**CS673 Software Engineering** 

**Team 5 - GearOnTheGo**

**Software Design Document**

| Team Member | Role(s) | Signature | Date |
| --- | --- | --- | --- |
| Lazaro Perez | QA Leader | *Lazaro Perez* | 9/7/2023 |
| Ahnaf Tajwar | Security Leader | Ahnaf Tajwar | 9/7/2023 |
| Shajee Ur Rehman | Configuration Leader | Shajee ur Rehman | 9/7/2023 |
| Jian Song | Architecture and Design | *Jian Song* | 9/7/2023 |
| Samantha Mathis | Team Leader | *Samantha Mathis* | 9/7/2023 |
| Saahil Vashishta | Requirement Leader | Saahil Vashishta | 9/7/2023 |

**Revision history**

| **Version** | **Author** | **Date** | **Change** |
| --- | --- | --- | --- |
| 0.1 | Saahil Vashishta | 9/23/2023 | 1. [First Draft - Planned Design Pattern](#_heading=h.1t3h5sf) 2. [First draft - Class Diagram](#_heading=h.1fob9te) |
| 0.2 | Ahnaf Tajwar | 9/25/2023 | 1. [Added current security logic](#_heading=h.tyjcwt) |
| 0.3 | Samantha Mathis | 9/25/2023 | [UI Design](#_heading=h.3znysh7) |
| 0.4 | Jian Song | 9/25/2023 | [Software Architecture](#_heading=h.30j0zll);  [Design Patterns](#_heading=h.1t3h5sf) |
| 0.5 | Lazaro Perez | 09/25/2023 | 1. Included Architecture of documents in Software Architecture section. 2. Included Database diagrams for each table. 3. Wrote the Introduction for the SDD document. |
| 0.6 | Shajee Ur Rehman | 09/25/2023 | 1. Slight Change to formatting 2. Added Business Logic/Algorithm section |
| 1.0 | Ahnaf Tajwar | 10/10/2023 | 1. Added Logout and routing to Security Design |
| 1.1 | Lazaro Perez | 10/10/2023 | 1. Updated Folder structure and Database Diagrams |
| 1.2 | Samantha Mathis | 10/10/2023 | 1. Updated UI design after implementation |
| 2.0 | Shajee Ur Rehman | 10/10/2023 | Final Version for Iteration 2 |

[Introduction](#_heading=h.gjdgxs)

[Software Architecture](#_heading=h.30j0zll)

[Class Diagram](#_heading=h.1fob9te)

[UI Design (if applicable)](#_heading=h.3znysh7)

[Database Design (if applicable)](#_heading=h.2et92p0)

[Security Design](#_heading=h.tyjcwt)

[Business Logic and/or Key Algorithms](#_heading=h.3dy6vkm)

[Design Patterns](#_heading=h.1t3h5sf)

[Any Additional Topics you would like to include.](#_heading=h.4d34og8)

[References](#_heading=h.2s8eyo1)

[Glossary](#_heading=h.17dp8vu)

# Introduction

In this document you will discover that we have outlined our software architecture which includes our backend components, frontend components, dependencies and interactions, as well as a folder architecture. We have outlined the class diagrams with entities consisting of User, Renting User, and Hosting User, and Profile, Equipment, Reservation, and PaymentInfo. We also included our UI Designs which are a blueprint of how our web application will look and our intent. Our database and design is still in a working progress but for now what we have constructed are the following tables Equipment, User, PaymentInfo, Reservation. For our security design we are demonstrating the ways in which we are using Hashing Passwords, User Validation, Password Validation, and Forgot Password. For our Design patterns we explain how we are using Abstract Factories to create Users that are RentingUsers and HostingUser. Observer design to notify users of reservations, and the composite design for our UI.

# Software Architecture

In this section, you will describe the decomposition of your software system, which includes each component (which may be in terms of package or folder) and the relationship between components. You shall have at least one diagram to show the whole architecture of . The interface of each component and dependency between components should also be described. If any framework is used, it shall be defined here too.

The team will implement the code using Python Flask in the backend and react.js in the frontend. Details are shown below:

**Backend Components (Python Flask):**

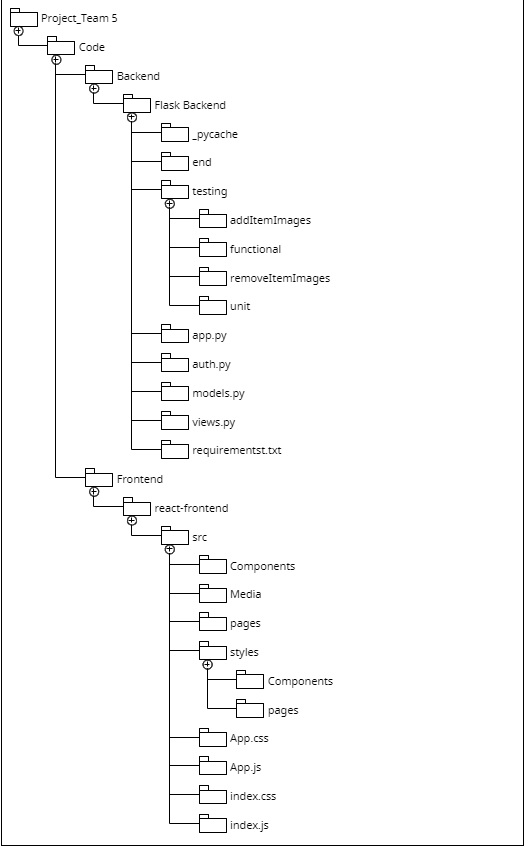
* Flask Application: This is the core component of our backend. It's responsible for handling HTTP requests and responses.
* API Routes: Flask defines API routes that handle specific HTTP requests such as GET and POST. These routes correspond to different functionalities in our application. For example:
  + /api/getReservation - To retrieve an existing reservation..
  + /api/makeReservation - To create a new reservation.
* Database Models: we defined models for our data objects via a relational database ElephantSQL using PostgreSQL. These models define the structure of our data and how it interacts with the database.
* Controllers (app.py): These components handle the business logic of our application. They receive requests from the API routes, interact with the database models, and prepare data to send as responses.

**Frontend Components (React.js):**

* React Application (app.js): This is the root component of our frontend application. It manages the overall structure and routing of our app.
* UI Components: React allows us to create reusable UI components that display data and interact with the user. Each UI component represents a part of our application's user interface.
* API calls: In our React app we make HTTP requests to the Flask API endpoints. This module abstracts the communication with the backend.

**Dependencies and Interactions:**

* The React application interacts with the Flask backend by making HTTP requests to the API endpoints (e.g., GET for fetching tasks, POST for creating tasks).
* UI components in React receive data from the backend via API calls and display it to the user.
* When a user interacts with the frontend UI (e.g., creating a new reservation or checking an existing reservation), the React components send corresponding API requests to the Flask backend to perform those actions.
* The Flask backend processes API requests, interacts with the database models to perform CRUD operations, and sends responses back to the React frontend.

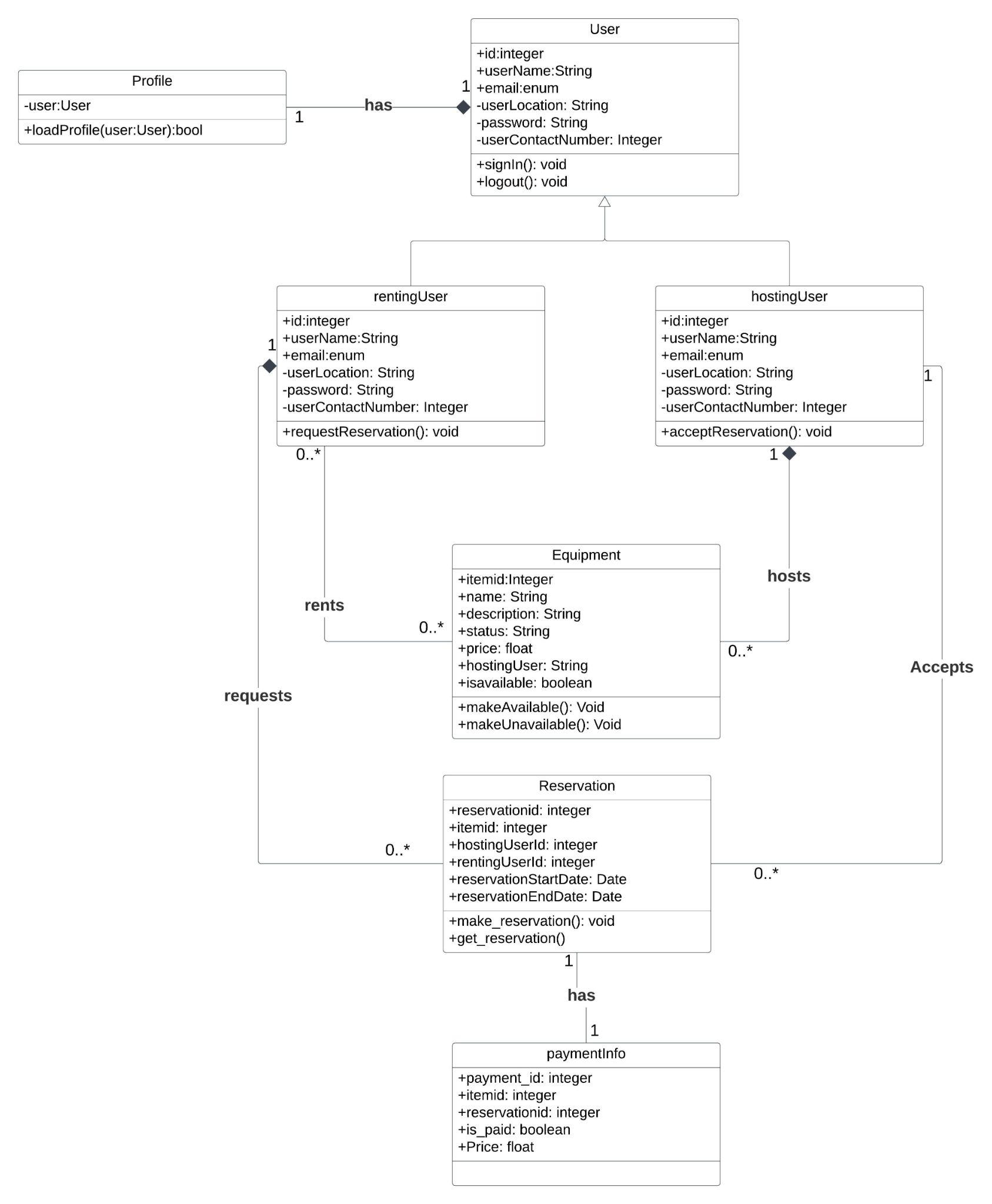


# Class Diagram

In this section, you will provide a detailed description of each component (or package) and use one or multiple class diagrams to show the main classes and their relationships in each component.

