BoilerTradeCo

CS307 Design Document

Team 32: Neha Rajamani, Isha Singhal, Puja Teakulapalli, Vinathi Muthyala

Table of Contents

Purpose

Students at Purdue University currently face challenges in efficiently buying and selling commonly needed items within the large campus network. Acquiring and disposing of furniture and other common items has always been a struggle expressed by college students, as most of the items they buy throughout the year are only intended for temporary purposes. The lack of a dedicated platform that focuses on these issues has resulted in an unorganized system where students find themselves having to navigate through a myriad of different channels.

Introducing a specialized website for students in West Lafayette can tailor these transactions, creating a consolidated space where students can easily find what they are looking for. Targeted users are students or faculty at Purdue who are interested in making money off items they no longer have use for and/or purchasing items in “good condition” rather than new ones to save money. These individuals at Purdue can be those who are usually moving out at the end of the year, switching between dorms or apartments, or graduating and wanting to sell their items away. Additionally, students can use BoilerTradeCo to sell any textbooks, iClickers, lab equipment, or other school supplies when students are done using these materials and would like to sell them for a cheaper price.

“Craigslist” is a platform that is designed for individuals to connect for a multitude of purposes that include: buying and selling used items, finding housing, or promoting local events. Similarly, “Facebook Marketplace” is a platform designed to allow users to buy or sell items within their local communities with their main form of communication being “Facebook Messenger” which is operated in-system. While our team acknowledges the fact that there are other platforms with a similar purpose to our application, like “Craigslist” and “Facebook Marketplace,” our application, BoilerTradeCo, distinguishes itself by being a channel solely dedicated to students at Purdue.

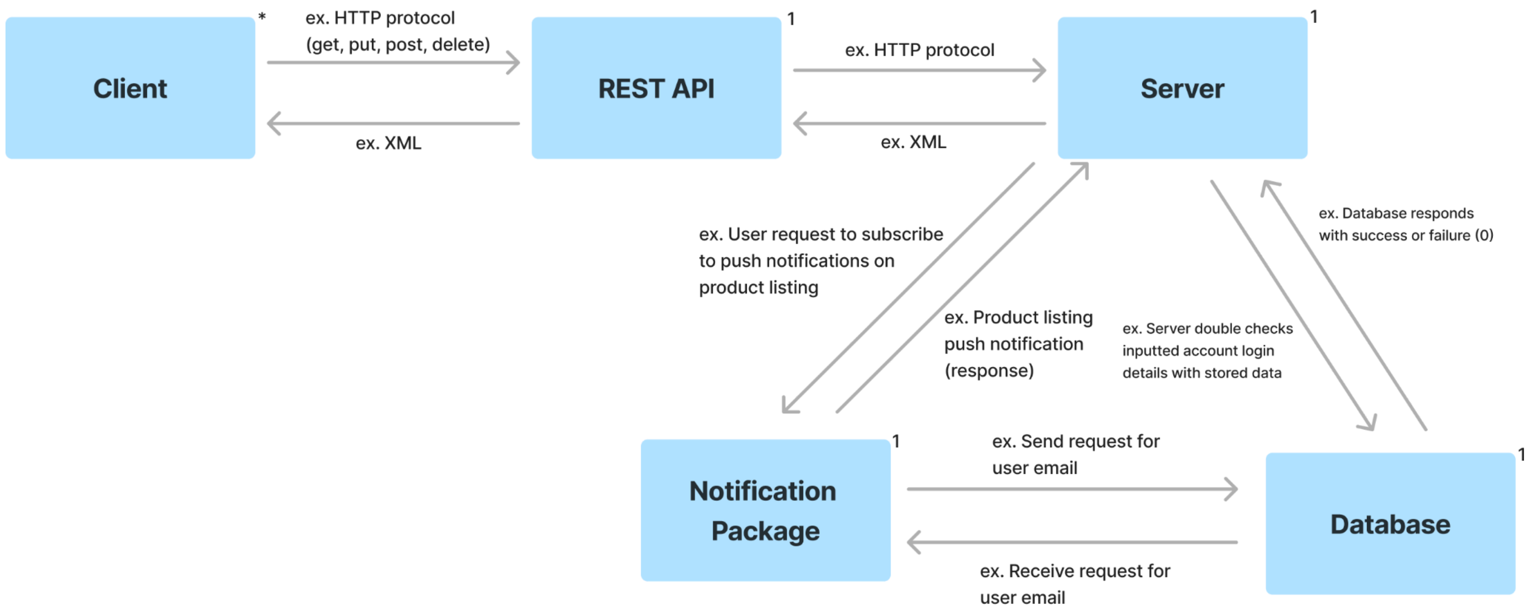
Unlike these other, broader platforms, BoilerTradeCo is catered exclusively to the Purdue community. The Purdue-specific focus of the problem transforms the platform into a trusted environment structured to seamlessly facilitate connections between students buying and selling items. The campus-centric focus promotes the exchange of Purdue-related niche products like professor-required materials for classes such as specific textbooks, lab material, and game day tickets, allowing greater access for Purdue students to buy cheaper items and participate in community events when sold out. This exclusive focus, by attaching the Purdue name, additionally adds a layer of reliability and trust to each product as the system is designed to connect students when going through the process of purchasing or selling any given item.

