Naiem Gafar
CSCI 370: Software Engineering
Dr. John Svadlenka
13 December 2020

Architecture and Design Document

## Introduction

The overall system is divided into five different subsystems, each of which houses similar functionality. Within each of these distinct components, the model–view–controller pattern is adopted to handle the interaction between the user interface, the business rules, a central database, and application logic that responds to user input. Utilizing Python's Django framework provides added functionality that not only reinforces the overall and micro architecture, but also simplifies future design choices. Whereas the larger architecture does not immediately align to any particular topology, it closely resembles a multi-tiered pattern, having a core layer above which other functionality is constructed. A brief description of each component is demonstrated in table 1 below.

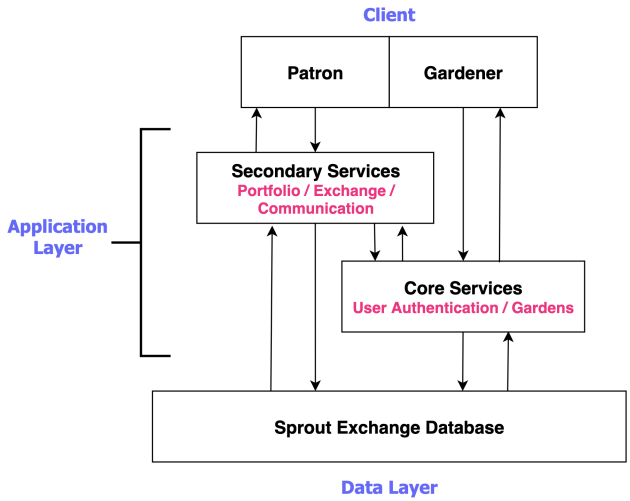
Table 1: An overview of the 5 subcomponents.

| # | Component | Description | Scope & Interaction |
|---|---|---|---|
| 1 | User Authentication | • Contains all functionality related to user authentication and security.<br>• Handles new user registration, login, logout, profiles, and ensures that users must be logged in to access certain functionality. | Core functionality that is used by all other components |
| 2 | Gardens | • All functionality related to the creation and management of all gardens and their associated tiers. | Core functionality used by portfolio, exchange, and communication |
| 3 | Portfolio | • Contains functionality related to the management of all of a patron's investments. | Secondary functionality that utilizes gardens component |
| 4 | Exchange | • Includes all of the buying and selling functionality for transactions of two types.<br>• Garden to Patron (in a primary market)<br>• Patron to Patron (in a secondary market) | Secondary functionality that utilizes the gardens and portfolio components |
| 5 | Communication | • All functionality related to the communication between all users of the system, including thorough updates and comments. | Secondary functionality that utilizes gardens component |

As is demonstrated below in Figure 2, there exists a data layer upon which the entire system rests. This layer houses a database responsible for the long-term storage of information that is served to all types of users. Immediately above this foundation resides the application layer, of which two classification of services are defined: core services, such as that for gardeners, and secondary services, such as that for patrons. The reason for this division of services is because the gardeners create gardens, upon which all transactions depend (req. 2, 3, and 4). Whereas the core services may directly access the

data layer, there are certain secondary functionalities that may require core services as well (req. 5, 6, and 7). At the very top resides two clients types, reflective of the two types of users of the system.

Figure 2: A multi-layered overall architecture.




The division of the system into different subcomponents, as well as the layering of functionality results in a degree of separation and independence that is highly desired. When similar functionality is grouped together, it allows for better organization not only of existing work, but also for any future features that may be considered. In terms of non-functional requirements, it allows for maintainability since the code may be easily changed without excessive rework of the system. Likewise, the overall security of the system is a top priority at all levels (req. 1, 8, and 9). As a result, in order to mitigate the threat of rouge users, hackers, and deliberate intrusion, there is an application layer complete with logic that will serve as a safety barrier around the database.