In the image below (Exported from Lucid chart), the first draft of the class diagram for Gear To Go can be seen. Here we have identified 7 entities so far:

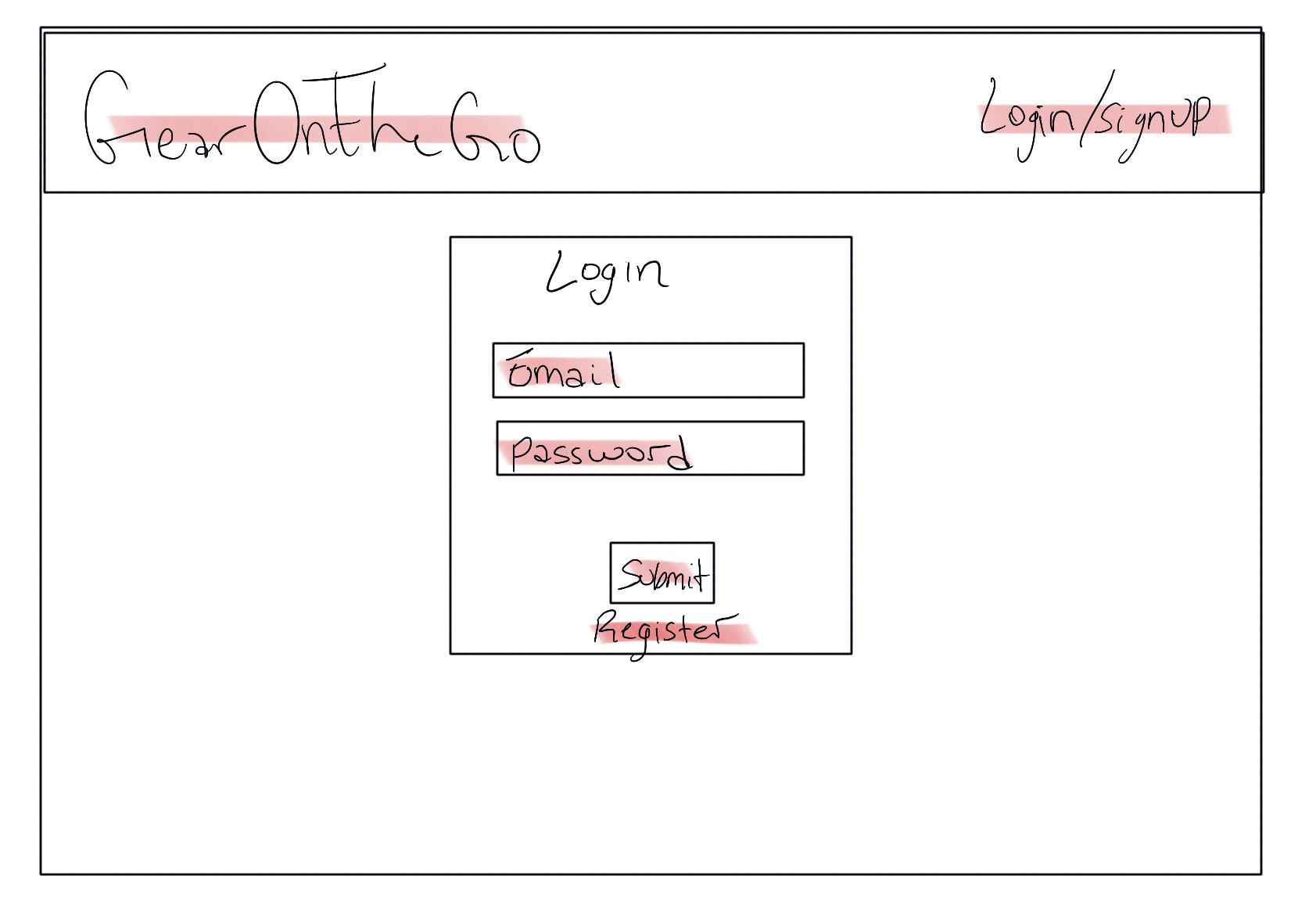
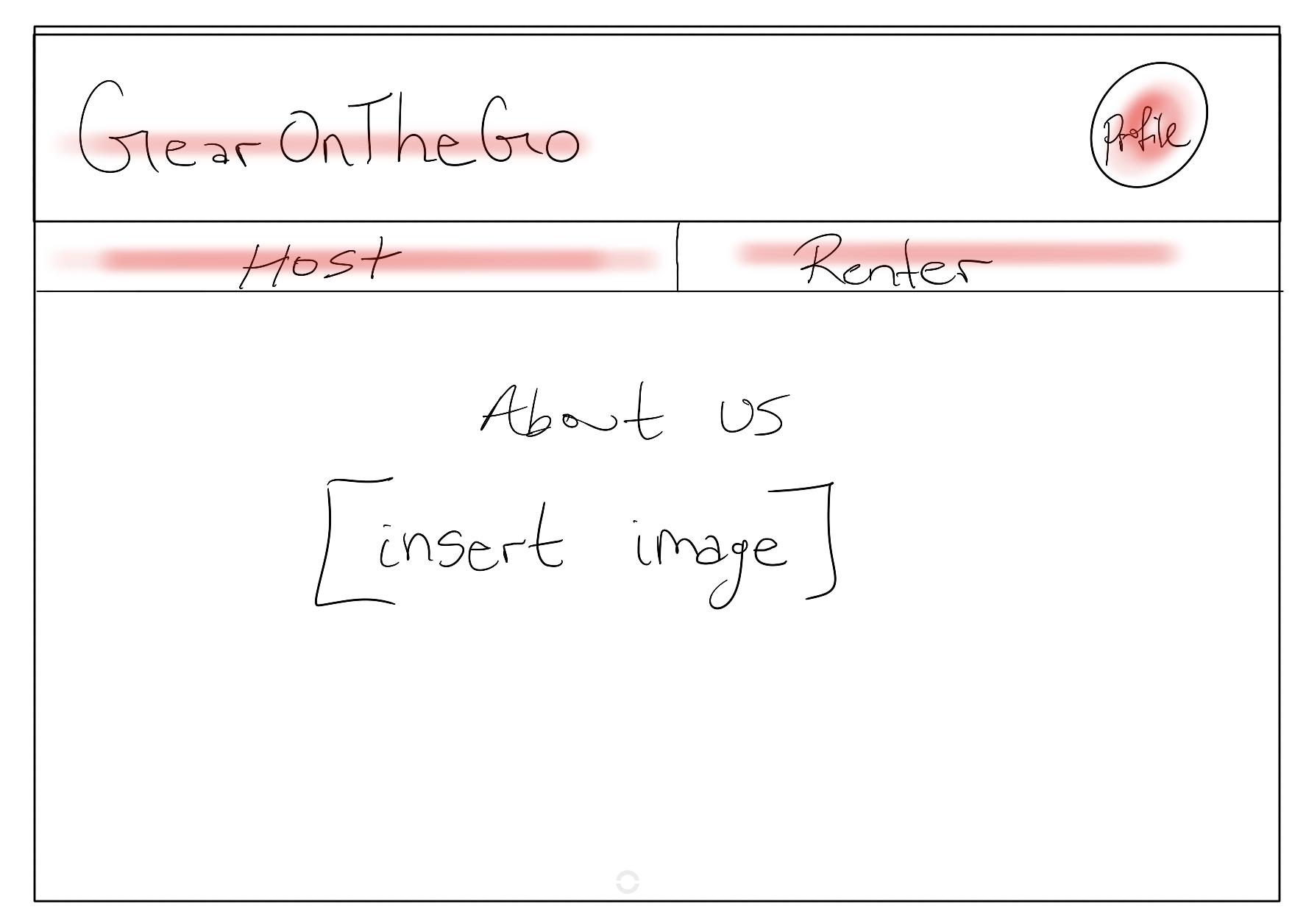
* **User:** User is the parent class for both renting user and Hosting user. It has variables that will access user information to add them to the application and facilitate their sessions. So far , id, username, email, user location, password and contact number have been added. The variable information can be viewed in the diagram below.
* **rentingUser:** The Renting User class (child class of users) is created for users that will rent out equipment from hosting users. Their unique method so far is request reservation and they share the same variables as users (parent class).
* **hostingUser:** The Hosting User class (child class of users) is created for users that will rent out equipment to renting users. Their unique method so far is accept reservation and they share the same variables as users (parent class).
* **Profile:** Profile entity has user information and potentially (time permitting), a user photo, bio and other information on their past rentals or current equipment, along with reviews. Each profile belongs to 1 user and 1 user can only have 1 profile. Profile can’t exist without the user thus the composite relationship can be seen.
* **Equipment:** Equipment class has the item id, name, description, status, price and availability and a hosting user. Each equipment instance is hosted by a hosting user and may or may not be rented by a renting user. Since the equipment can’t exist without the person hosting it, we can see the relationship in the diagram below.
* **Reservation:** The reservation class tracks each reservation made, which can be made by the renting user (so it can’t exist without the renting user) and accepted by the hosting user. It includes an id, item id for reservation, jousting user id, renting user id, reservation start date and end date.Each renting user can request (make) 0 or many reservations and hosting users can accept 0 or many reservations.
* **Paymentinfo:** Lastly the payment entity deals with payments for each item and provides status on whether the reservation is paid for or not. Each reservation has only 1 instance of payment info and each payment info instance belongs to 1 reservation.