Design Outline

High-Level Overview:

This web application will be utilized as a marketplace for Purdue students and faculty to buy or sell their second-hand, commonly-needed college items. The application will use a client-server model integrated with a REST API to ease interactions between the client and the server along with a Notification API. The Notification API will be implemented with API integration as well and will facilitate sending push notifications to buyers or sellers when there is product interest. The server will receive the API-modified requests from the client and complete those requests by communicating with the database if necessary to retrieve any data required for the request. The database will be utilized for long-term storage and management of data and information pertaining to any users and the web application; Heroku will be used in this system. The web application will use Python with a Django framework for the back-end implementation of the client-server model and HTML, CSS, and Javascript for the front-end website structure, styling, and interactivity.

UML Diagram of High-Level Structure of System:


Client
*
ex. HTTP protocol (get, put, post, delete)
ex. XML
REST API
1
ex. HTTP protocol
ex. XML
Server
1
ex. User request to subscribe to push notifications on product listing
ex. Product listing push notification (response)
ex. Server double checks inputted account login details with stored data
ex. Database responds with success or failure (0)
Notification Package
1
ex. Send request for user email
ex. Receive request for user email
Database
1

Client

The user interface (UI) provides a visually appealing interface for users to interact with and make requests. The client will prompt user interactions and there will be local storage and caching for certain data. When a user interacts with the client, such as pressing a button on the website, the client will process the interaction and update the display simultaneously. When necessary, the client communicates with the server through the REST API to get the appropriate response or fetch the necessary data to complete the user's request. Some examples of client-side requests are the HTTP protocol and CRUD actions, such as get, put, post, update, etc, applicable to the web application. Languages used for the client front-end web application will be HTML for structure, CSS for website styling, and Javascript for interactivity. For the client-server interaction, Python with a Django framework will be used.

Server

When the client sends a request to the server through the REST API, the server is responsible for processing the request and executing any necessary steps to send the appropriate response back to the REST API which will convey it to the client, so that it can be updated for the user in real-time. When necessary, the server communicates with the database to retrieve relevant data needed for the API-modified request from the client, and the database responds with success. An example of a server interaction with the database would be to confirm account login details or product listing details by a given seller with the stored data in the database. The language being used for the server back-end will be Python using a Django framework to help implement the client-server interaction. The Application Programming Interfaces (API) will also facilitate the data transfer between the client and server.

Database

The database is a collection of all data relevant to users and the web application, and is used for long-term storage and management. The management of the data will be stored in a systematic way to facilitate more efficient retrieval or modifications for client-API-server requests and corresponding responses. When necessary, the database will communicate with the server to output the data requested by the client-API-server interaction or by other Application Programming Interfaces (APIs) utilized in the web application. The database will either respond with a success value, the data retrieved, or zero to represent an error or no data returned. An example of a database purpose would be to store login credentials for each user with an account. Additionally, the database will meet any data requests from packages, an example being the push notification package to send notifications to users about their product listing. The database used in this system will be Heroku.