However, it must be acknowledged that such a degree of separation will have consequences for the performance of the system. With regard to time behavior and responsiveness, after the system exceeds some limit, there might be a slight degradation of service that stems from the fact that there is much communication between different subcomponents. However, maintainability and building a system that will be easy to understand in the future as new developers are incorporated has precedence over performance, especially at this primitive stage. It must also be acknowledged that certain functionality of the system relies on external systems. For instance, address validation requests services from a Google API, and payments are handled by Stripe API.

## Architecture and Design Philosophy

The decision to utilize the model–view–controller pattern stems from the requirement that the system must be a website that is developed using the Django framework. Django is the epitome of this pattern, a fact that is reinforced when examining the file structure and organization of projects in the framework. For instance, the views are all contained in each component's HTML template files, the models in the `models.py` file, and the controller is embedded within the logic of the `views.py` file. Utilizing such a framework eliminates the guesswork and mistakes that occur when working with a new architecture since from the inception of the project, a well defined architecture is established and maintained.

In the initial architecture phase, a monolithic pattern was considered, in which all of the system functionality would be incorporated into a single component. However, this idea did not translate into the final plan because it would neglect to provide separation. Given the scope of the project, all of the database models, business functionality, and views would be squeezed into only a few files. This would introduce dependencies that would complicate the process later on if new changes were to be introduced. Likewise, this choice would have resulted in immense stress on the system once user capacity passed a certain threshold. With the layered approach and the model view controller, especially as it pertains to using Django, user load is not a concern because the framework is optimized to handle large numbers of users. Whereas the current system is only projected to serve about 1000 customers in the first year of operation, Django has been proven to serve millions of customers as is demonstrated by services like Instagram and Pinterest.

## Architectural Views

The overall architecture is divided into subcomponents that are logically similar. For example, all of the functionality related to gardens are in the gardens app. Those concerning transactions are in the exchange app, and so forth. Structurally speaking, this translates into different tables that reside on a single database, known as the "Sprout Exchange" database. See figure 5 for a more detailed view.

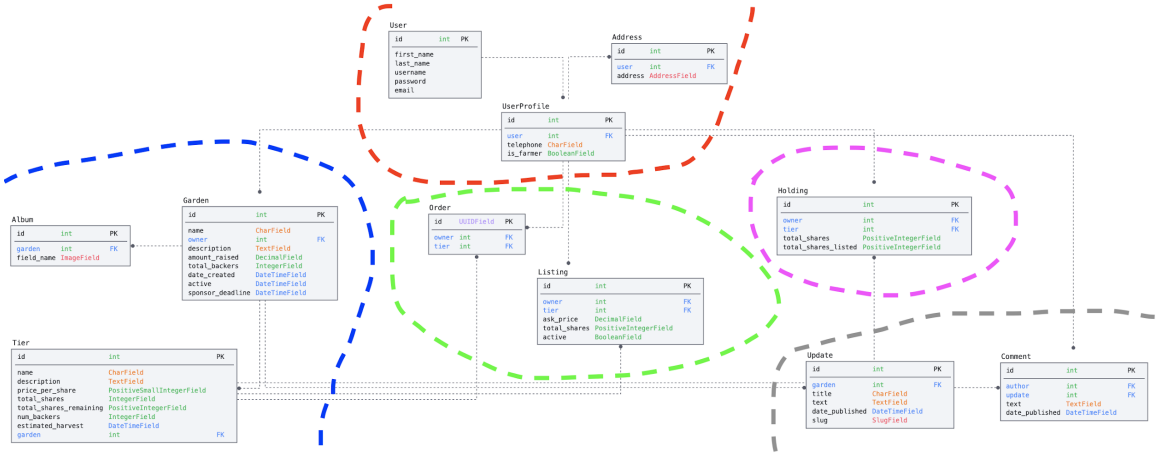
The business logic allows for the system to have interactive processes that work on these data members. For instance, when an individual purchases a share from a garden, data concerning the garden as well as the patron is modified. This is best represented by diagram 4 below:

In logistical terms, where it regards the development of the software, this software is being developed by a single developer. Thus, the focus is more on division of labor across time rather than across individuals. The focus should be to develop the components in the following order or priority, given that dependencies exist between them.