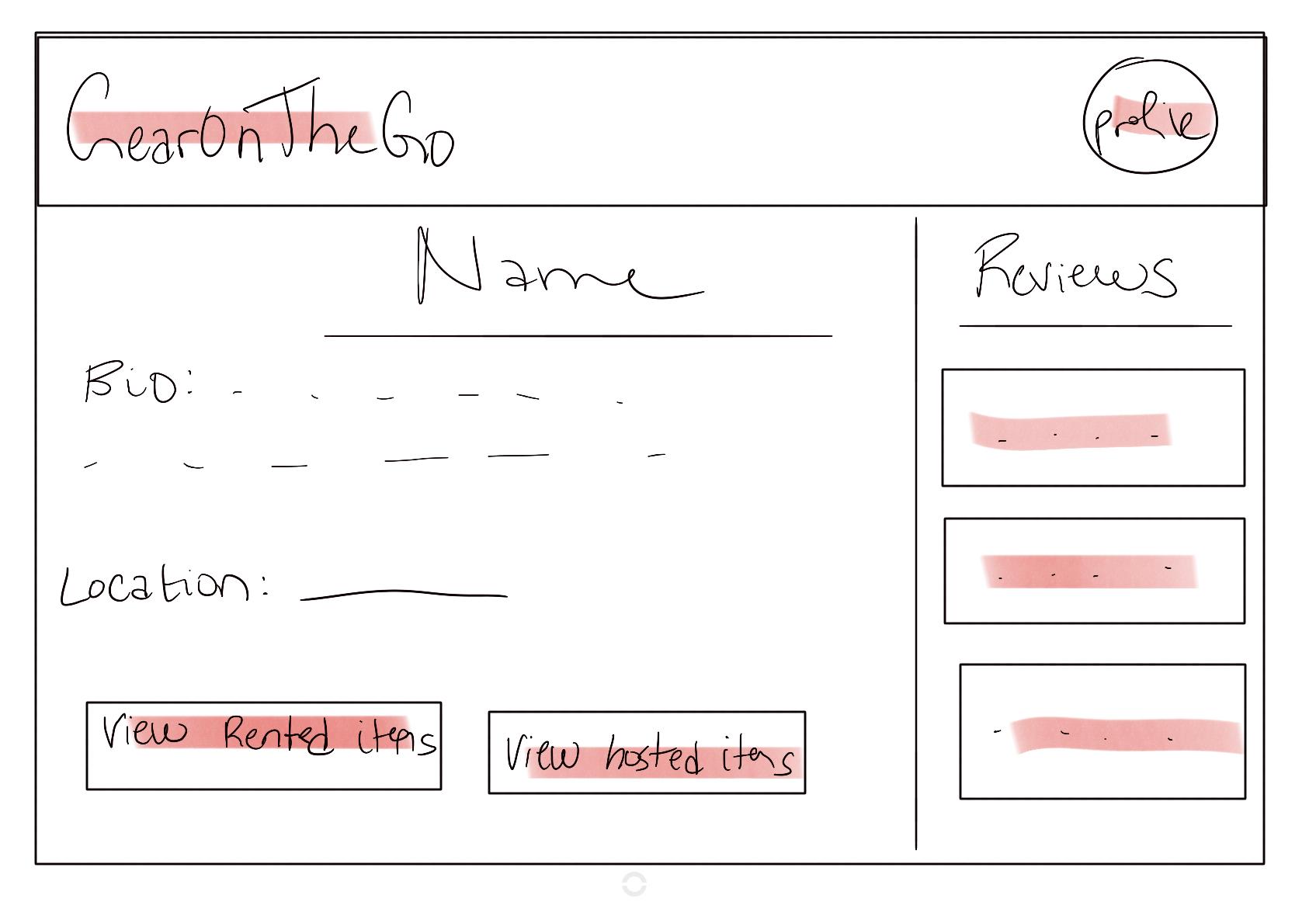
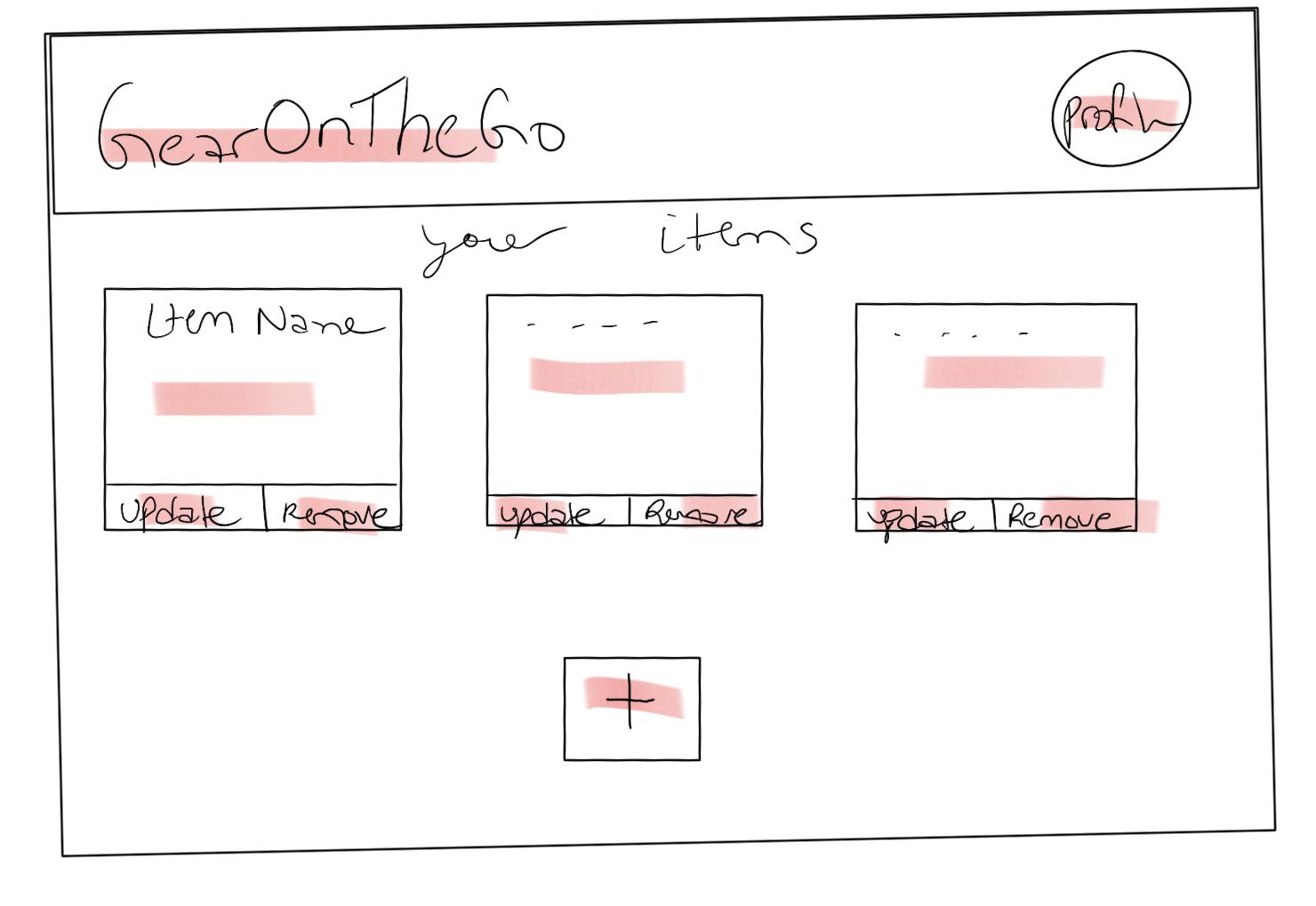
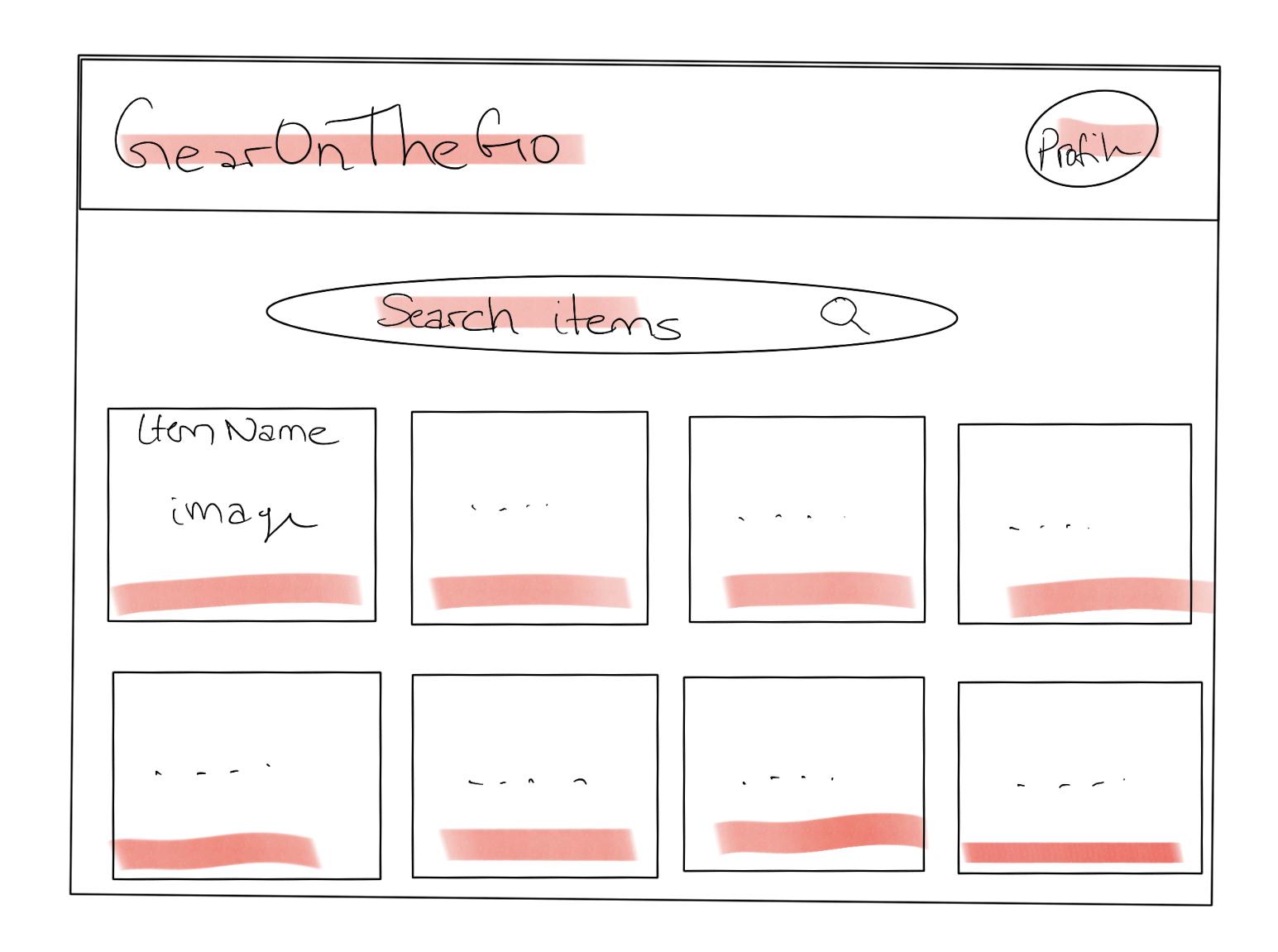