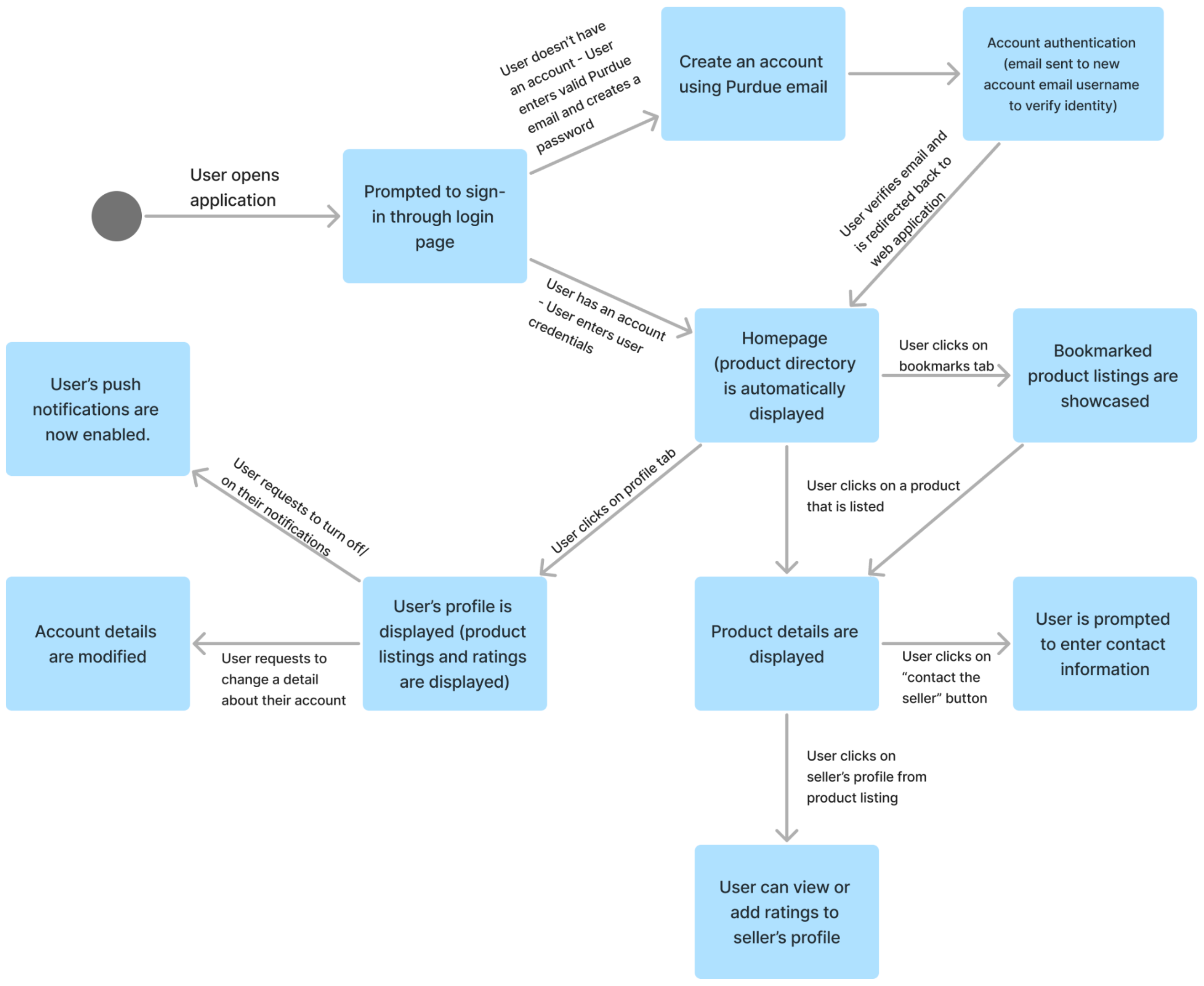
REST API

The Representational State Transfer Application Programming Interface's, or REST API, architectural style is responsible for the client and server interactions management where it emphasizes communication and resource-based interactions. Each request from the client for the server contains all the information needed which creates an effortless, efficient transaction between the client and server. REST APIs use standard HTTP protocols as well as communication in JSON or XML. The web application will use API integration for the REST API implementation.

