**Sprint Report**

Contents

[1 Document Ownership 1](#_heading=h.1fob9te)

[2 Class Diagram 1](#_heading=h.2et92p0)

[3 Data Persistence 1](#_heading=h.3dy6vkm)

[4 Video Demo 2](#_heading=h.4d34og8)

# Document Ownership

This document is contained in your GitHub repository in a folder named *docs*.

| | Team | 4 | | --- | --- | | Team Member Names |  |
| --- | --- | --- | --- | --- |
|  | | 1. | Mashayla | | --- | --- | | 2. | Leslie | | | 3. | TJ | | --- | --- | | 4. | Joren | | 5. | Jasmine | |

Video Link(s) (See [Section 4](#_heading=h.1t3h5sf)):

| <https://www.youtube.com/playlist?list=PLIrtFZDPJfxdNHLcB420xkIw-oFpJI1Hw> |
| --- |

# Class Diagram

| **Deliverable**  At the conclusion of this sprint:   * Write a narrative explaining your overall design including how you implemented MVC. This should be accompanied by at least one accurately drawn class diagram. You can make multiple diagrams at different levels of granularity. * **The diagram(s) should be included in this document.** The diagram(s) must be readable. * The diagram(s) should also be saved as image files (jpg or png only) in your *docs* folder on GitHub. |
| --- |

In our project, we implemented the MVC (Model-View-Controller) design pattern to structure Bidsy (the name of our auction system) and logic in an effective and efficient manner. MVC is a popular way to organize code, as each section of code has a distinct purpose. Having this organization improves the maintainability, scalability, and testability, making it easier for future modifications. Here’s a breakdown for what we did, and how we did it:

