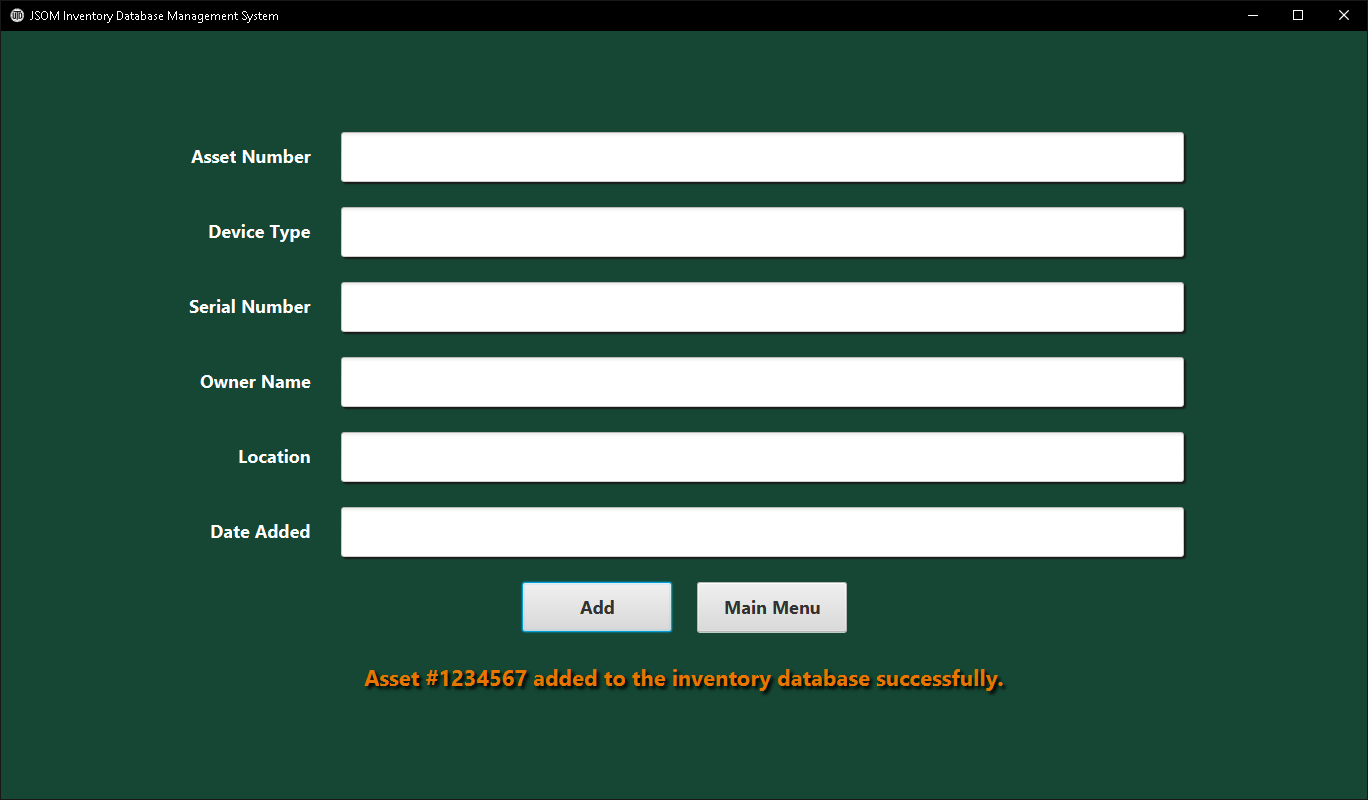
*Figure 6*: Search Results Scene with Valid Asset Number Found in Database – results.fxml



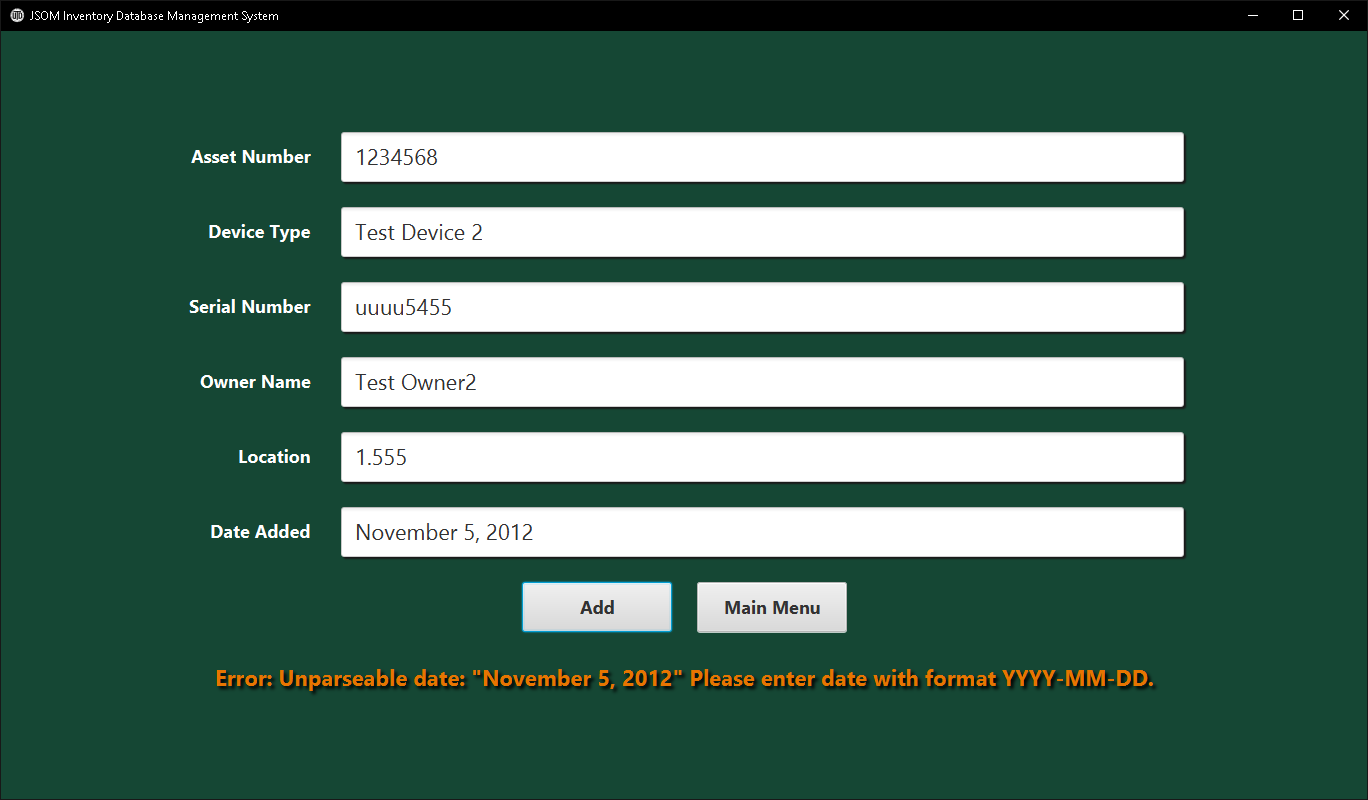
*Figure 7*: Search Results Scene with Valid Asset Number Not Found in Database – results.fxml

***User Cases – Test Case 2: Add***

The user adds an asset to the inventory database. Input validation for incorrect field entries by the user is given, as well as a success confirmation message when an asset has been added successfully to the database.