1. User Auth
2. Gardens
3. Portfolio
4. Exchange
5. Communication

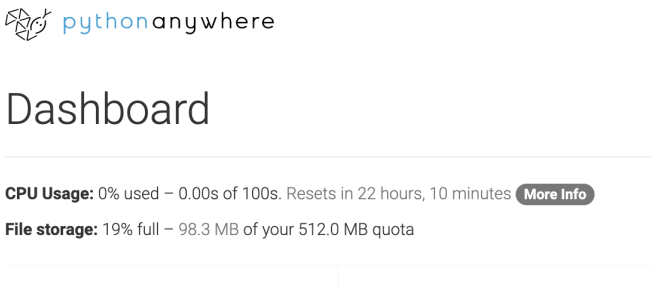
Figure 5 -

The different subsystems to be developed with colors corresponding to the above schedule.



Physically, the software will run on a remote server using the PythonAnywhere service. An advantage of using Django is that, at a very basic level, it takes care of what hardware is being allocated to what software processes. When it comes to developing this website, since it is not interacting with any hardware as an embedded system would, and is not engaging in CPU intensive processes, this is not a main concern at this moment in time. In terms of usage, the following screenshot from the PythonAnywhere dashboard provides insight:

Figure 6 - Resource Usage from PythonAnywhere




However, it should be noted that this is as much control as developers are allowed with regard to hardware. For future purposes, this information may be used for performance and enhancement of the system to minimize usage.

## Design Models

With regard to the context model, a majority of interactions occur between different components of the system. This is best illustrated below in figures 7a,b,c,d,e. The dotted lines represent the boundaries of each subsystem. Likewise, this also reflects on the structural model, especially with regard to the multiplicities among the different entities.

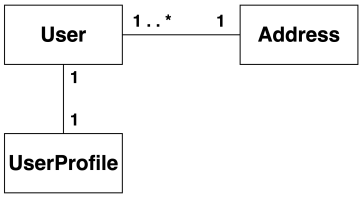



Fig. 7a - The userauth component does not depend on any other components.

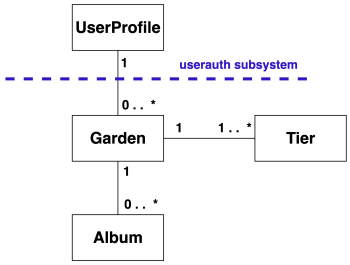



Fig. 7b - The gardens' component depends on the userauth component.

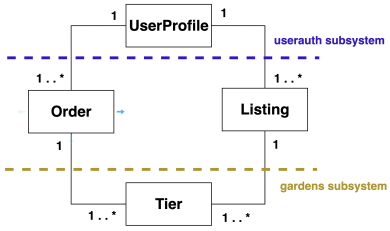



Fig. 7c - The exchange component depends on the userauth and gardens components.

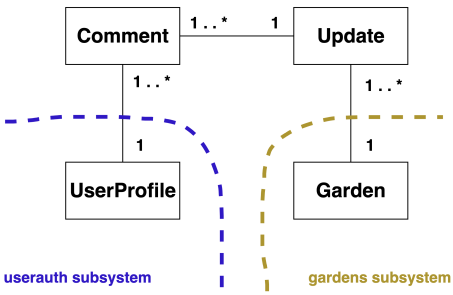



Fig. 7d - The communications component depends on the userauth and gardens components.