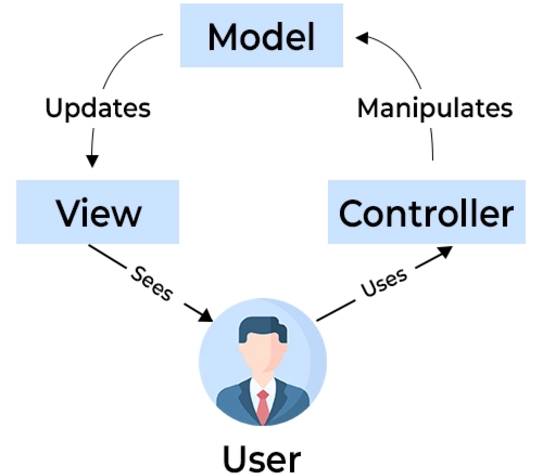
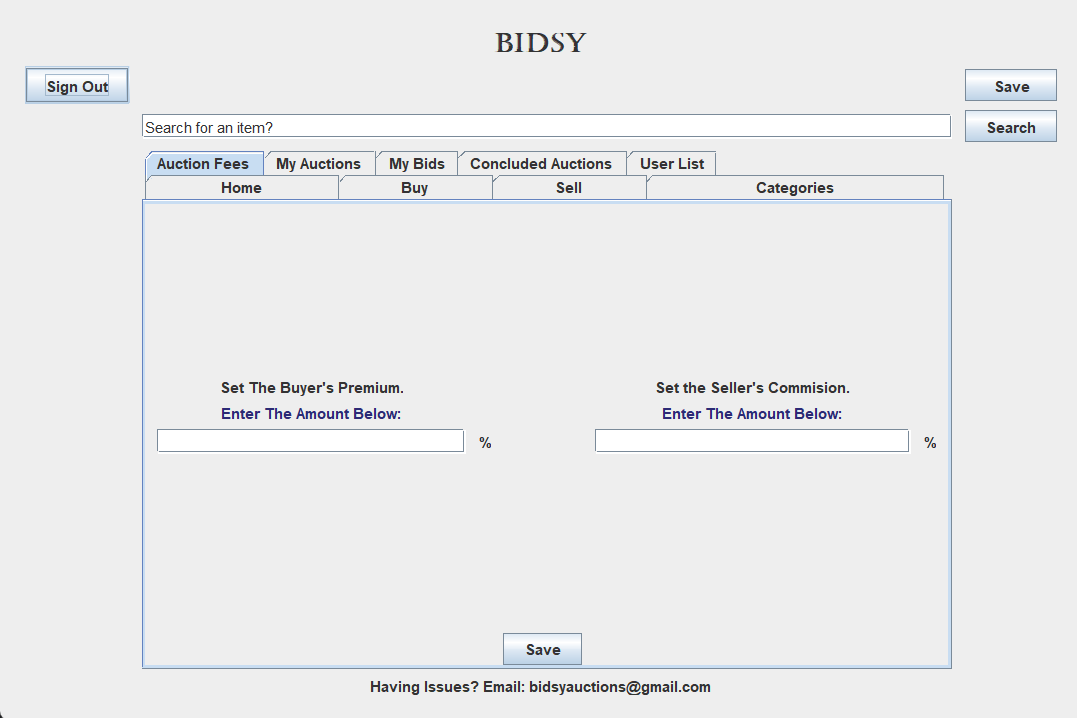