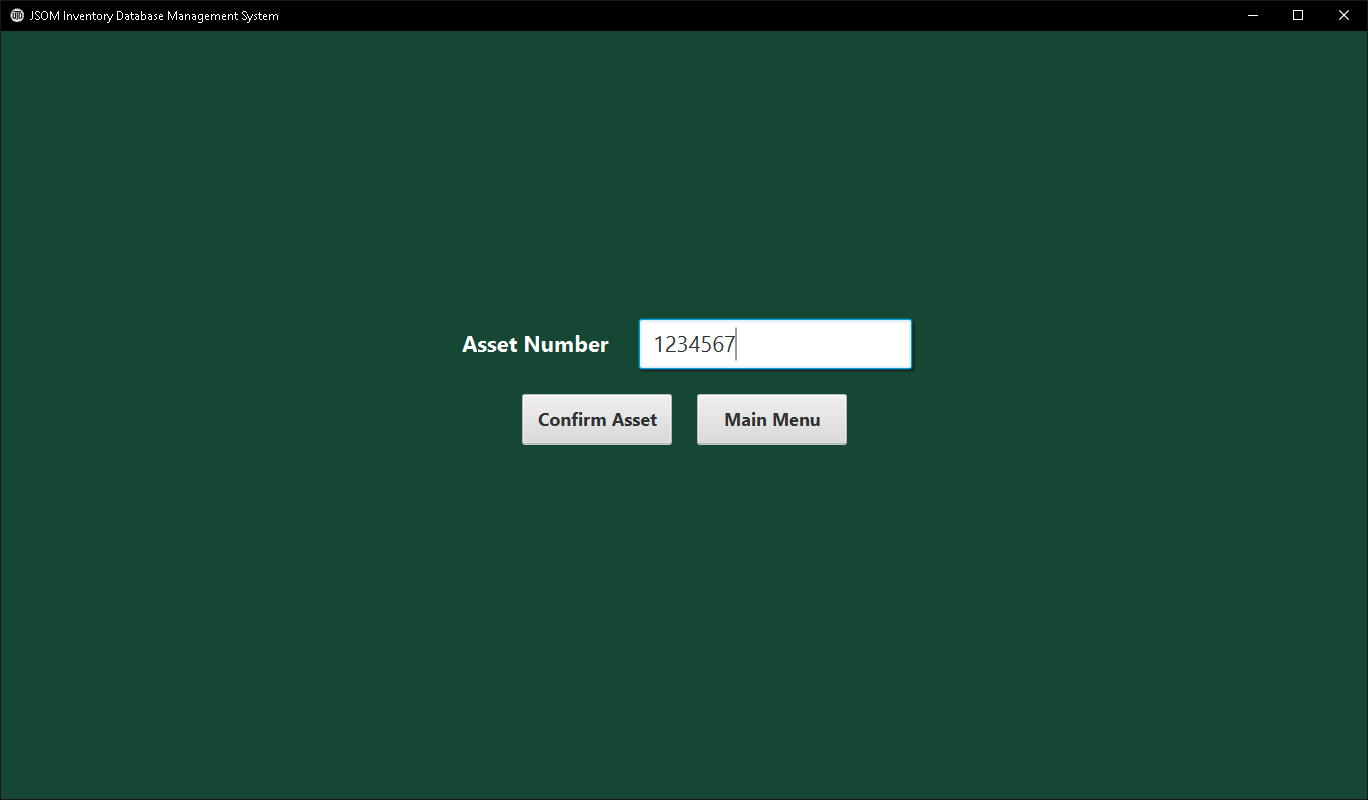
*Figure 8*: Add Asset Scene with Asset Successfully Added – add.fxml



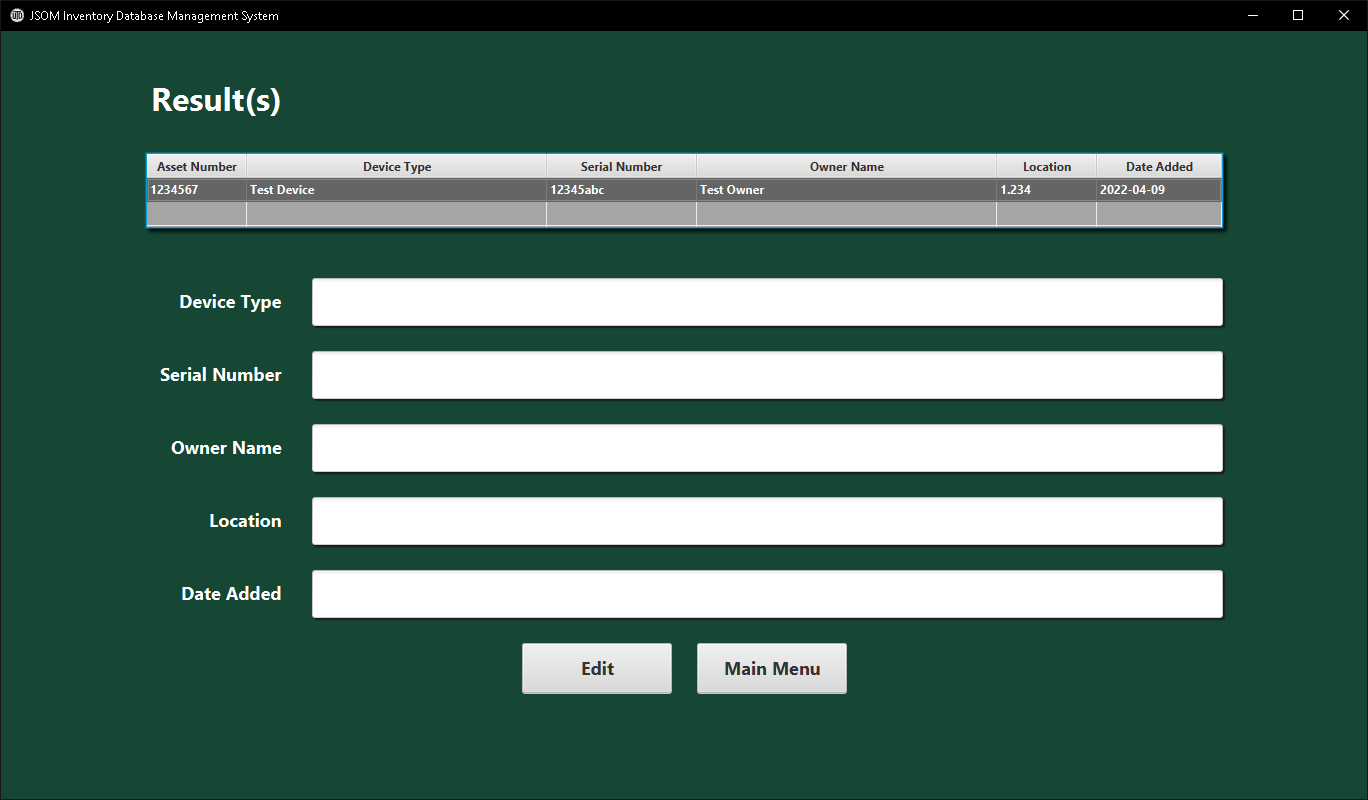
*Figure 9*: Add Asset Scene with Input Validation Error Message – add.fxml