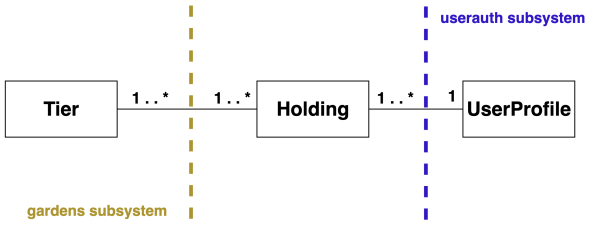


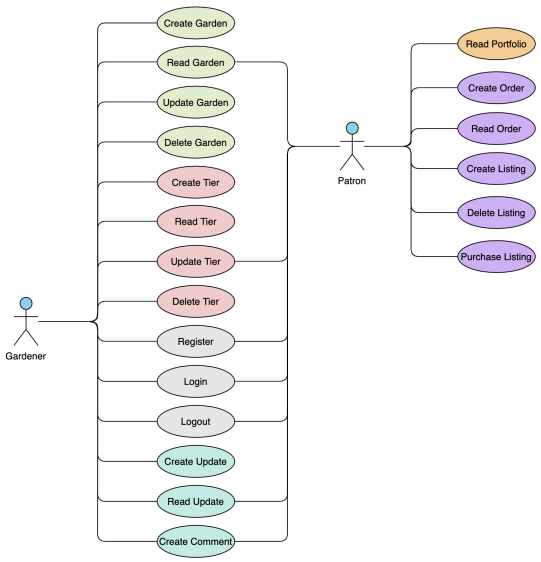

Fig. 7e - The portfolio component depends on the userauth and gardens components.

The Django framework complements this architecture because it allows for the creation of groupings called “Apps.” A Django App is represented by each of the above diagrams. The interaction between apps is as simple as including an import statement that follows the format:

```
from <App Name> import <File Name>
```

With regard to the interaction model, it must be noted that all of the functionality can be demonstrated below, along with the actors that they involve. It should be recognized that there is a lot of overlap in many of the features, and thus, additional emphasis should be paid during implementation to ensure that the correct privileges and access is being provided. The use case diagram below, shown in figure 8, demonstrates the interaction between two types of users: gardeners and patrons, and how they interact through the means of several entities such as gardens and updates.

Figure 8 - Overall Use Case for the system




| Subsystem | Functionality | Actor |
|---|---|---|
| UserAuth | Register<br>Login<br>Logout | All |
| Gardens | Create Garden<br>Read Garden<br>Update Garden<br>Delete garden | Mixed |
| | Create Tier<br>Read Tier<br>Update Tier<br>Delete Tier | Mixed |
| Communication | Create Update<br>Read Update<br>Create Comment | All |
| Exchange | Create Order<br>Read Order<br>Create Listing<br>Delete Listing<br>Purchase Listing | Patron |
| Portfolio | Read Portfolio | Patron |

In a similar manner, the all functionality requires that users are authenticated. While this contributes to the overall security of the system, it likewise confirms that a user is able to behave in a certain way. For instance, only gardeners can create a garden, only patrons can purchase shares in a garden, and only logged in users can comment on an update. The sequence of interactions that occur during authentication may best be shown figures 9a and 9b below.

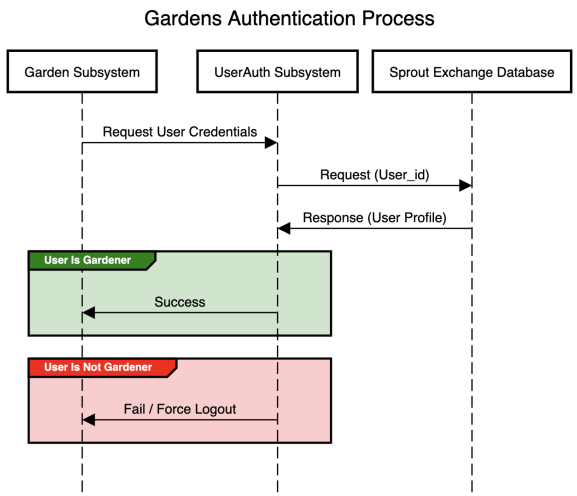



Figure 9a - This process ensures that a user is a **gardener** by checking the database.