Push Notification Package

A Push Notification package provides a set of tools and instructions to help develop and facilitate adding a push notification system to their applications. Push notifications are alerts that are sent directly from the server to the user, through email in this case, even when the user is inactive on the web application. This package helps to directly send notifications to their email when triggered and ensures reliable and efficient delivery of the message. This helps users not only stay aware of product listings but also reminds them of the application's existence in the first place, which helps promote the application. The package is directly implemented in the parts of the server in a class of its own to adhere to general standards of code cleanliness. The main package incorporated will be one specific to Django titled: "Django-notifier". The package contains various methods to help implement all kinds of notifications for one backend. For our function of email notifications, the package uses the help of some Django email settings to assist in continuous notifications.

<u>Activity/State Diagram</u>:


User opens application
Prompted to sign-in through login page
User doesn't have an account - User enters valid Purdue email and creates a password
Create an account using Purdue email
Account authentication (email sent to new account email username to verify identity)
User verifies email and is redirected back to web application
User has an account - User enters user credentials
Homepage (product directory is automatically displayed
User clicks on bookmarks tab
Bookmarked product listings are showcased
User clicks on a product that is listed
Product details are displayed
User clicks on "contact the seller" button
User is prompted to enter contact information
User clicks on seller's profile from product listing
User can view or add ratings to seller's profile
User clicks on profile tab
User's profile is displayed (product listings and ratings are displayed)
User requests to change a detail about their account
Account details are modified
User requests to turn off/ on their notifications
User's push notifications are now enabled.

This diagram displays basic features the user will have access to. A more detailed activity diagram is located in Design Details.

Design Issues

Functional Issues:

1. How should we verify that the person creating an account has a valid PUID?
    a. Option 1: Connect through their career account
    b. Option 2: Verify their @purdue.edu email address
    c. Option 3: Submit an image of proof

   Decision: Verifying their Purdue email address is the best way to ensure their valid ID status. Connecting through their career account may be difficult because the web application, while affiliated with Purdue, is not directly connected to the University. If we try to ask for access through their career accounts may be a long, unwarranted process. Submitting proof will create a reason to tinker with visual sensors which isn't plausible in the timeline we have. As for possibly entertaining manual verification, it could cause a large delay. The most realistic option out of the three proposed would be email authentication. If the purdue.edu email they submit for their username is valid, they'll be able to click on the verification link provided by the system. The server will check if the email provided by the user has a suffix of purdue.edu and send a verification email respectively. The user will be expected to check their inbox for a verification link to properly verify their account. We're planning on using some of Django's built-in features and packages to help provide us with templates to correctly develop this feature (namely the django-registration package).

2. How should sellers process a transaction/trade for a product with buyers?
    a. Option 1: Billing and shipping page for product
    b. Option 2: Communicate externally to decide on payment and product retrieval methods

   Decision: Communicating externally is the best way for buyers and sellers to decide how they would prefer to pay and get their product. While a billing or shipping page seems functional, the ease and safety of the web application would be reduced as sellers would have to package their product before giving it to the buyer, and both buyers and sellers would have to reveal their living/shipping addresses to the other. If they choose to communicate externally, both users can choose a common meeting spot to coordinate the drop-off/retrieval of the product and the payment at the same time. Implementing a billing page could also be difficult because we would have to process and verify various credit cards and ensure the payment goes through successfully to the seller. This is more difficult than allowing users to choose their preferred payment method, which is often Venmo, Zelle, or cash, which are also all easily done by students on their own. Additionally, a shipping page would be difficult because the web application would have to take care of sending the product package from the seller to the buyer and coordinate with shipping companies and their fees, which is not realistic or very feasible for the timeline and scope of this project.