***User Cases – Test Case 3: Edit***

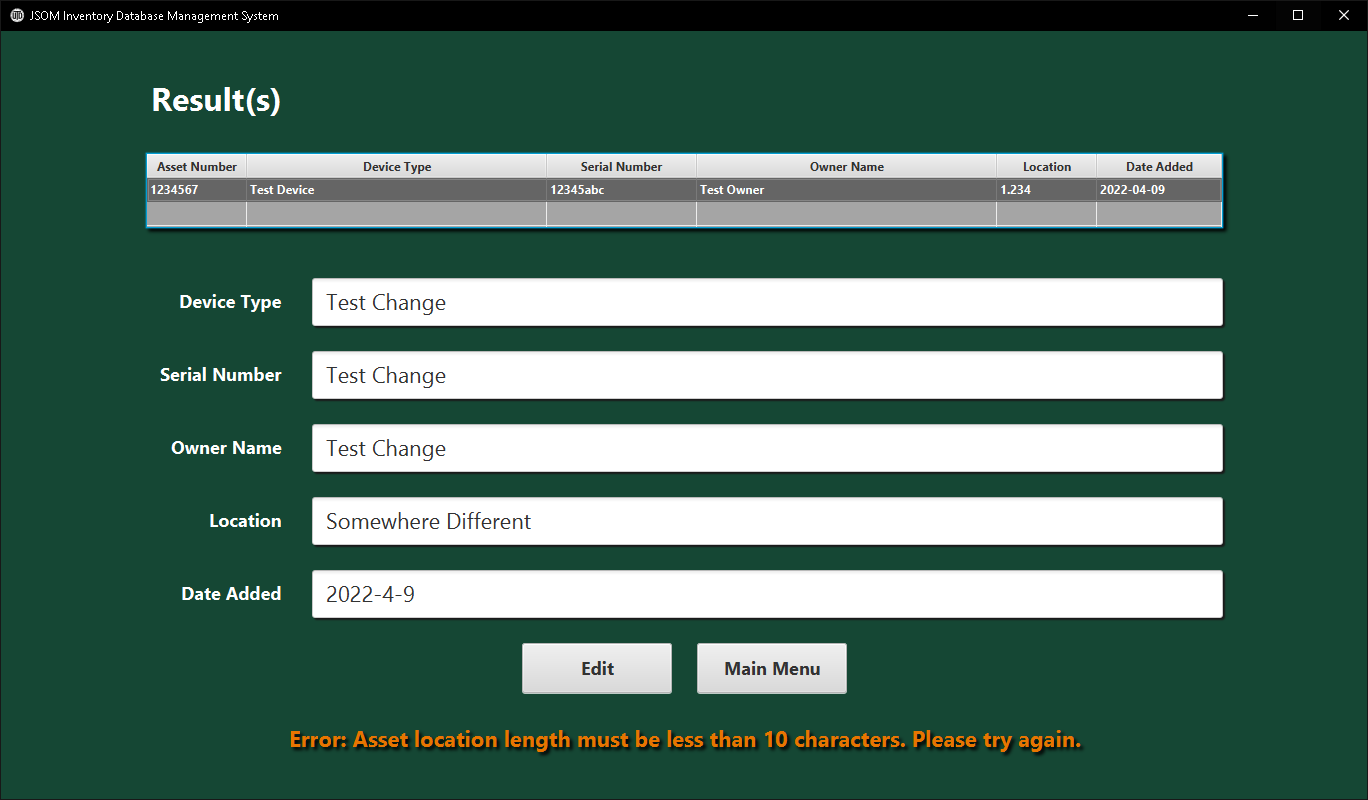
The user first is presented with the confirm asset scene after clicking the Edit button from the primary scene. After entering an asset number of an asset within the database, the results\_edit.fxml scene is loaded. The data associated with the asset number are displayed and text fields are presented to the user for input on what changes to the asset need to be made. Appropriate input validation and clear error/success messages are present.



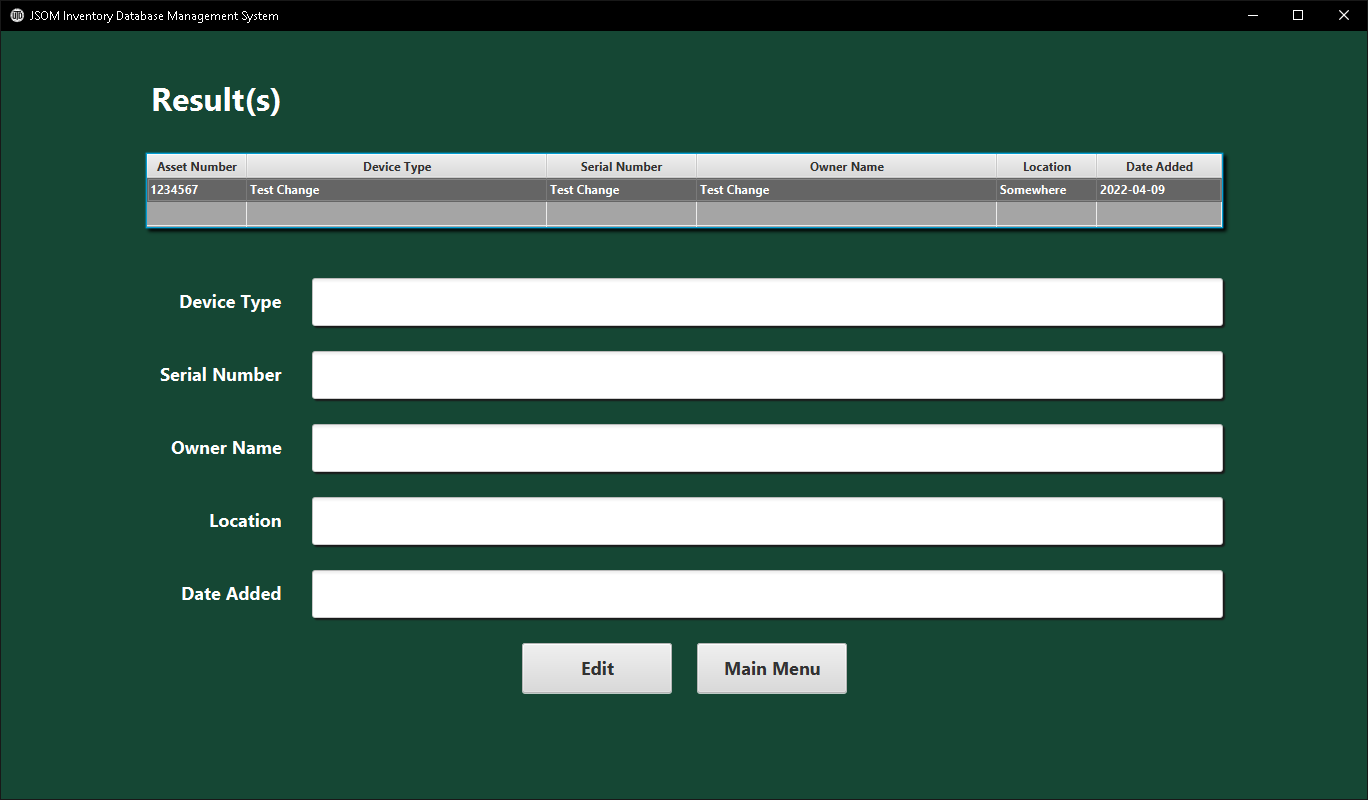
*Figure 10*: Edit Asset Confirmation Scene with Valid Asset Number– edit.fxml



*Figure 10*: Results Edit Asset Confirmation Scene Displaying Asset Data – results\_edit.fxml