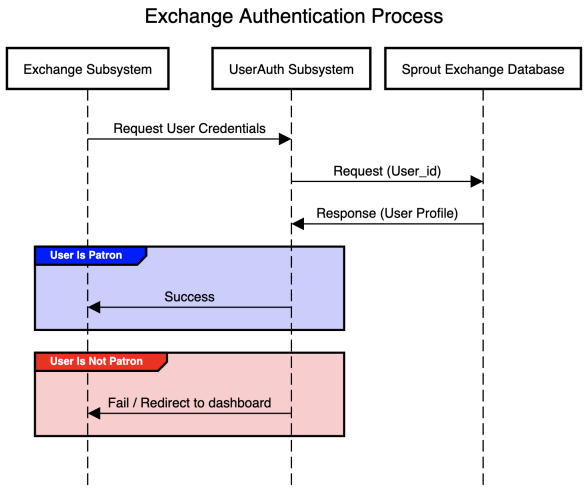



Figure 9b - This process ensures that a user is a **patron** by checking the database.

With respect to the structural perspective, it is important to note that the database design of the system is imperative to the design process because it provides insight into the functionality listed in the use case in figure 8. The UML class diagrams below in figure 10 lists all of the database entities and their associated fields. It should be noted that the relationship between many of them is through a foreign key, thus this allows for aggregation. For instance, a garden entity is composed of multiple tiers.