3. How should buyers be able to review their experience with a seller?
    a. Option 1: Leave a star rating on the profile of the seller
    b. Option 2: Leave a descriptive review on their interactions with the seller

Decision: Allowing the users to **leave a star rating on the seller's profile** was the better option in this case. While the idea of being able to leave a descriptive review sounds appealing, it takes away from the simplicity of the website. Having every single descriptive review present on a user's profile causes the overall display to be messier than what we're advertising. By creating a star system, we can also create statistics that other users will easily comprehend. Every star rating that is inputted will be averaged and displayed accordingly on the user's profile.

4. How will sellers know if there is/are buyer interest in a product?
    a. Option 1: Direct message seller directly on web application
    b. Option 2: Click on an "I'm interested!" button which will send an email to the seller about potential buyer interest and preferred contact methods
    c. Option 3: Click on an "I'm interested!" button which will send a notification to the seller about potential buyer interest to an inbox on the web application

Decision: Clicking on an "I'm interested!" button which will send an email to the seller about potential buyer interest and preferred contact methods seemed to be the most optimal option. Implementing a direct messaging system on the web application itself for buyers and sellers to communicate on seemed to be a more difficult and unnecessary feature on the platform, as users often prefer to communicate via email, phone number, or their social media accounts like Instagram over a messaging platform only available on the web application since it is more efficient and easily accessible. Direct messaging may also be harder because the system would have to support adding a message, editing or removing a message, and saving messaging data on the platform in between logins. On the other hand, clicking on an "I'm interested!" button that will send a notification to the seller about potential buyer interest to an inbox on the web application would pose a similar problem as users would have to access the message on the website instead of their preferred method of contact such as email, phone number, or social media which they check and get regular notifications on already. By clicking on an "I'm interested!" button that sends an email to the seller about potential buyer interest, the seller does not have to repeatedly check the web application if buyers are trying to communicate with them and can contact buyers using email and/or their preferred contact method.

Non-Functional Issues:

1. Which framework should we use for the back end?
    a. Option 1: Node.js
    b. Option 2: Spring Boot
    c. Option 3: Django

    Decision: Django was the framework we went with. Django has a large, extensive set of different tools and libraries available to use to assist us with our development process. It also encourages code cleanliness, so it's far easier to look back on. Node.js uses JavaScript to create all of its features. Not only does our group not have that much experience with JS, but through our research, Node.js may prove to be problematic when trying to connect all of our frameworks to make one cohesive application. Spring Boot seemed like a viable option at first, but we wanted a somewhat new experience with this project. Java is a language we all find comfort in, but it won't expand our knowledge in the manner we would like it to. Django, being a Python-based platform, could offer us that new information we're lacking. We also find that at times, Java can be a messier language to code in, and organization is something we strive to maintain. Although we recognize that debugging any issues we run into might be easier with Java, we see Python being much more succinct and readable as we cross-collaborate on code. Django also automatically implements security features by creating a web application through its platform, which adds an extra measure to prevent any common vulnerabilities that could arise, such as a robust user authentication system (securely handles authentication for the protection of the user with features like password recovery), a secure URL routing system (prevents directory traversal attacks), etc.

2. Which cloud platform should we use for our database?
    a. Option 1: Firebase
    b. Option 2: Heroku

    Decision: In this scenario, **Heroku** was the optimal choice. Primarily, for a project of our magnitude, opting for a developer-friendly platform to host our database proved to be the better decision. Heroku has built-in commands, being a PaaS, that aid developers of our skill level. An easy-to-use platform along with its free services rendered Heroku the superior option compared to Firebase. Firebase is not known for being an easy platform to quickly understand – working with Firebase comes with an intense learning curve on its own. Since Firebase is a platform that is exclusively owned by Google, it results in somewhat of a proprietary lock-in, which would make it difficult to understand across platforms which might not be the best for a project where our main ambition is to expand our knowledge in an applicable way.