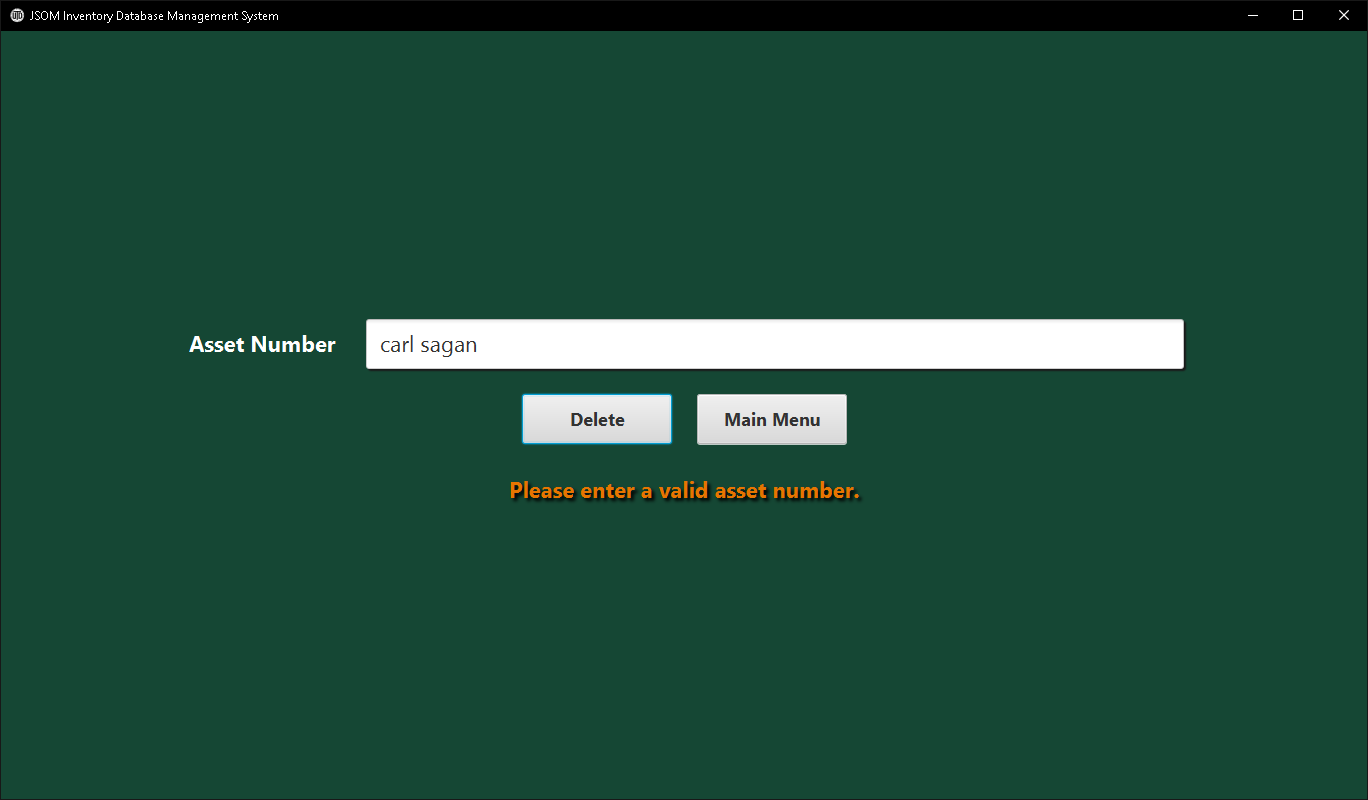
*Figure 11*: Results Edit Asset Confirmation Scene Displaying Asset Data Edit Input Validation – results\_edit.fxml



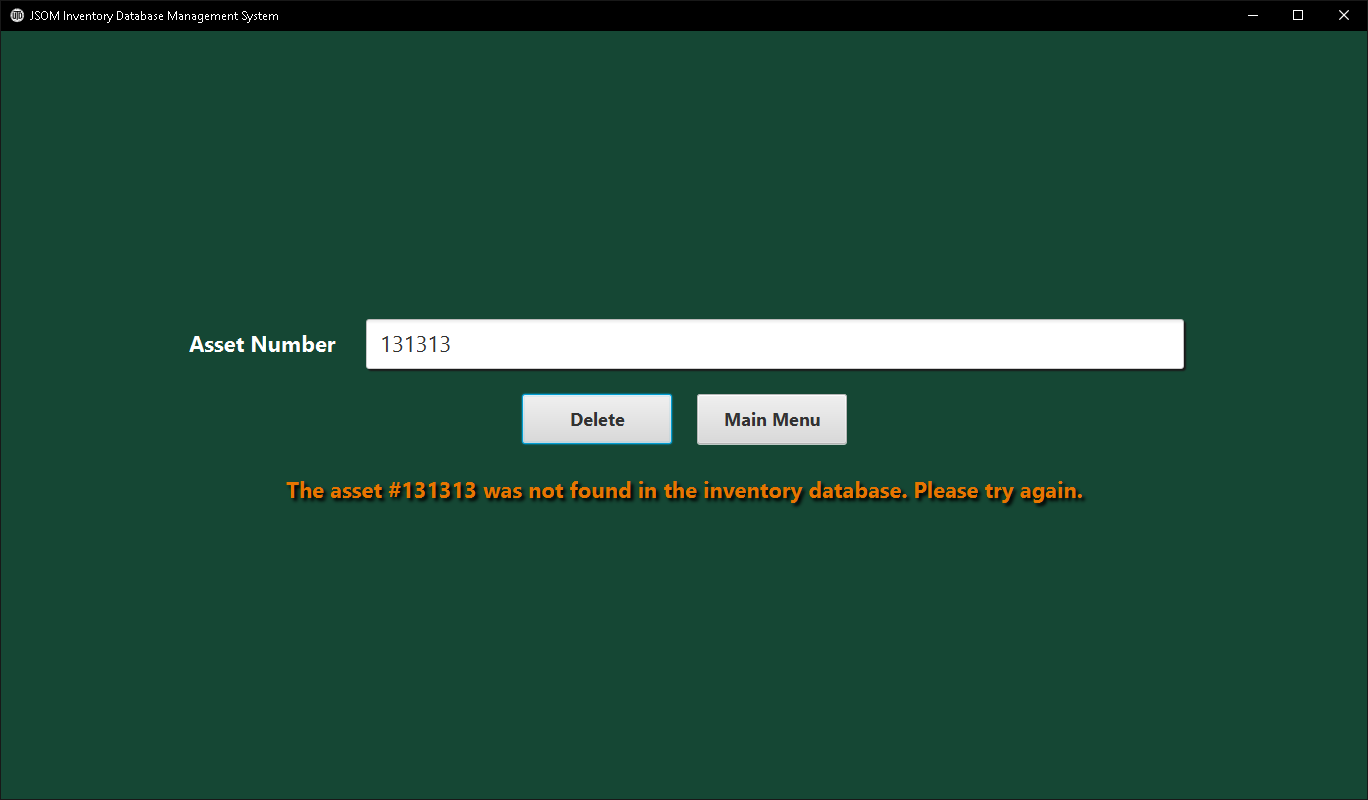
*Figure 11*: Results Edit Asset Confirmation Scene Displaying Asset Data Edit Input Validation – results\_edit.fxml

***User Cases – Test Case 4: Delete***

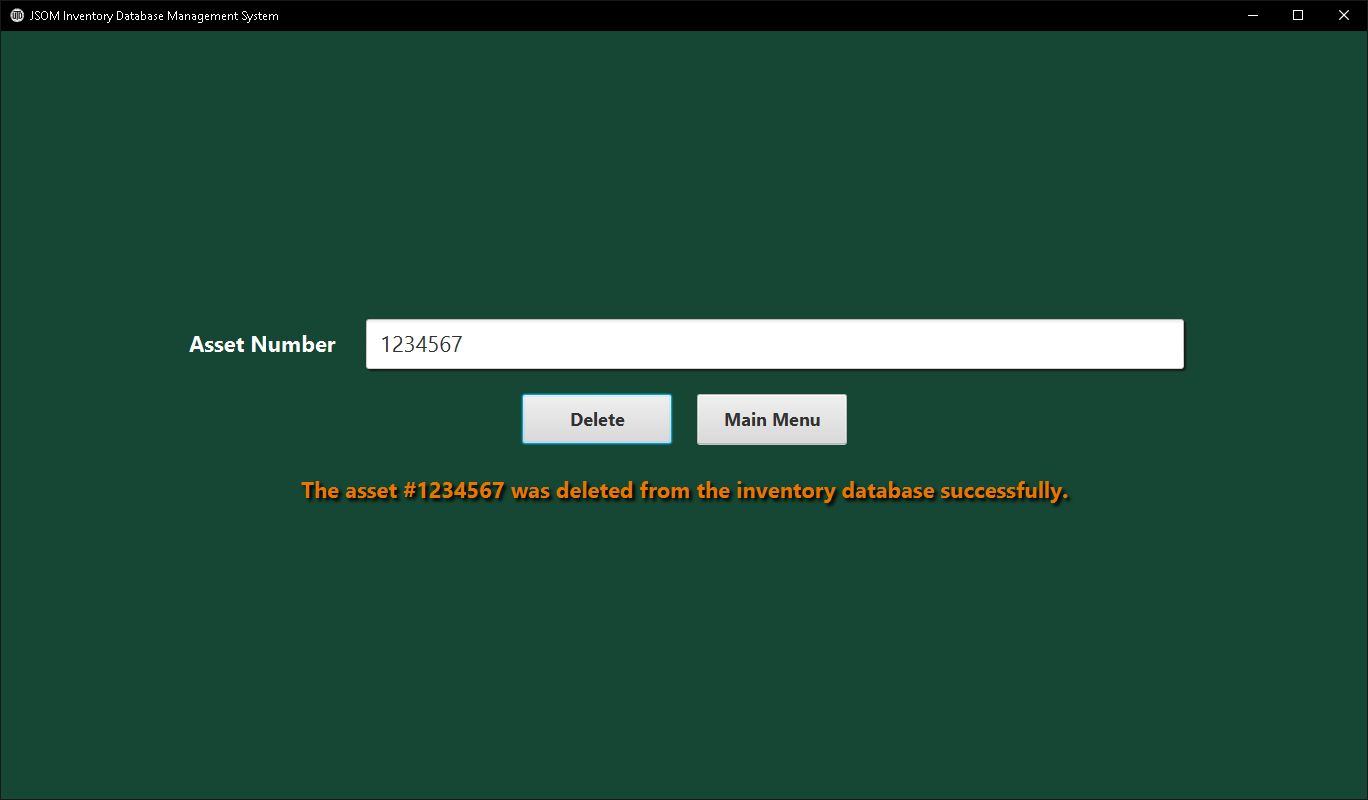
The user selects the Delete button from the main, or primary scene, page of the application. The delete scene is loaded and the user is presented with a text field to enter an asset number of an asset they would like to delete. If the user inputs an invalid asset number or the asset is not in the database, an error message is given. If the asset is successfully deleted from the database’s inventory table, a success message is displayed.