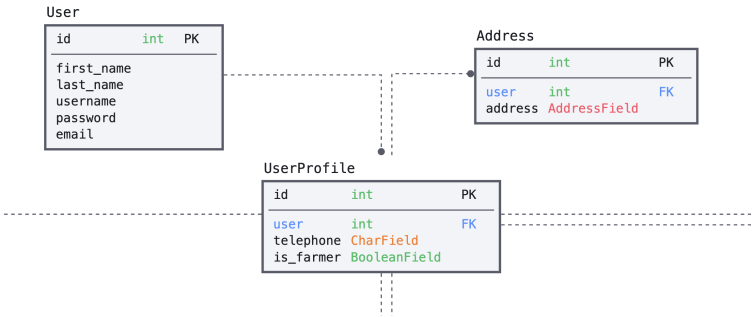



Figure 10a - User Auth Class Diagrams

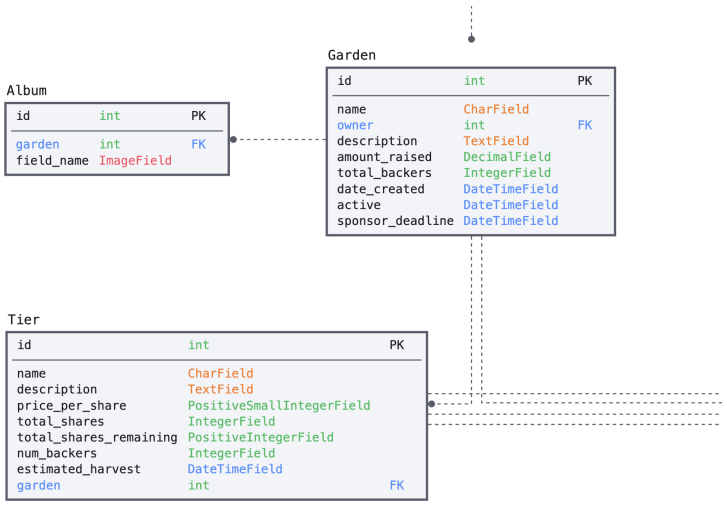



Figure 10b - Garden Class Diagrams

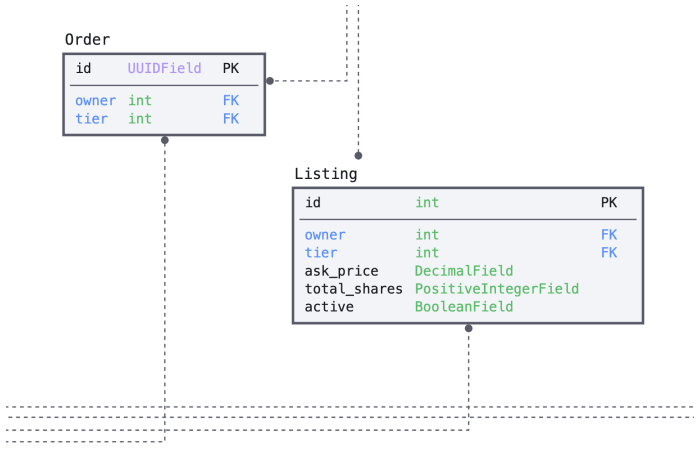



Figure 10c - Exchange Class Diagrams

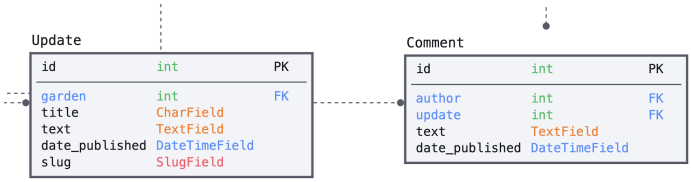



Figure 10d - Communications Class Diagrams

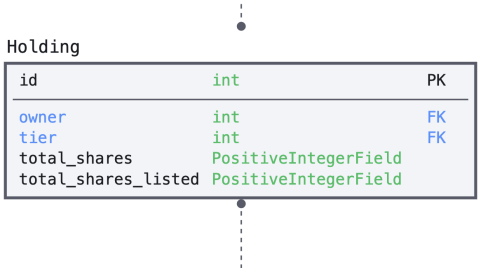
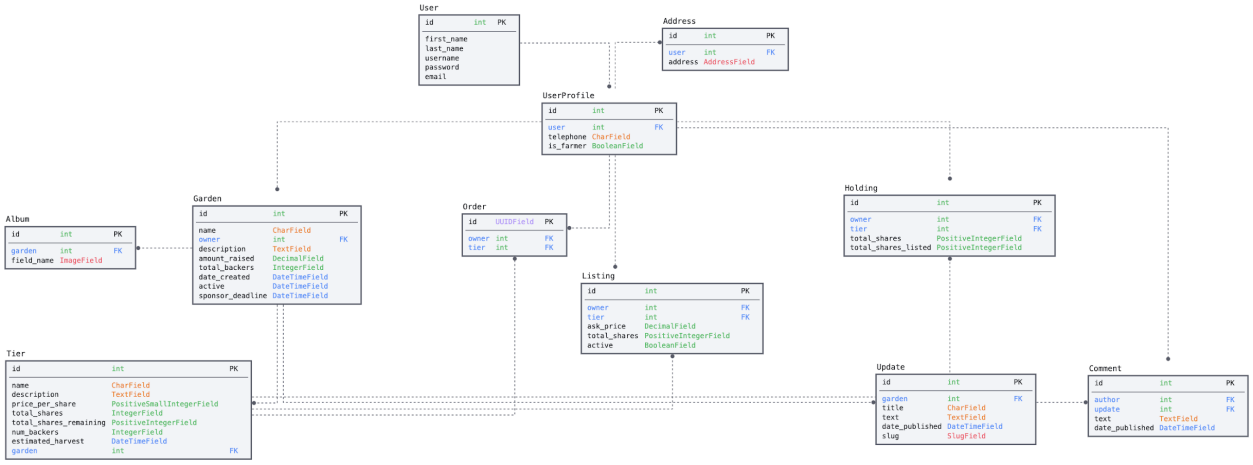



Figure 10a - Portfolio Class Diagrams

The overall database design is shown below in figure 11. Although it seems very elaborate, the architecture chosen will ensure the scalability and maintainability of the system.




With respect to the behavior of the system when presented with different inputs, there are several design choices that are employed. First and foremost, all inputs are validated and exceptions are provided as error messages to ensure that good quality information is being collected and stored. Similarly, the URL structure and the incorporation of CRUD operations for each of the data items ensures that the behavior is divided among different data.

Finally, the `settings.py` file in the main directory allows for settings to be set without having to incorporate them directly within the code. This allows for easy changes to information like API keys, file settings, and redirect links. Instead of having to conduct regression testing, when changes are made to this single file as opposed to directly in the code, this ensures that the functionality will work even as keys are altered.