# UI Design

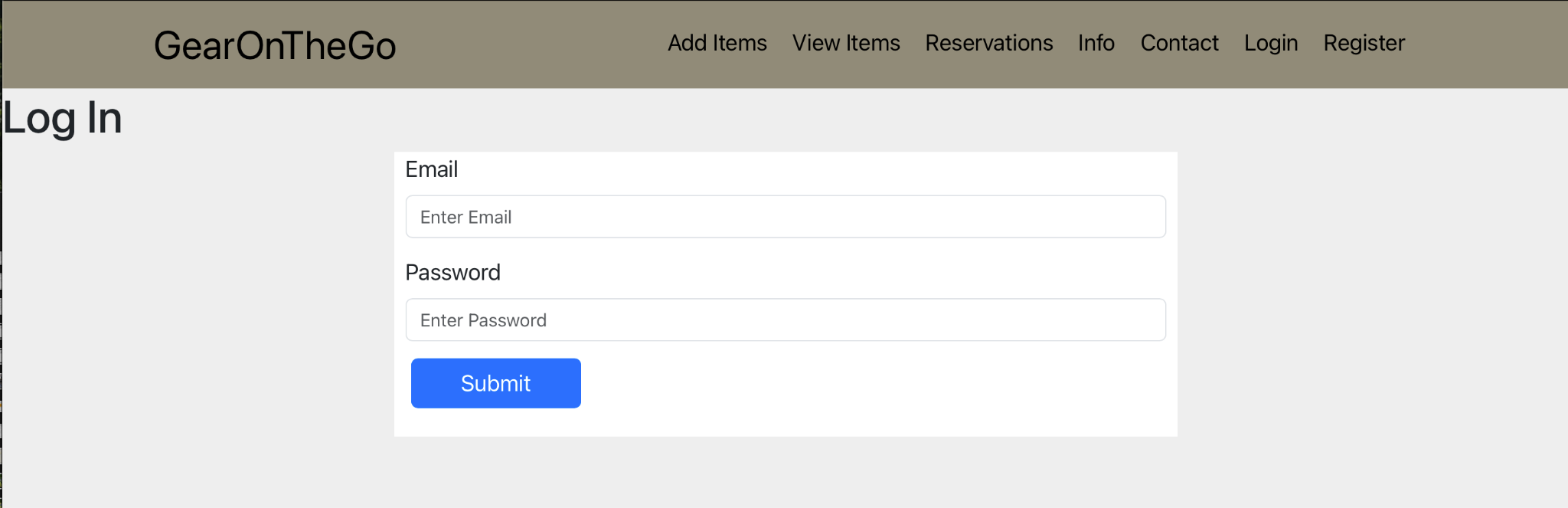
In this section, you can describe your UI design. You can include both your initial design before the implementation and the screenshots of your UI after the implementation.