*Figure 12*: Delete Asset Scene Displaying Input Validation – delete.fxml



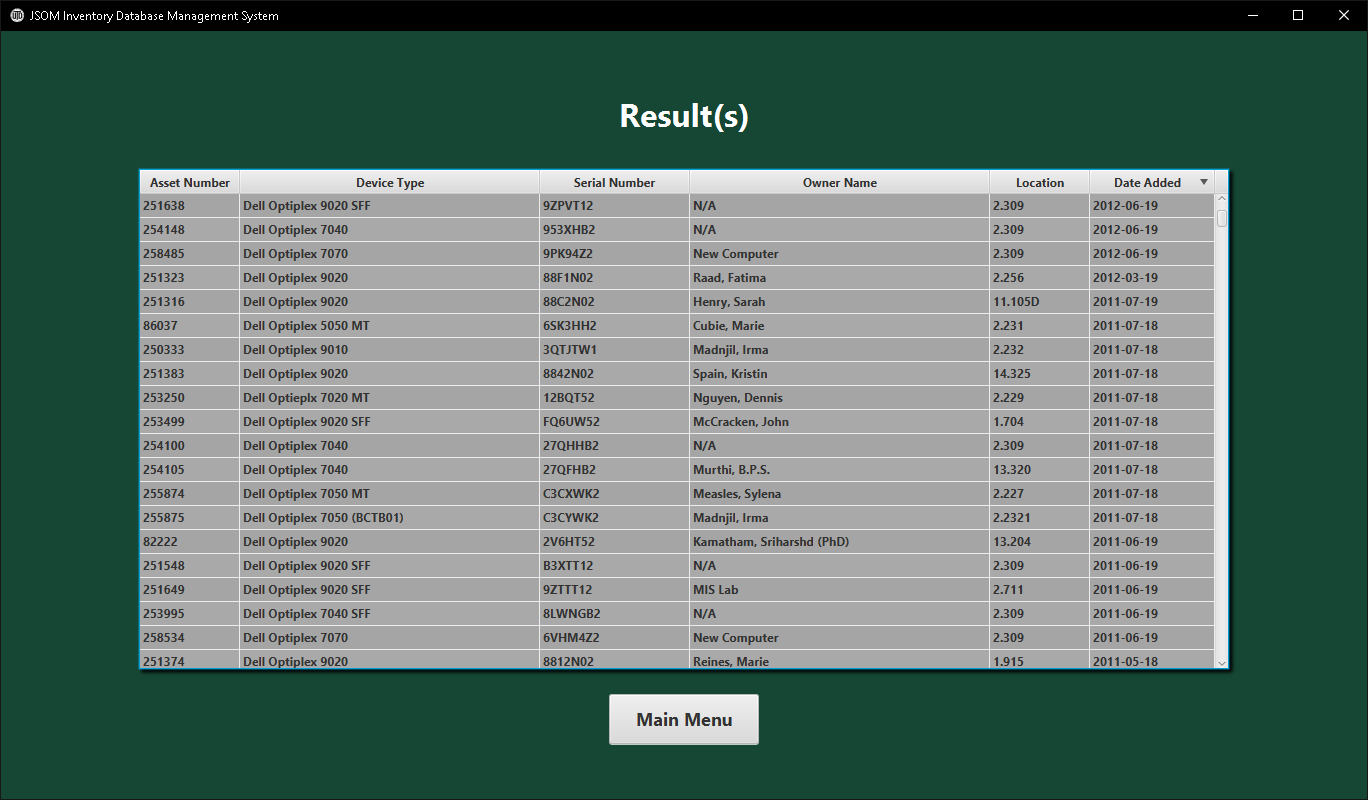
*Figure 13*: Delete Asset Scene Displaying Asset Not Found in Database – delete.fxml



*Figure 14*: Delete Asset Scene Displaying Successful Deletion of Asset – delete.fxml

***User Cases – Test Case 5: Display All***

The user first selects the Display All button on the main, or primary scene, of the application. The display\_all.fxml scene is then loaded. The application presents the user with a table that contains the data for all the assets currently in the inventory database table.



*Figure 15*: Display All Scene Displaying All Assets in Database – display\_all.fxml

***Inputs and Expected Outputs***

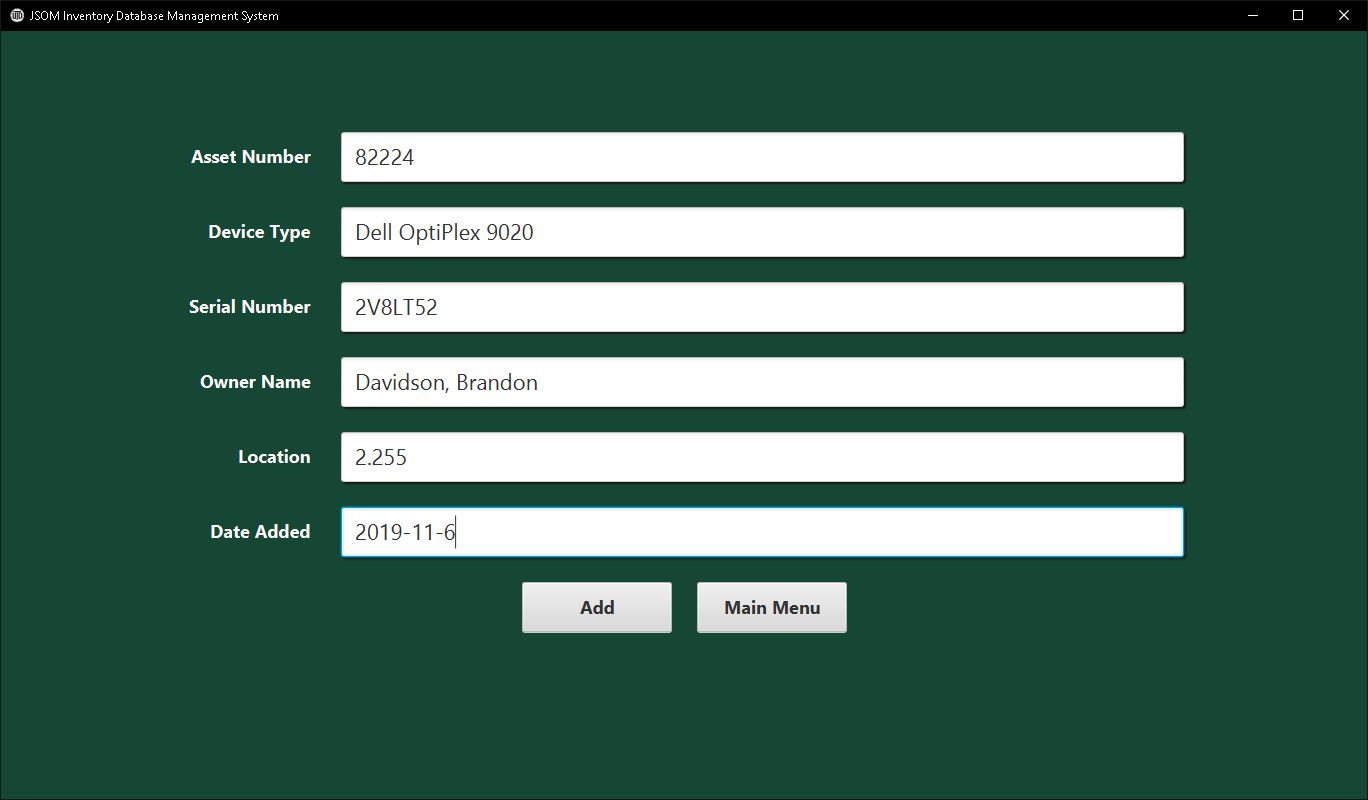
|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Input Number | Asset Number | Device Type | Serial Number | Owner Name | Location | Date Added |  |  | Button |  |
| Input 1 | 82224 | Dell OptiPlex 9020 | 2V8LT52 | Davidson, Brandon | 2.255 | 11/06/2019 |  |  | Add |  |
| Input 2 | 251258 | Dell OptiPlex 7010 | F93HJ02 | Lathen, Patricia | 14.218 | 12/02/2021 |  |  | Add |  |
| Input 3 | 251258 |  |  |  |  |  |  |  | Search |  |
| Input 4 | 82224 |  |  |  |  |  |  |  | Edit |  |
| Input 5 | 82224 |  |  |  |  |  |  |  | Delete |  |