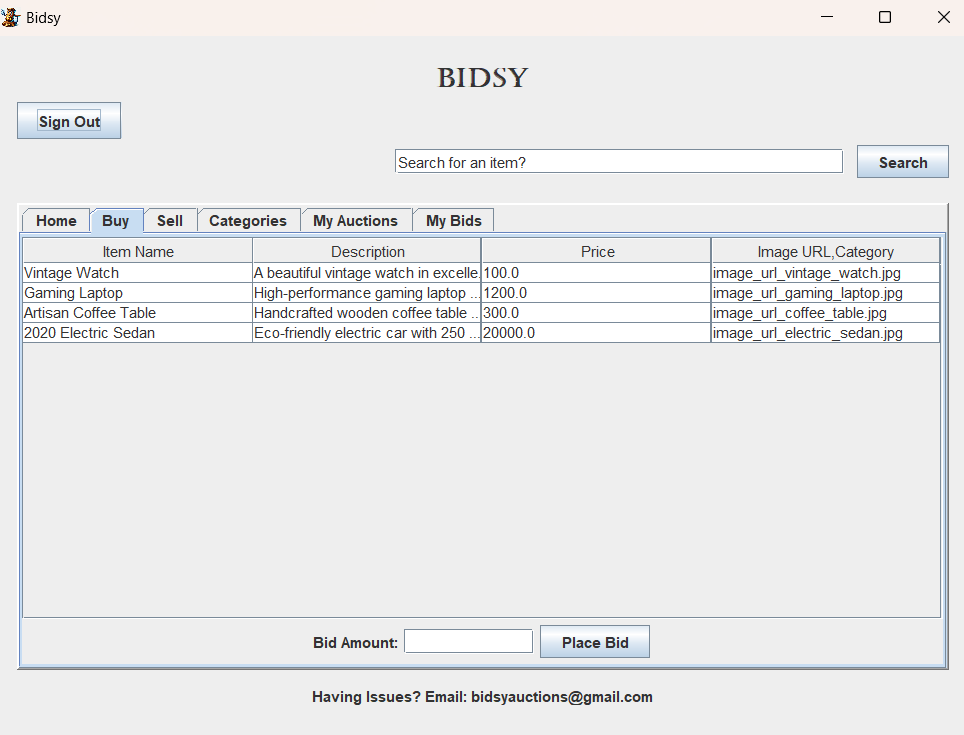
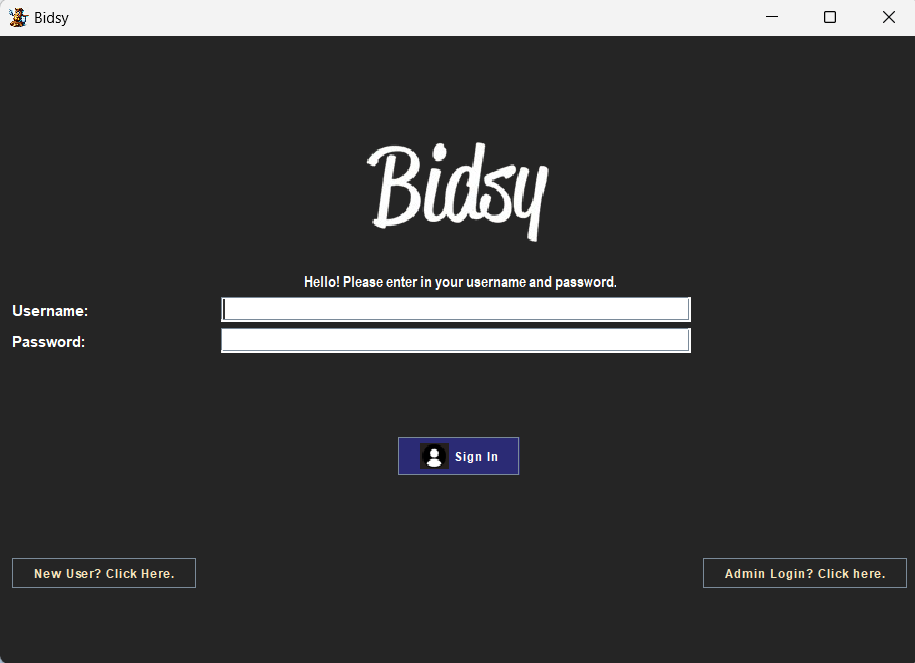
1. **Model**