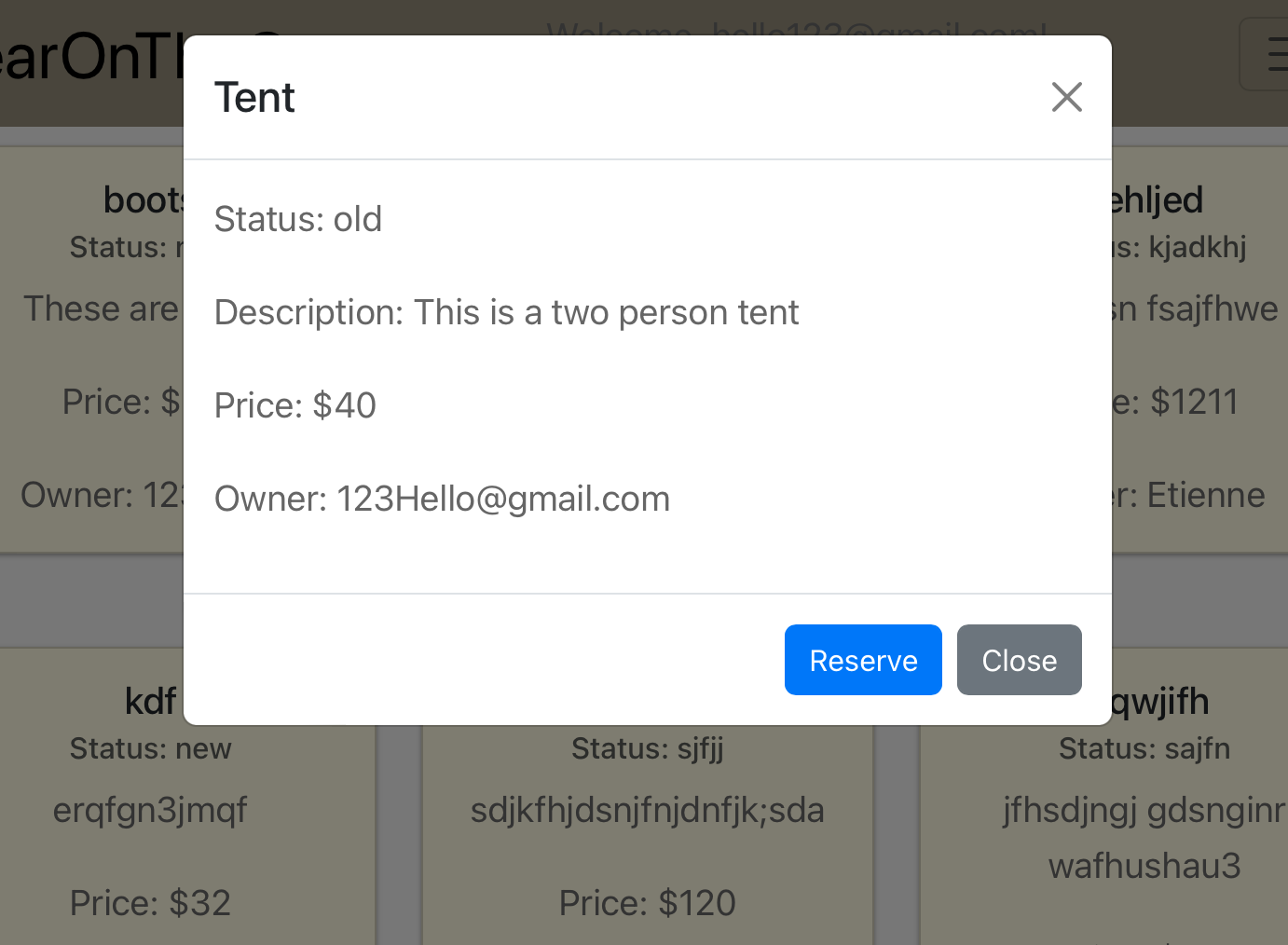
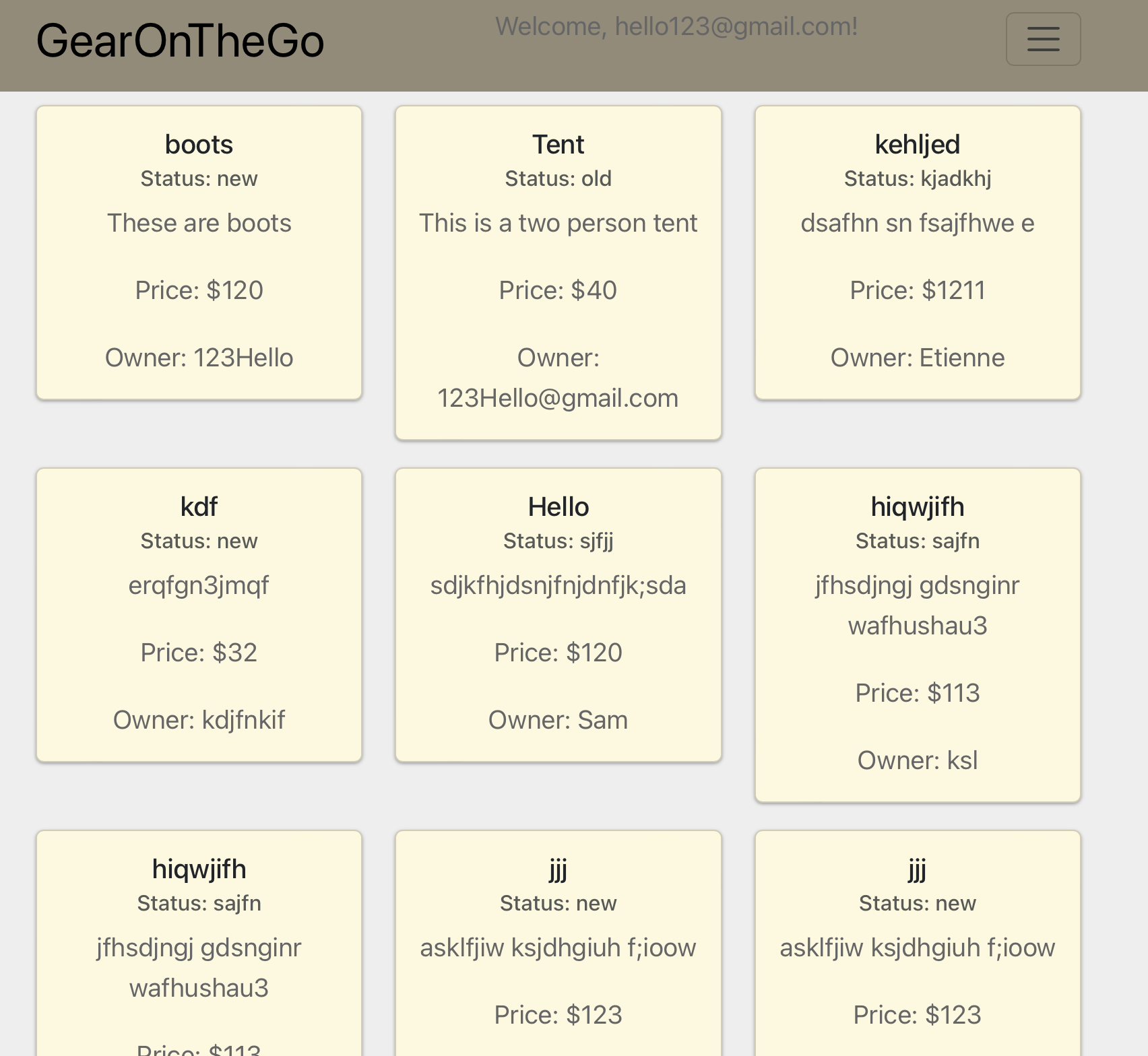
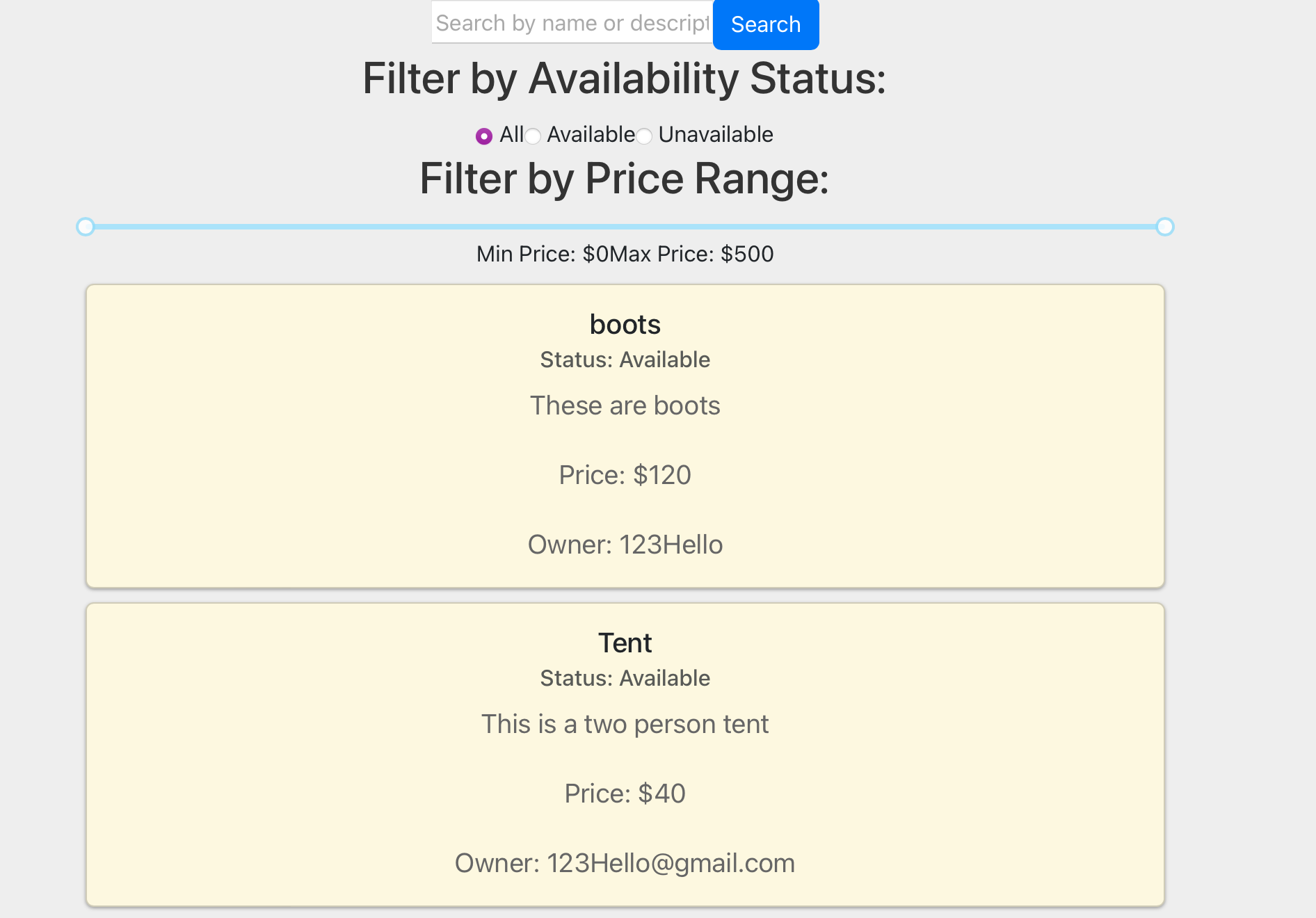