*Table 1*: Set of Inputs to Inventory Database Software System

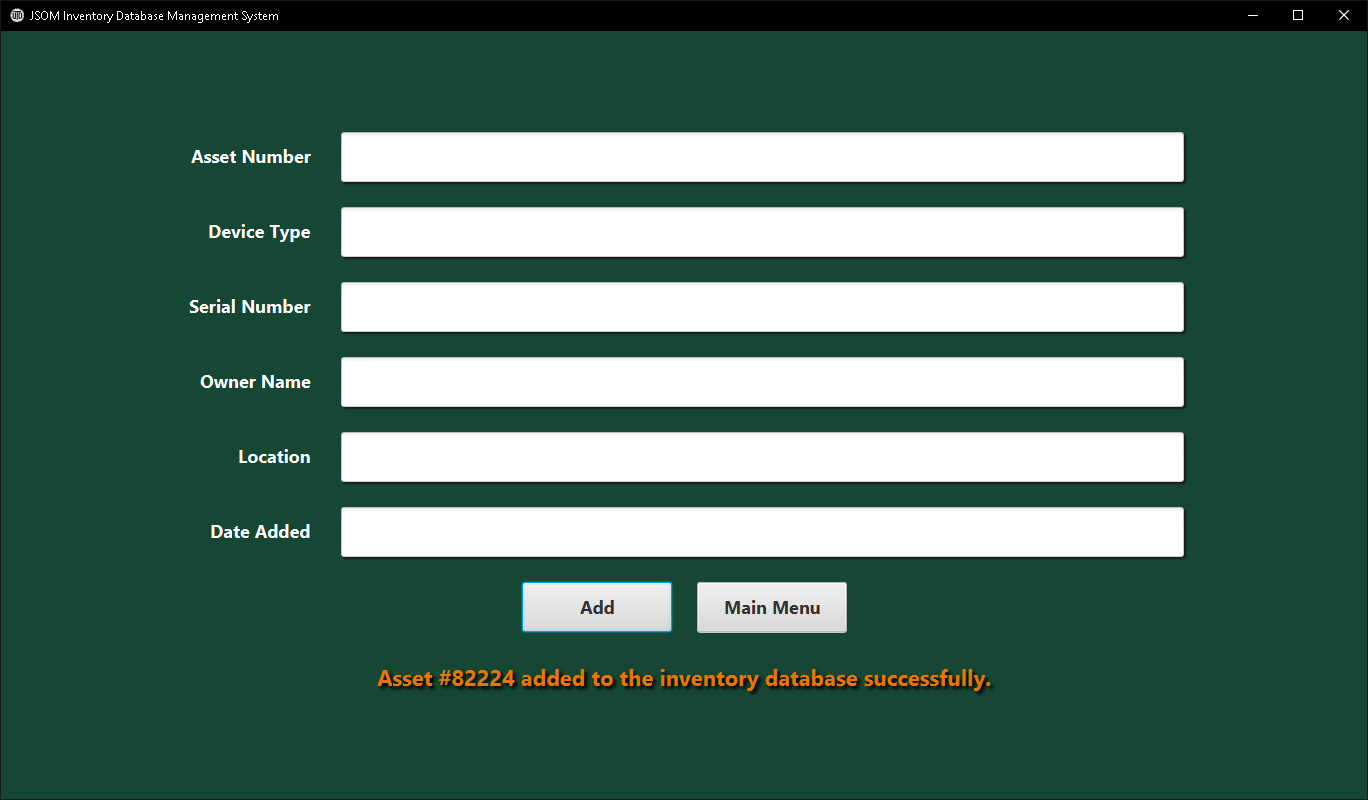
|  |  |
| --- | --- |
| Input Number | Expected Output |
| Input 1 | Confirmation message that asset has been added to database. |
| Input 2 | Confirmation message that asset has been added to database. Display of table showing asset and corresponding data pulled from database. |
| Input 3 | Display of table showing the data corresponding to the asset number 251258. |
| Input 4 | Display of table showing the data corresponding to the asset number 82224. Text fields available to user to input changes to make to the asset. |
| Input 5 | Confirmation of asset deletion from inventory database. |