3. How should we implement our REST API?
    a. Option 1: API integration
    b. Option 2: Create our own REST API

   Decision: <u>API integration</u> makes the most sense for the scale of our web application. In order to stay within the guidelines of the project, we don't have enough time to spend on the process of developing our own REST API. While it is possible, the process is pretty tedious and not super approachable. There are many REST APIs available on the internet that will be able to accommodate our project comfortably. All REST APIs are used for similar purposes so there shouldn't be any issues with using API integration compared to coding one ourselves.

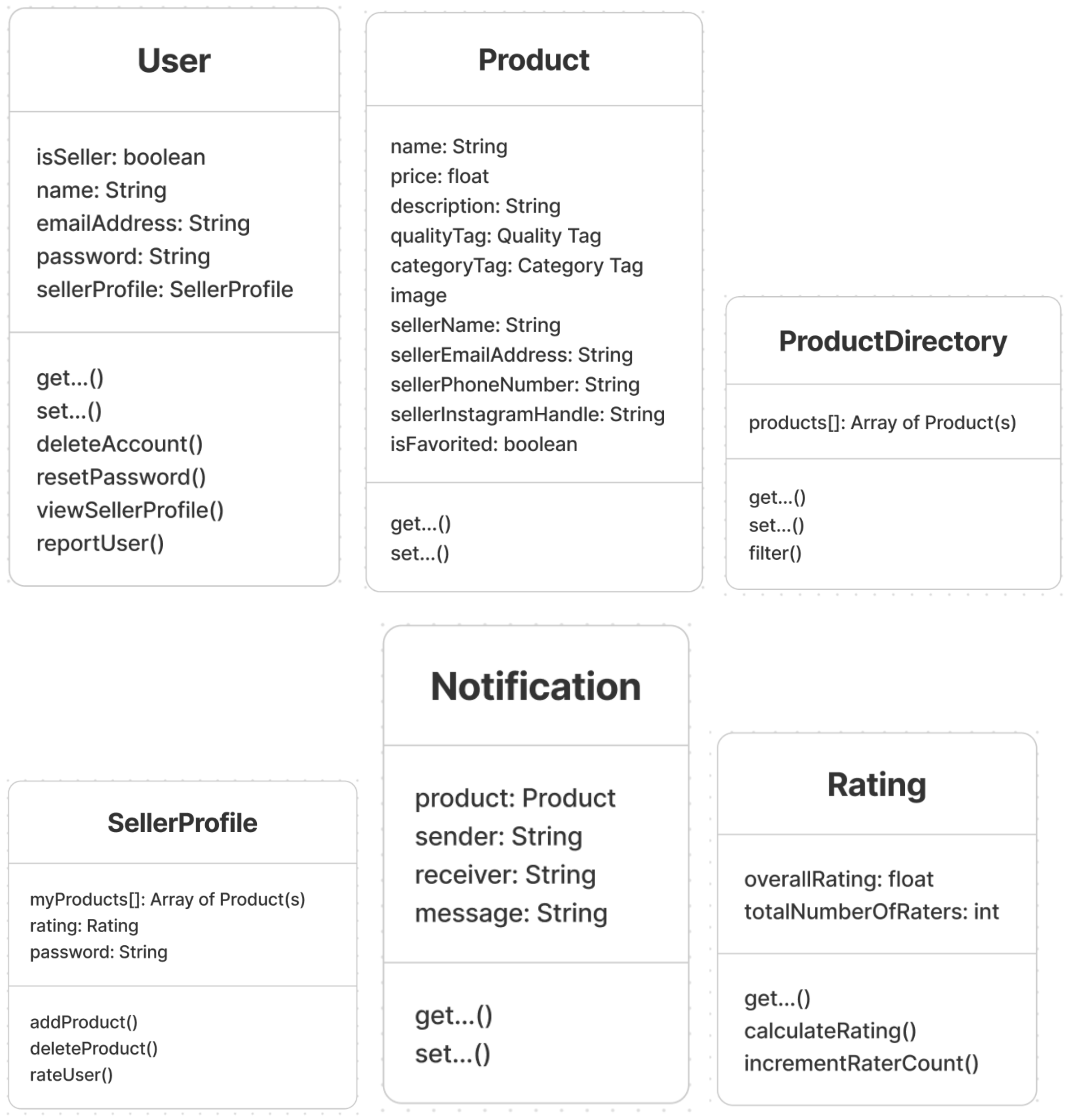
4. How should we implement push notifications for users?
    a. Option 1: Push notification API to send notifications via email
    b. Option 2: Push notification package to send notifications via email