The model is responsible for the core logic and data handling of Bisdy. It contains all of the classes and methods to manage auctions, bids, and user interactions. The model can be thought of as the “brain” of the system, as it focuses solely on internal logic, and ensures that everything is functioning and working properly without worrying about how the data is presented to the user. The latter is the View’s job. Some of the Model’s responsibilities in our auction system included Item Management, User Management, and so on. For instance, inside the Item Management class was the logic on how to place a bid, start a bid, buy an item now, and so on.

1. **View**

The view component is responsible for presenting the data to the user. It consists of GUI elements. More specifically buttons, tabs, text fields, tables, ect. The View listens to events and displays updates from the Model in a user-friendly way. In our design, there are several views such as the Login view (for logging in users), the User Home Page View (user homepage with a plethora of tabs for all the user functions), the System Admin View (System Admin Page with all the user functions and some addition functions specifically for admins), and so on and so forth. The view is notified whenever the model updates, and refreshes what is displayed accordingly. For example, whenever a new bid is placed, the auction information is updated, and the highest bid is displayed. The view doesn’t directly manipulate the model. Instead, it relies on the controller to handle interactions.

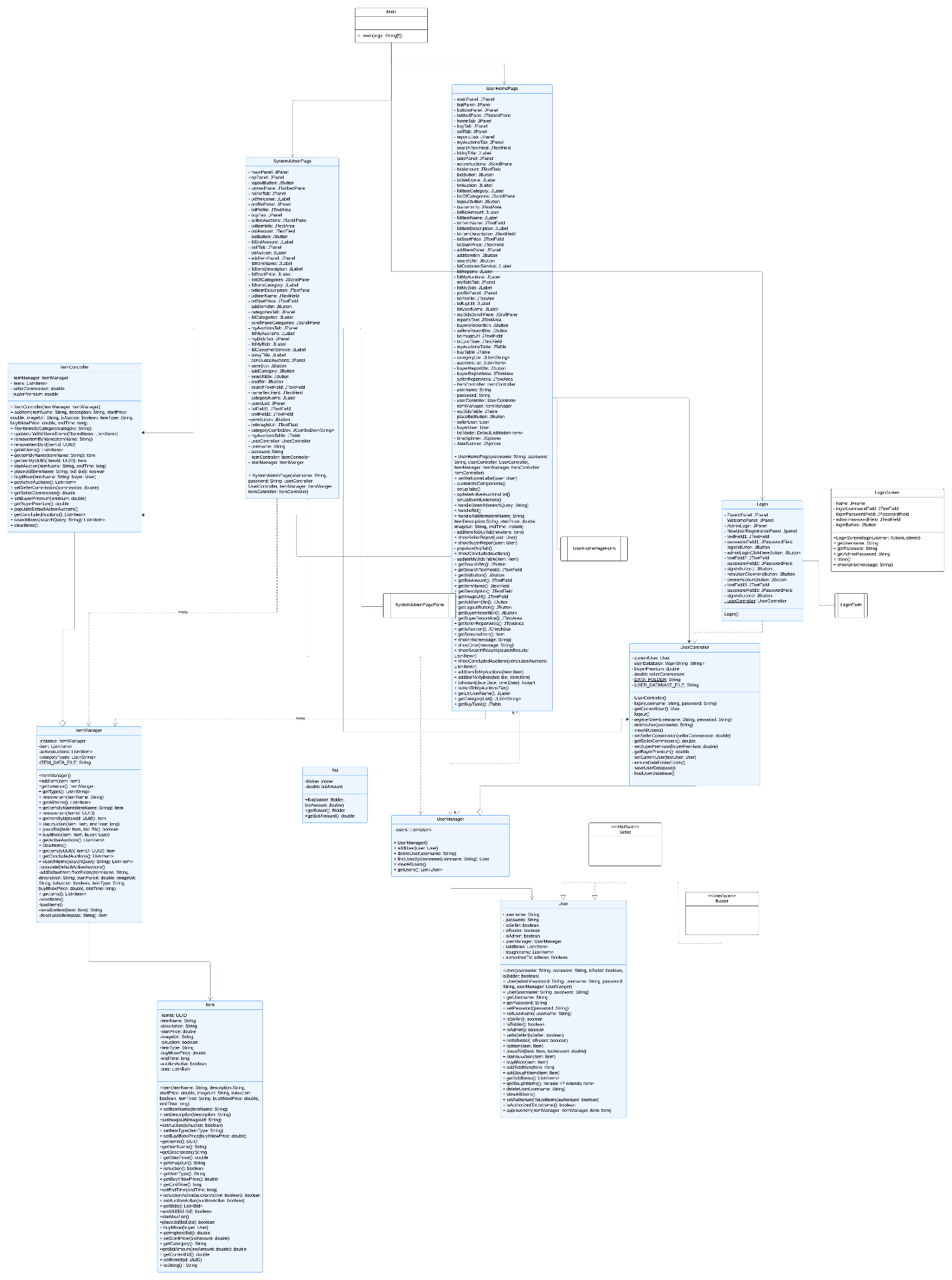
*Below are some of the different views we designed…*