*Table 2*: Set of Expected Outcomes for Inputs of *Table 1*

***Inputs and Expected Outputs – Input 1***

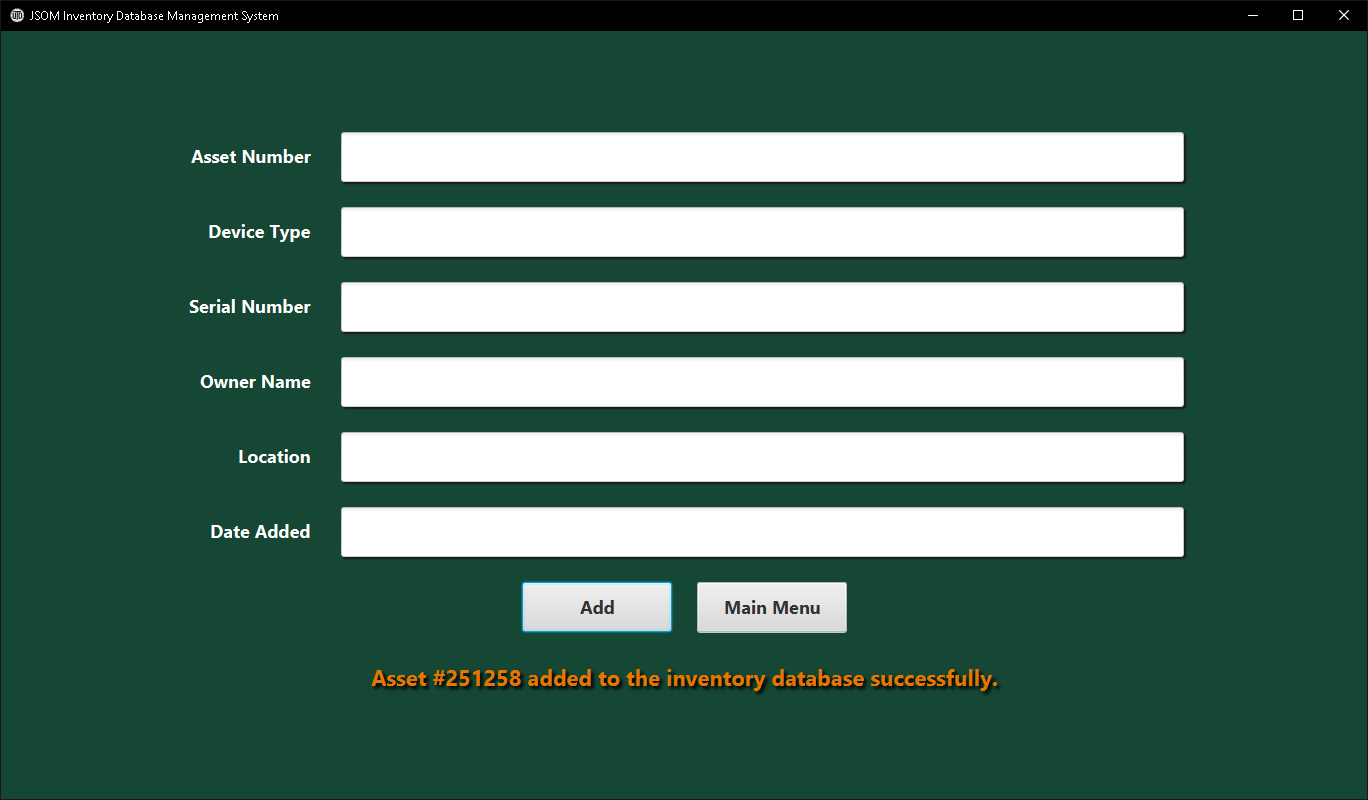


*Figure 16*: Add Scene View Before Add Button Pressed



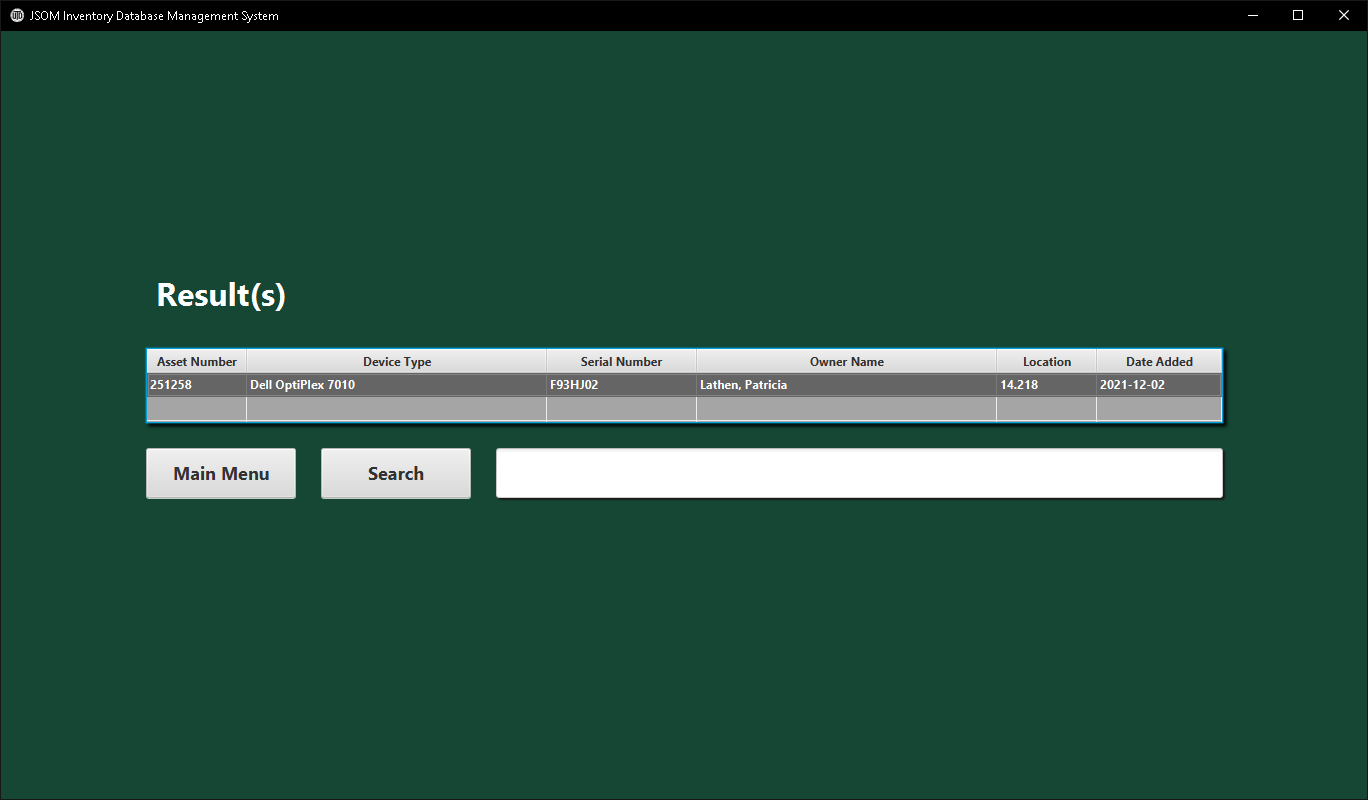
*Figure 17*: Expected Output Realized – Confirmation of Asset Addition to Database After Add Button Pressed

***Inputs and Expected Outputs – Input 2***



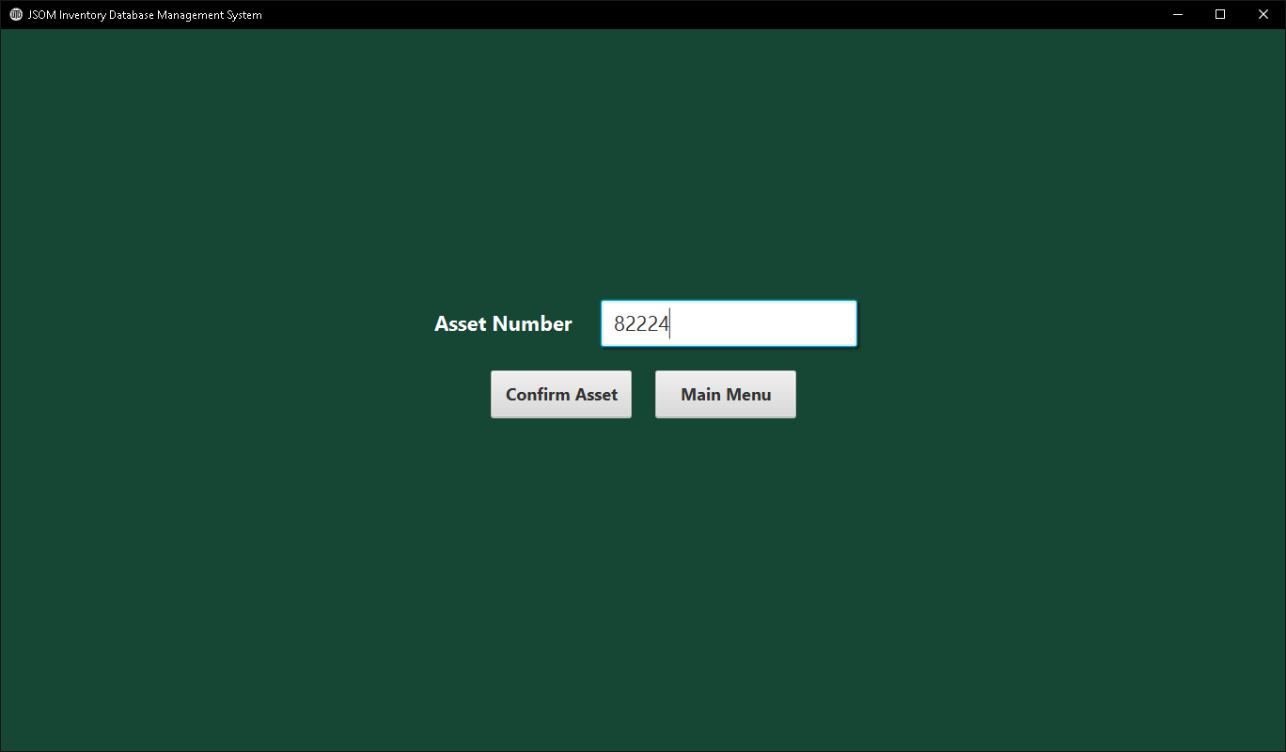
*Figure 18*: Expected Output Realized – Confirmation of Asset Addition to Database After Add Button Pressed