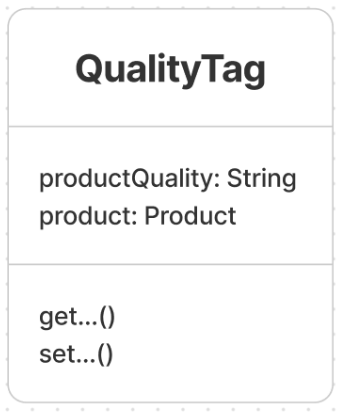
   Decision: <u>Push notification package</u> is the optimal choice for the web application as it is easier to use and is more functional for our usage of push notifications. Since push notifications will be sent via email and not another method of contact to buyers and sellers, a push notification-specific API is not necessary and can become more complicated since implementing a package available in Django for push notifications is sufficient for the usability of notifications in our web application. Additionally, through extensive research, a push notification API required a considerable sum of money with many not compatible with Django making the choice favor in the option of using a package. Instead of integrating a separate API for push notifications, using the package allows for various methods of notifications to be implemented using one backend. The main package incorporated will be "Django-notifier," and will efficiently satisfy our functionality of email notifications as the package uses the help of some Django email settings to assist in continuous notifications.

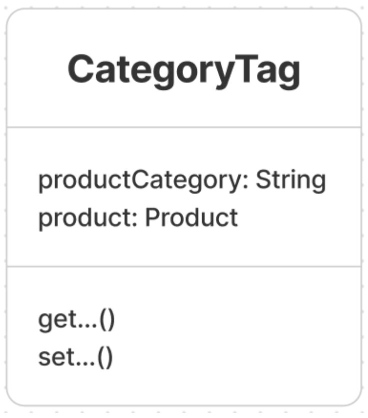
Design Details

(a) Include class level design of the system (i.e. class diagrams) and be as detailed as you can.