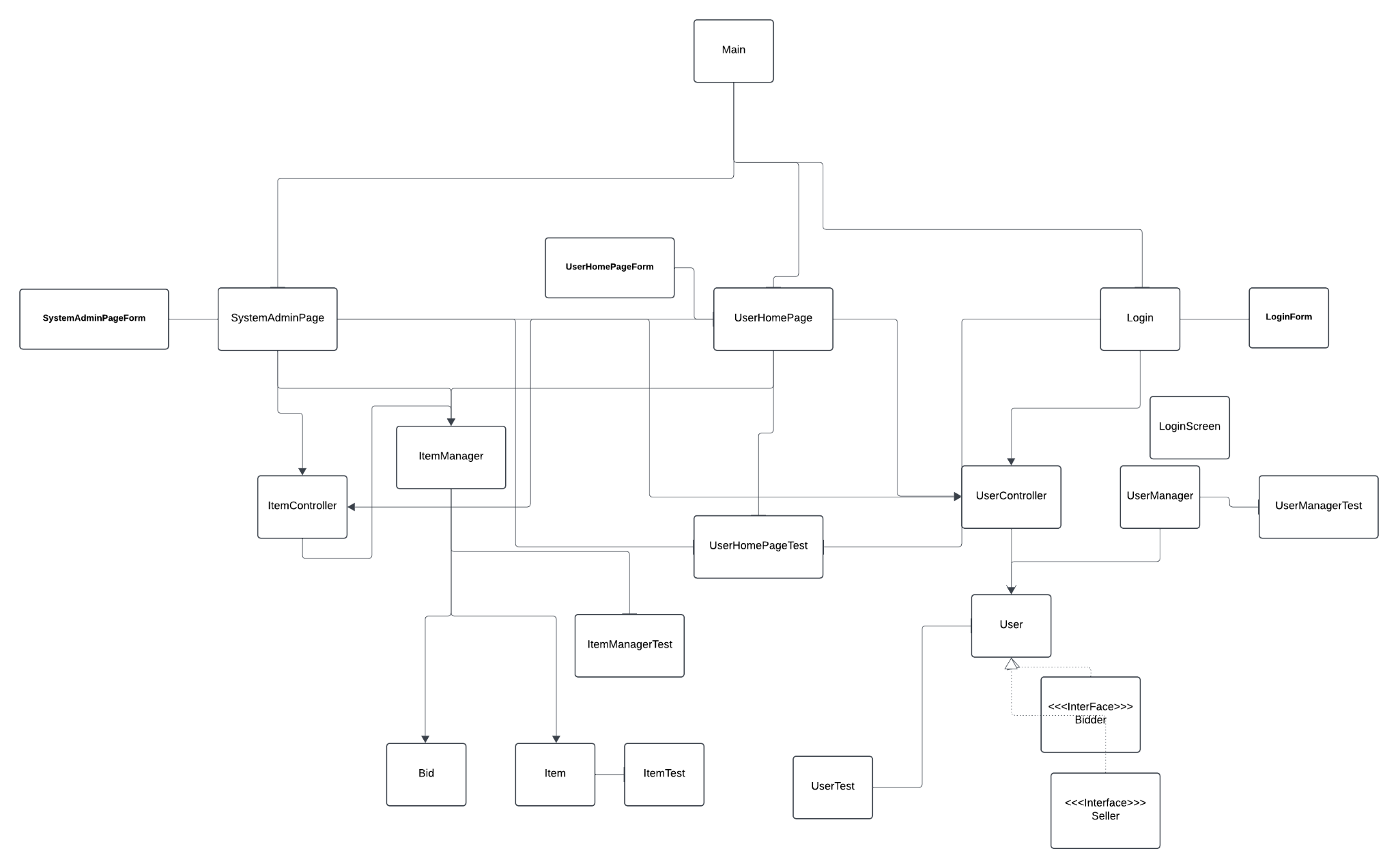


1. **Controller**

The controller serves as the “intermediary” between the model and the view. The controller is responsible for handling all of the interactions. It does this by taking the user input from the view and updating the model accordingly. It also listens to the changes in the Model and updates the View accordingly. There is this back and forth happening, and the Controller is in the middle of it all. Take the User Controller Class from our project for example. The User controller interacts back and forth between the View (in this example, this would be the user home page) and the Model (in this example, this would be the user class), taking the user's input from the view and updating the model accordingly.

Below are our class diagrams: one exhaustive version, and one condensed version.





# Data Persistence

| **Deliverable**  At the conclusion of this sprint, write a brief narrative explaining:   * How you went about data persistence, classes involved, *etc.* * The format of data file(s) used for data persistence, including a brief example of each file. |
| --- |

Our application implements data persistence by saving state information to text files, ensuring that all critical user and auction data are retained across sessions. User data is managed by the UserController class. Usernames and passwords of registered users are stored in a text file named “user\_data.txt”. Any time a user registers, it updates the data file, and anytime a user is deleted it also updates it. This file is loaded on application startup to populate the user database and ensure existing users can log in. Auction data, whether inactive or active, is managed by ItemManager class. All auctions are stored in a text file named “auctions\_data.txt”. Whenever changes occur, such as adding, removing, or updating an item, the text file is updated. On startup, this file is read to restore the state of all auctions. Brief examples of the code for each class is below:

**UserController:**

private static final String DATA\_FOLDER = "src/ebay/datafiles";

private static final String USER\_DATABASE\_FILE = DATA\_FOLDER + File.separator + "user\_data.txt";

public UserController() {

private void saveUserDatabase() {

ensureDataFolderExists();

try (BufferedWriter writer = new BufferedWriter(new FileWriter(USER\_DATABASE\_FILE))) {

for (*Map*.*Entry*<String, String> entry : userDatabase.entrySet()) {

writer.write(entry.getKey() + "," + entry.getValue());

writer.newLine();