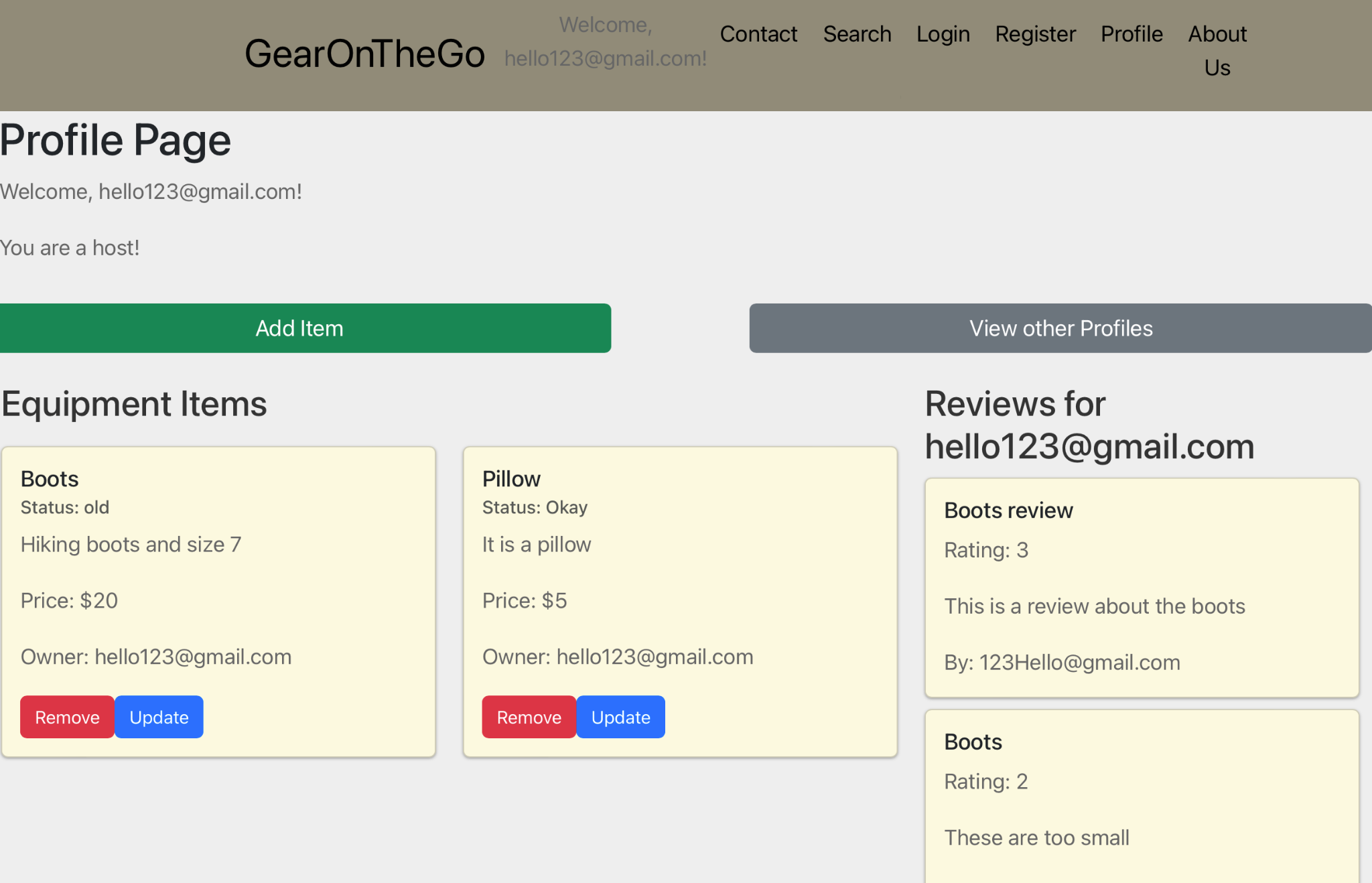
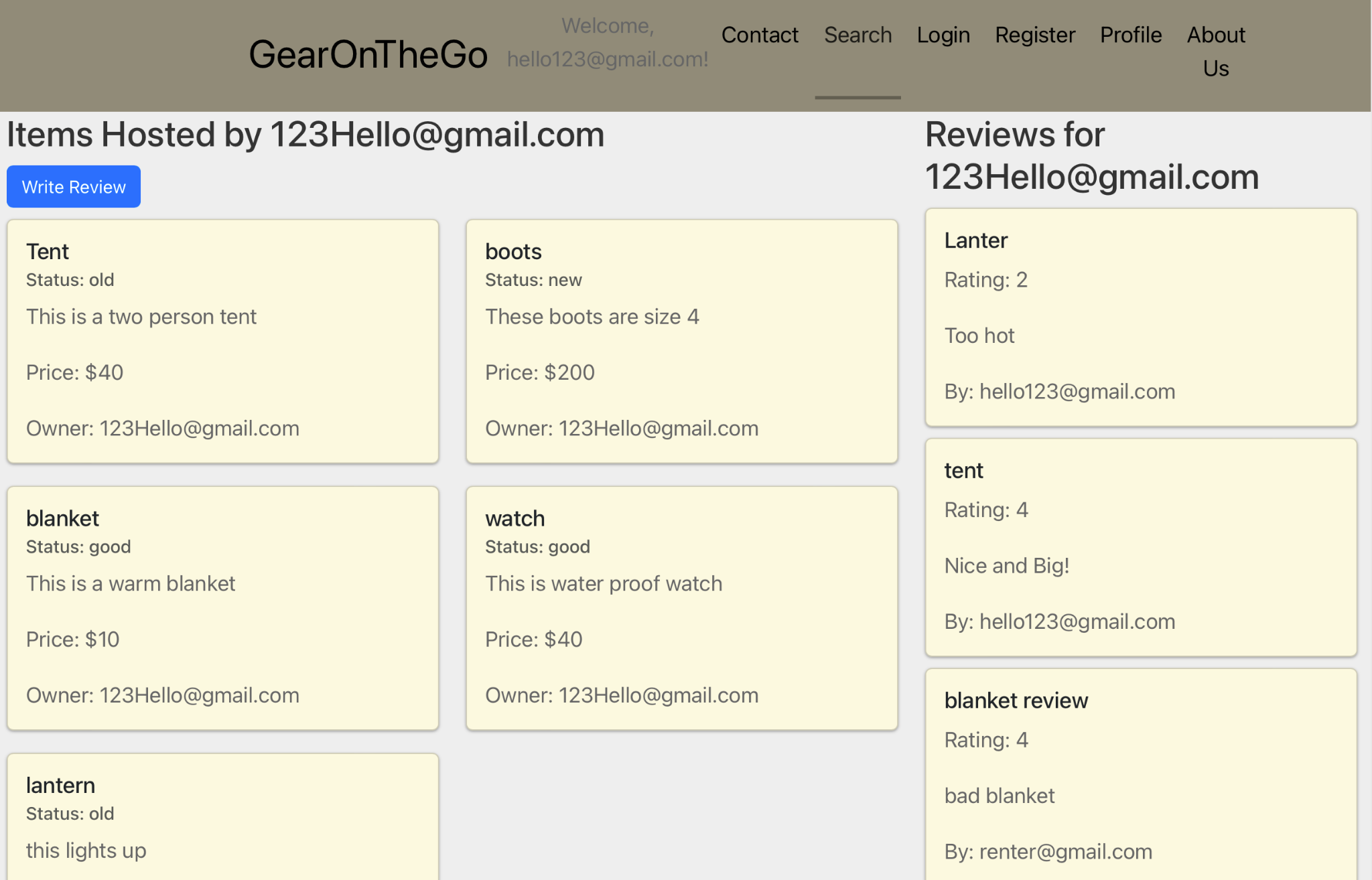
**Design Ideas:**