***Inputs and Expected Outputs – Input 3***



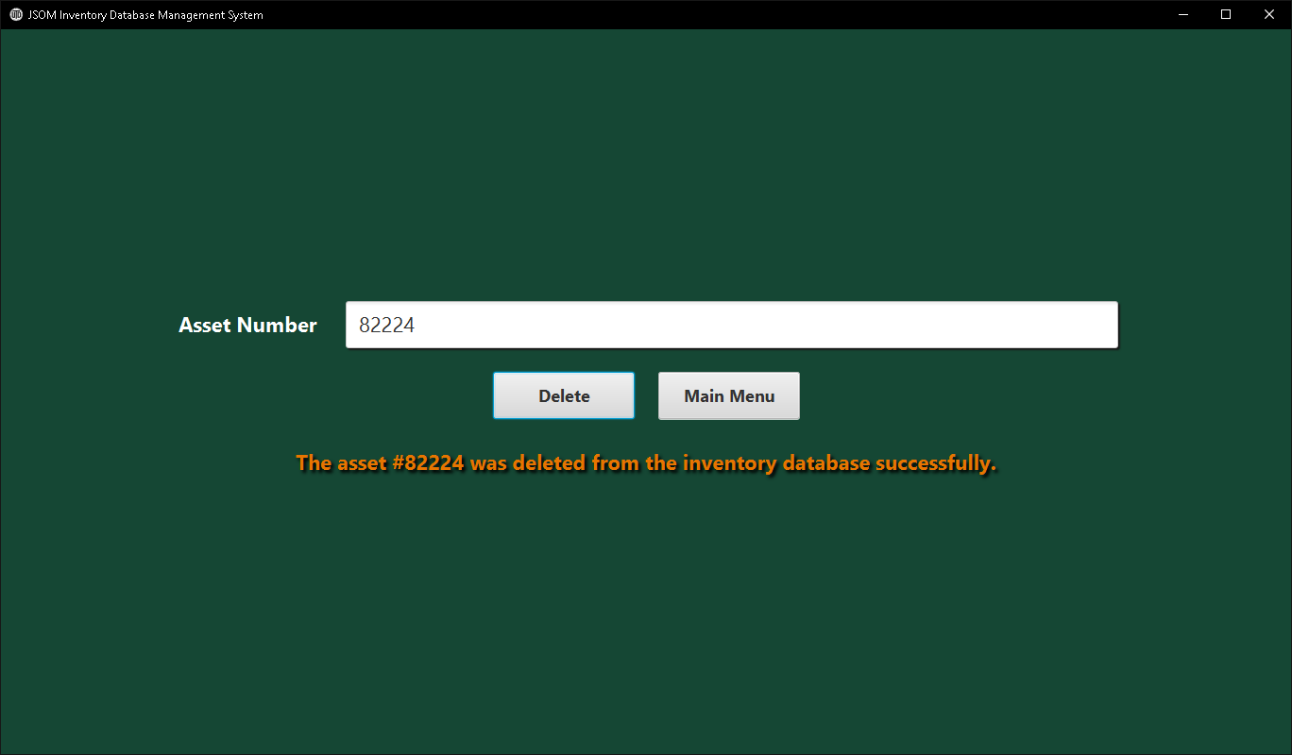
*Figure 18*: Expected Output Realized – Confirmation of Asset Displayed in Table After Search Button Pressed

***Inputs and Expected Outputs – Input 4***



*Figure 19*: Edit Asset Confirmation Scene – Before Button Press

***Inputs and Expected Outputs – Input 5***



*Figure 20:* Expected Output Realized – Confirmation of Asset Deletion Displayed to User

***Performance Expectation Testing***

As implemented the JavaFX GUI application runs without delay with the test data sizing in at 431 items within the inventory table of the database. As more rows are added to the database’s inventory table, the SQL queries execution times will inevitably increase. The action performed by the software system that will be most effected is the display all action as it retrieves data for all the assets within the database. Response times were so quick with the test data set of 431 that other tests were cancelled. The application is so responsive as is that scaling up to thousands of rows would not impact the performance enough to cause the application to fail or be sluggish.

**Final Capstone Project**

The final iteration of the capstone project delivers on the design specifications and solves the business need identified at the beginning of the course. The GUI developed follows the UTD asset standards set forth by the university and is presented in an easy to understand and use manner to the user. Essential basic database functionalities such as search, add, edit, delete, and display all assets within the database are provided to the user. The application was developed in a modular structure so that new features and functionalities can be easily added to make a more robust and useful application. GitHub was utilized for version control and was directly integrated into the workflow within Microsoft VS Code during development. As a version control system, it allows for the developer to track the iterative changes that are made to the code (Blischak et al., 2016).

**Concerns and Future Improvements**

There are several areas of concern and areas improvement that have been identified regarding the inventory database asset management system. The first being that it was designed to be a 1366 by 768 pixel sized window application. Future iterations should implement dynamically sizing UI components so that it doesn’t matter what screen the application is being used on, the UI makes sense. The MySQL database table should have more columns added to fully reflect the complexity of the data being stored on site within the IT department. Instead of only being able to search by asset number, the ability to search by any field should be provided to the user. As more than one device can be in a location, and one user can have many different devices for example. This type of information is often needed and useful for the technicians utilizing the system. A user database login and authentication page would be beneficial as well to add to the system. It could be integrated with the campus AD so that credential management is handled by that system. Furthermore, the inventory database can be hosted on a university server instead of a local host machine.

**Conclusion**

An inventory database asset management software system was designed and implemented to address the identified business need of the digitization and streamlining of asset management within the IT department of the School of Management at the University of Texas at Dallas. First, the topic approval process was described in detail. Next, a description of the project and the system to be designed was explored. UML diagrams of the system’s initial design followed along with the final design information. The system’s fault tolerance was tested during the next phase of the project. Both the testing process and results were provided. Lastly, the details of the final implementation of the software system developed were given. Through rigorous testing it was shown that the developed software system addressed the identified business need successfully.

**References**

Baresi, L., & Pezzè, M. (2006). An Introduction to Software Testing. *Electronic Notes in Theoretical Computer Science*, *148*(1), 89–111. https://doi.org/10.1016/j.entcs.2005.12.014

Blischak, J. D., Davenport, E. R., & Wilson, G. (2016). A Quick Introduction to Version Control with Git and GitHub. *PLoS Computational Biology*, *12*(1), 1–18. https://doi.org/10.1371/journal.pcbi.1004668

GitHub. (2022). *Creating and managing repositories*. GitHub Docs. https://docs.github.com/en/repositories/creating-and-managing-repositories

Huang, Y., & Kintala, C. (1993). Software implemented fault tolerance: Technologies and

experience. In *FTCS* (Vol. 23, pp. 2-9). IEEE Computer Society Press.

Keeling, M. (2017). Design It!. vbk://9781680503449

Kruk, G., Alves, O., Molinari, L., & Roux, E. (2017). Best practices for efficient development of JavaFX applications. In *Proceedings of the 16th International Conference on Accelerator and Large Experimental Control Systems* (pp. 1078-1083).

Murach, J. (2019). Murach’s MySQL (3rd ed.). vbk://9781943872466