System.out.println("Saving to: " + new File(USER\_DATABASE\_FILE).getAbsolutePath());

}

} catch (IOException e) {

System.err.println("Error saving user database: " + e.getMessage());

}

}

private void loadUserDatabase() {

File file = new File(USER\_DATABASE\_FILE);

if (file.exists()) {

try (BufferedReader reader = new BufferedReader(new FileReader(file))) {

String line;

while ((line = reader.readLine()) != null) {

String[] parts = line.split(",");

if (parts.length == 2) {

userDatabase.put(parts[0], parts[1]);

}

}

} catch (IOException e) {

System.err.println("Error loading user database: " + e.getMessage());

}

} else {

System.out.println("User database file not found.");

saveUserDatabase();

}

}

public void deleteUser(String username) {

if (currentUser != null && currentUser.isAdmin()) {

userDatabase.remove(username);

saveUserDatabase();

System.out.println("User deleted successfully.");

} else {

System.out.println("Current user is not authorized to delete users.");

}

}

public void registerUser(String username, String password) {

if (!userDatabase.containsKey(username)) {

userDatabase.put(username, password);

saveUserDatabase();

System.out.println("User registered successfully.");

} else {

System.out.println("Username already exists.");

}

}

**ItemManager:**

private static final String ITEM\_DATA\_FILE = "src/ebay/datafiles/auctions\_data.txt";

public ItemManager() {

this.items = new ArrayList<>();

this.activeAuctions = new ArrayList<>();

this.categoryTypes = new ArrayList<>();

loadItems();

}

private void saveItems() {

try (BufferedWriter writer = new BufferedWriter(new FileWriter(ITEM\_DATA\_FILE))) {

for (Item item : items) {

writer.write(serializeItem(item));

writer.newLine();

}

} catch (IOException e) {

System.err.println("Error saving items: " + e.getMessage());

}

}

private void loadItems() {

File file = new File(ITEM\_DATA\_FILE);

if (!file.exists()) {

System.out.println("No existing item data found. Starting fresh.");

return;

}

try (BufferedReader reader = new BufferedReader(new FileReader(file))) {

String line;

while ((line = reader.readLine()) != null) {

Item item = deserializeItem(line);

items.add(item);

if (item.isAuction() && item.isAuctionActive()) {

activeAuctions.add(item);

}

}

} catch (IOException e) {

System.err.println("Error loading items: " + e.getMessage());

}

}

public void removeItem(UUID itemId) {

items.removeIf(item -> item.getItemId().equals(itemId));

activeAuctions.removeIf(item -> item.getItemId().equals(itemId));

saveItems();

System.out.println("Item removed with ID: " + itemId);

}

public void addItem(Item item) {

items.add(item);

if (item.isAuction()) {

activeAuctions.add(item);

}

if (!categoryTypes.contains(item.getItemType())) {

categoryTypes.add(item.getItemType());

}

saveItems();

}

# Video Demo

| **Deliverable**  When your project is complete, create a video that demo’s your User Stories and provide the link in [Section 1](#_heading=h.1fob9te). Requirements:   * You can make a single video, or, if needed, 2 or 3. * The total length should be whatever is needed to accomplish the agenda below. * Preferably, post your video(s) on YouTube. * Agenda:  1. Explain your design. You should use class diagrams sized so that the relevant portions fill as much of the screen as possible. As part of this, explain how your design implements MVC. 2. Choose one User Story and step through the code as if it were being executed. You’ll start by showing the code where the appropriate event handler responds to the user. Next, to whatever it calls, etc, explaining as you go. 3. Explain how you handled data persistence showing code and data files. 4. Explain how you implemented the ability to start and stop time (User Stories 14 & 15) showing code. 5. For each completed User Story: 6. State the number of the User Story, and then state the User Story itself. 7. Demonstrate it with your software. If you have multiple system tests, you can demo all of them, or just the main (success) one. You can decide on the basis of time that you have. |
| --- |