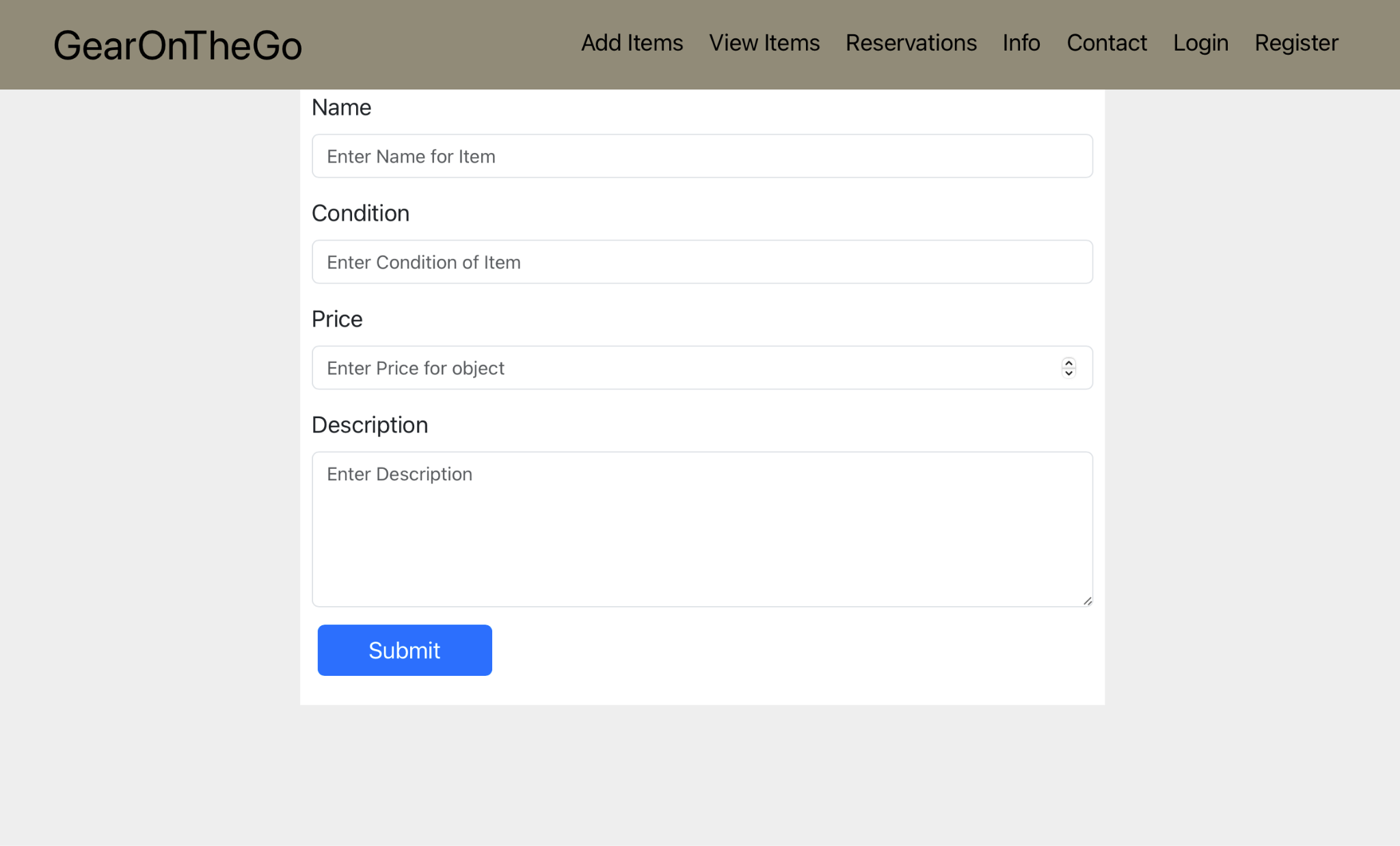


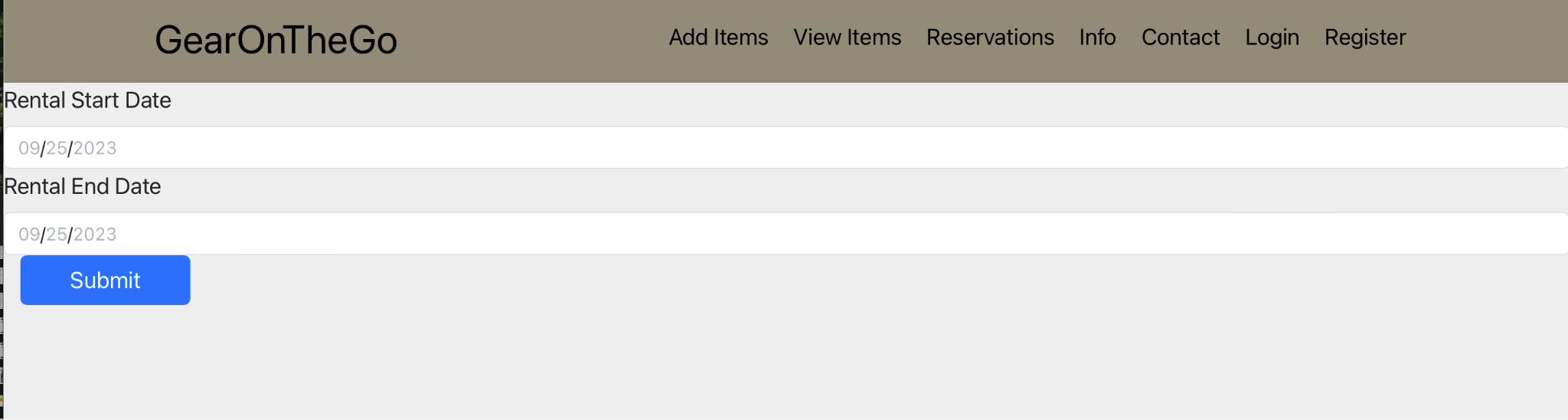


**Implementation after Iteration 1 (will be updated)**

****

****

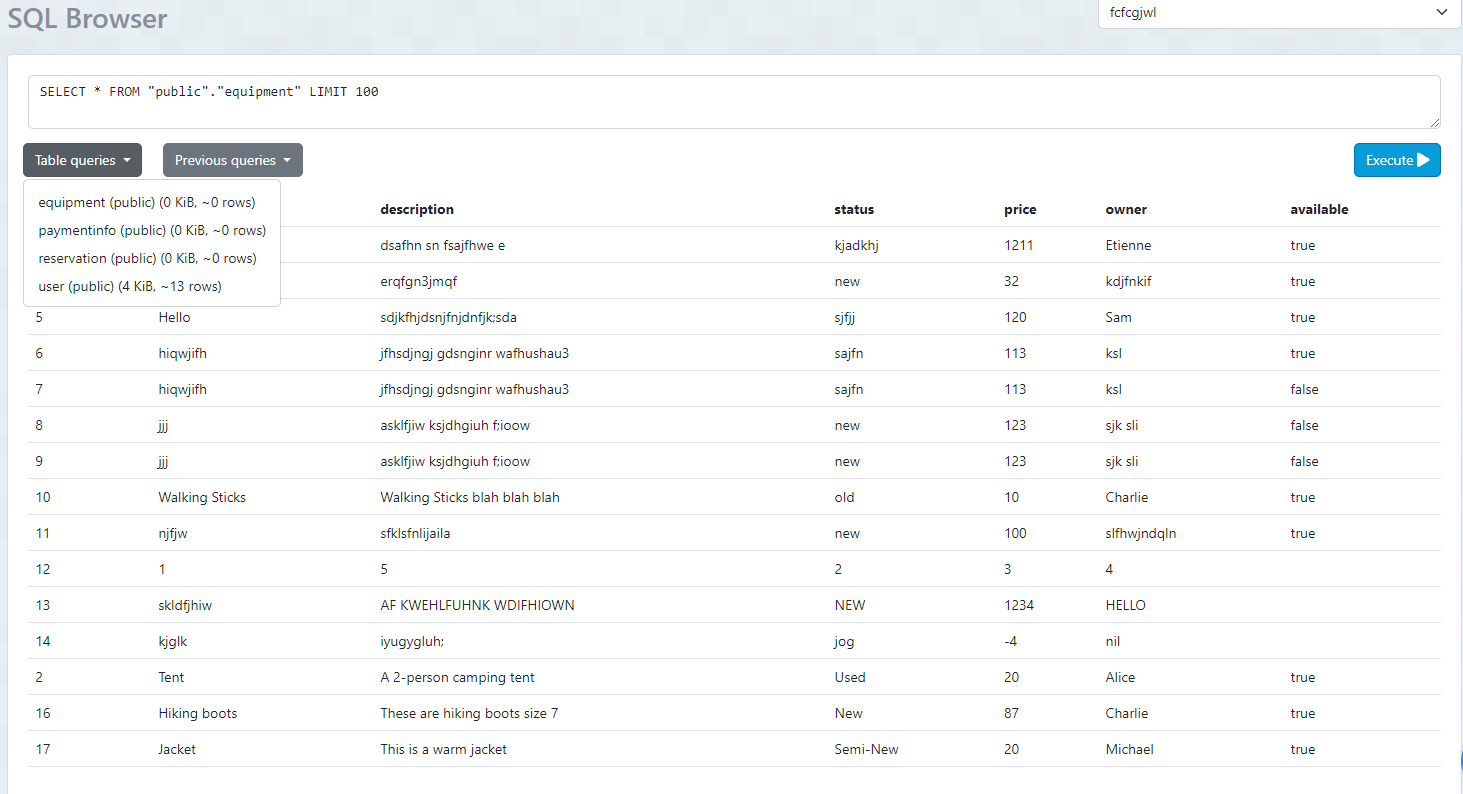
****

****

# Database Design

In this section, you shall describe any database schema if used in your software system.

The below is a representation of how our databases UI looks like on ElephantSQL



The below description of our database tables.

**Equipment Table:**

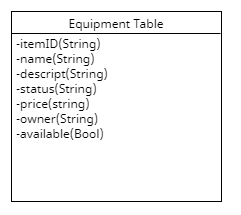
Fields :

itemID(String), name(String),descript(String),status(String),price(string),owner(String),available(Bool)

Primary Keys: Item Id

Foreign Keys(Owner)

Relationship: An Owner must own Equipment that will be tied to the User.



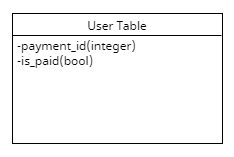
**User Table:**

Fields: id(String),email(string),password(String)

Primary Keys:id

Foreign Key: TBD

Relationships: A User will have Equipment



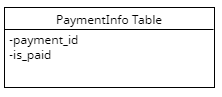
**PaymentInfo Table:**

Fields: payment\_id(integer), is\_paid(bool)

Primary Keys: Payment\_ID

Foreign Key:

Relationships: item id in payment\_info is related to equipment.

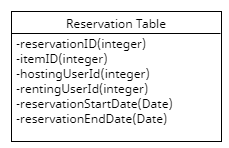


**Reservation Table:**

Fields: reservationID(integer), itemID(integer), hostingUserId(integer), rentingUserId(integer),reservationStartDate(Date),reservationEndDate(Date)

Primary Key: reservationID(integer)

Foreign Key: itemID(integer)



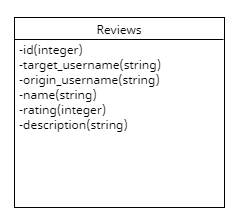
**Reviews Table**

Fields: id(integer), target\_username(string),origin\_username(string),name(string), rating(integer), description(string)

Primary Key: id(integer)

Foreign Key: target\_username(string),origin\_username(string)

Relationships: Reviews table will be related to the User table



# Security Design

In this section, you shall describe any security design in your software system.

* + Hashing password
    - When a user register with their email and password, their password is first hashed and then saved in the database.
    - When a user logs in with their credentials, the entered password is first hashed and then compared with the saved hashed password.
  + User validation
    - Email is validated using regular expressions along with built-in validators with FlaskForm
    - Checks for proper email format such as [xxx@xxx.com](mailto:xxx@xxx.com)
    - Checks for minimum and maximum length
  + Password Validation
    - Password is validated with builtin validators with FlaskForm
    - Checks for minimum and maximum length
  + Forgot password
    - Will include Forgot Password feature later on
  + Routing to specific pages depending on user type
    - Renter and Host have different pages that they can only access
    - Will remove this to have both be able to access all pages
  + Logout user and end session
    - User should be able to logout of their session

# Business Logic and/or Key Algorithms

In this section, you shall describe any key algorithms used in your software system, either in terms of pseudocode or flowchart, or sequence diagrams.

The purpose of the website is for a renting user to rent equipment from a hosting user; however, this process cannot be completed without a payment transaction to the hosting user. In order for the user to make a reservation, upon clicking a ‘make reservation’ tab, they proceed to a Checkout page. The information that is required from the renting user is very straightforward and one that they would be familiar with from other Checkout pages; such as, Full Name on Card, Billing Address and Card Details.

The most critical aspect of this page is to ensure that the renting user does not input inaccurate information, therefore, the Checkout page has multiple checks to prompt the user while they input their information.

The payment does not submit until the Card Number is determined to be valid. The Card Validation has been coded using an algorithm based on the Luhn Method for Credit Card Number Validation. **The pseudocode for this implementation is as follows:**

Split cardNumber into array and store it in the const cardDigits.

Reverse cardDigits array.

Initialize i to 1.

While i is less than length cardDigits:

i. Multiply cardDigits[i] by 2.

ii. If cardDigits[i] is greater than or equal to 10, then subtract 9 from

cardDigits[i].

iii. Increment i by 2.

Initialize a const valid to 0.

For each element val in cardDigits:

i. Add val to valid.

If valid mod 10 is equal to 0:

i. Display alert: "Card number is valid".

ii. Return true.

Else:

i. Display alert: "Invalid card number".

ii. Return false.

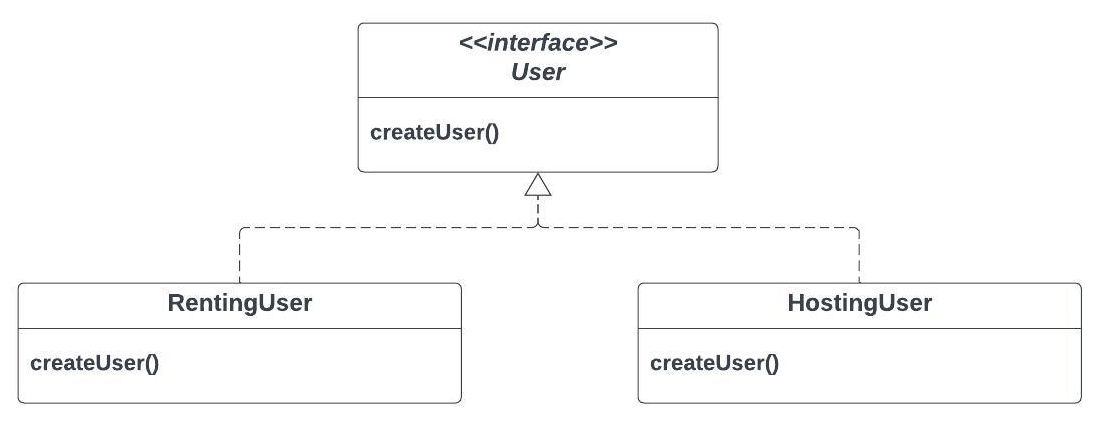
**End pseudocode**

From a monetary standpoint, the checkout page and its payments are what would essentially generate the revenue for the rental equipment service via a commission received on each purchase.

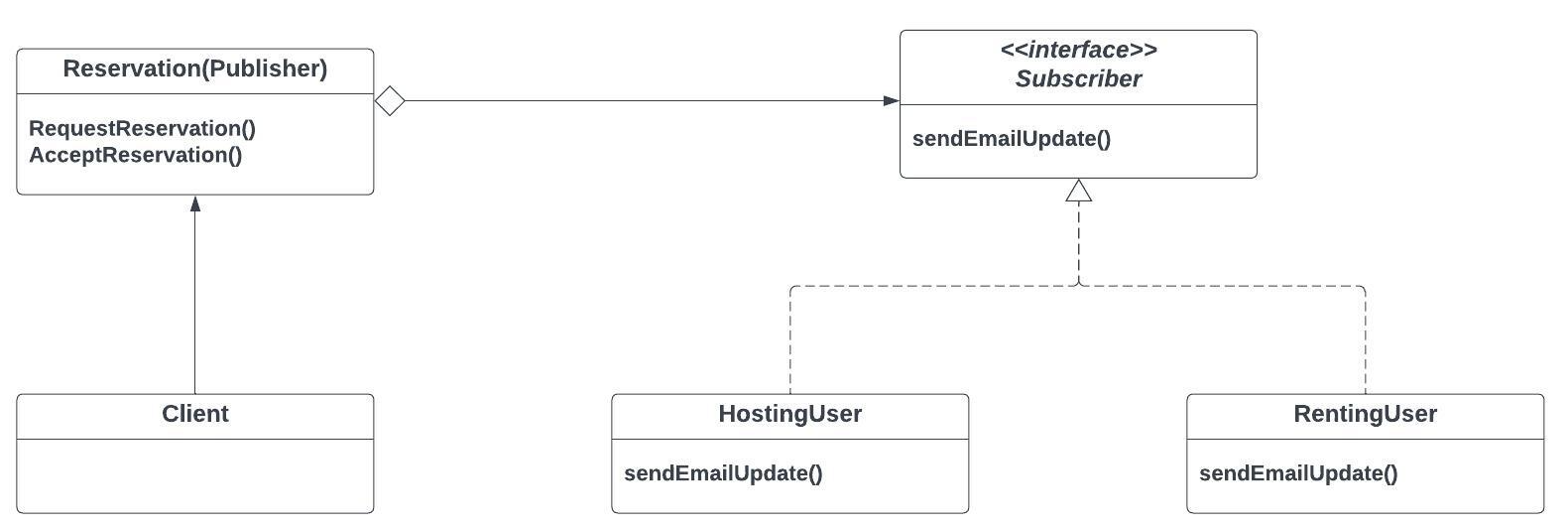
# Design Patterns

In this section, you shall describe any design patterns used in your software system.

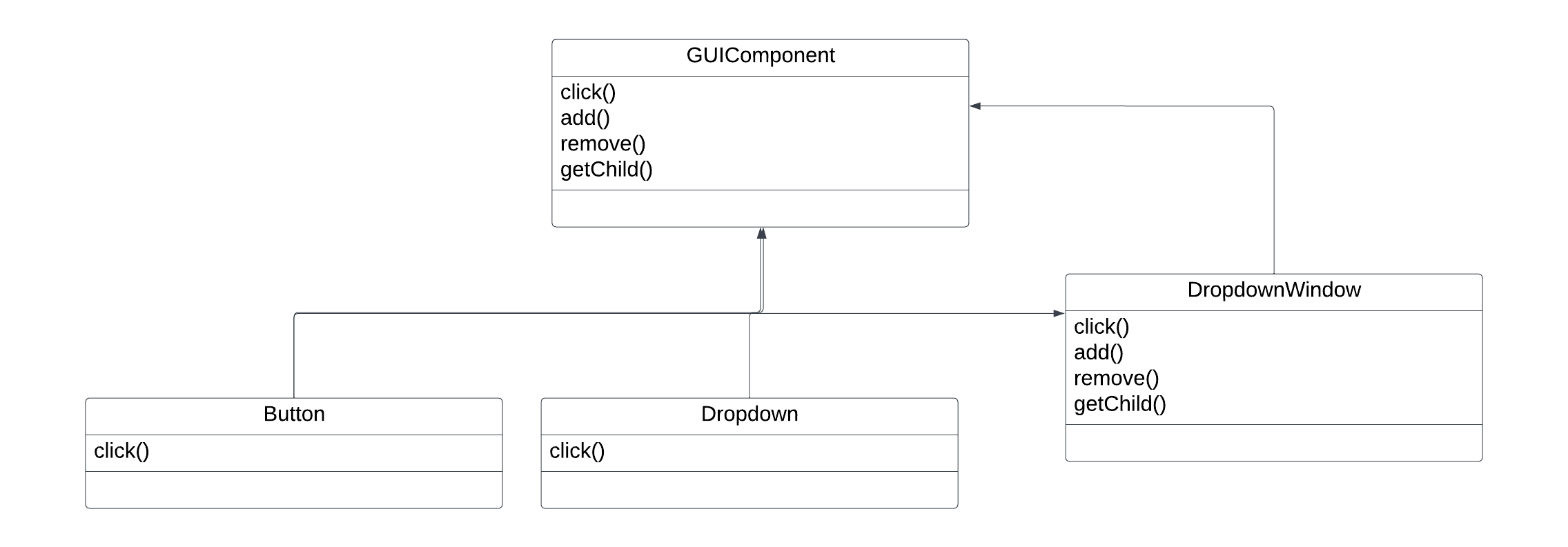
**Abstract Factory:** To create renting users and hosting users, we shall be creating an interface ‘User’ which will have methods for the common characteristics that the inheriting classes, renting user and hosting user can use, so in creation of users, we may not need to declare concrete classes while just creating users.



**Observer :** Since observer design pattern is a behavioral design pattern that lets the subscribers get notified, we shall be utilizing this design pattern for sending notifications to renting user and hosting user everytime a reservation has been requested or approved. This will help users get confirmation emails.

****

**Composite**: Composite design pattern is a structural design pattern that breaks the parent component into granular child components. We plan to be utilizing this design pattern for putting together frontend GUI components.



# Any Additional Topics you would like to include.

# References

* + [Lucidchart](https://www.lucidchart.com/pages/landing?utm_source=google&utm_medium=cpc&utm_campaign=_chart_en_us_mixed_search_brand_exact_&km_CPC_CampaignId=1457964857&km_CPC_AdGroupID=57044764032&km_CPC_Keyword=lucidchart&km_CPC_MatchType=e&km_CPC_ExtensionID=&km_CPC_Network=g&km_CPC_AdPosition=&km_CPC_Creative=354596046394&km_CPC_TargetID=kwd-33511936169&km_CPC_Country=9061293&km_CPC_Device=c&km_CPC_placement=&km_CPC_target=&gclid=CjwKCAjwmbqoBhAgEiwACIjzEEb4a8cj9JSyks3w0Lkmb5rbOHjylfUWDmkog8YQQ6C045RavMXBjhoCkGUQAvD_BwE): Used for creating UML diagrams
  + Umlet: Used for creating folder structure

# Glossary