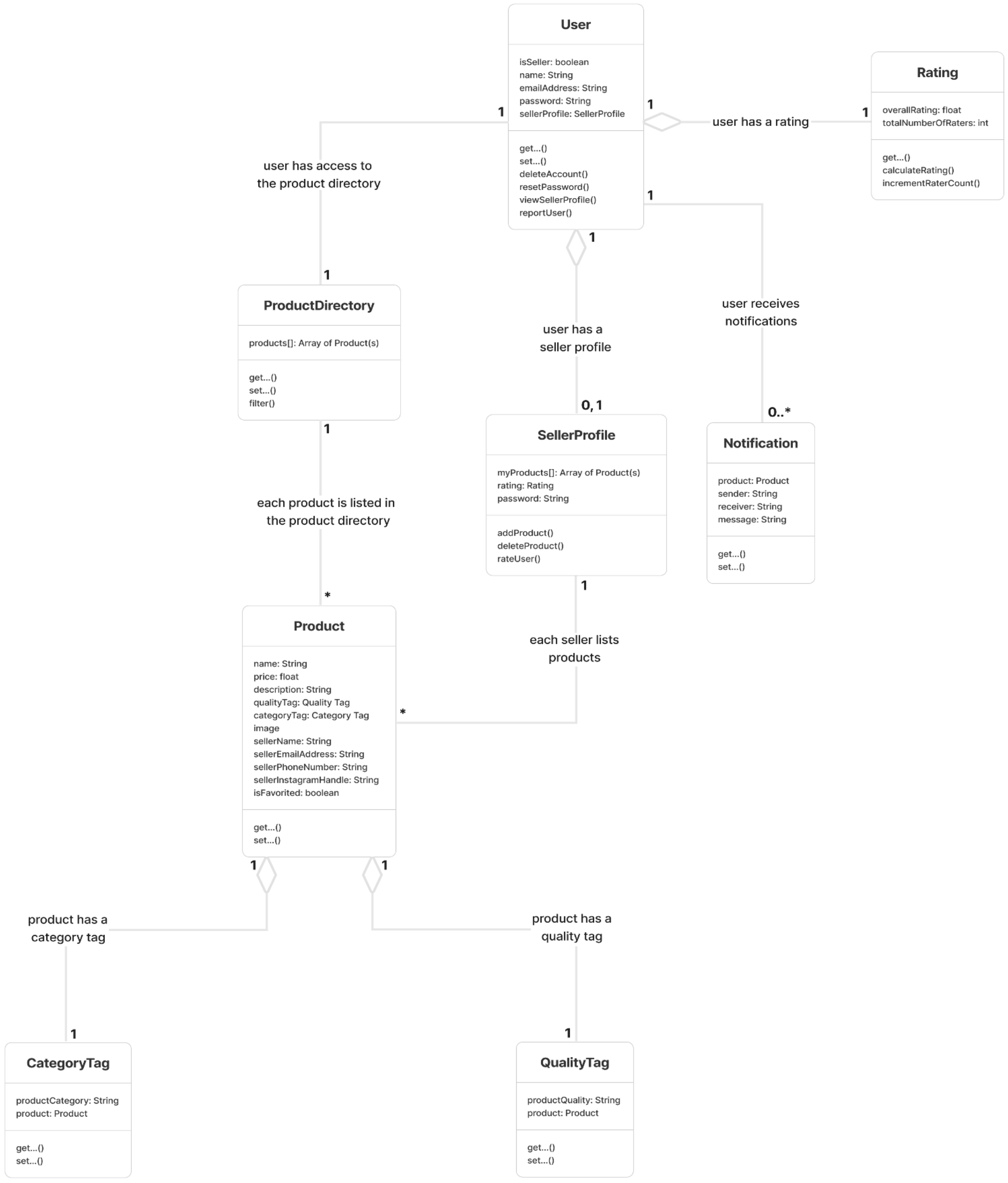
<u>Class Diagrams</u>:




QualityTag
productQuality: String
product: Product
get...()
set...()


CategoryTag
productCategory: String
product: Product
get...()
set...()

<u>Class Diagram with Associations</u>:



Description and Interaction between Classes:

The classes in our web application are structured according to the objects below. Each class contains a set of attributes representing specific characteristics.

User

- The user object is created whenever an individual signs up on our application and they can be either a seller or buyer
- Every user will have a username, password, and name
- Every user will also have a Notification object and a ProductDirectory object
- There is also an option for a user to choose if they want to be a seller, which is the IsSeller attribute
- If the user chooses to also be a seller, then they will have a seller profile object

SellerProfile

- The SellerProfile is created whenever a User chooses to opt into being a seller, rather than simply being a buyer
- The SellerProfile object has an array of Products which are listed by that specific user
    - Any given buyer can view the SellerProfile for a seller from a product posting and view all of the other products the seller currently has listed
- The SellerProfile object also has the Rating object, phone number, and instagram handle

ProductDirectory

- The ProductDirectory is visible to a user upon creation of their account
- The ProductDirectory will consist of an array of Product objects
    - This array consists of all the products currently listed by all sellers, such that buyers can browse the available items

Product

- A product object is created when a user adds a new product
- Each product will have a product name, price, and image
- Each product will also have a description. When users search for a certain keyword, that keyword will be used to match the words in the product's description
- Each product will also have an isFavorited value if a user wants to add a product to their favorites
- Each product will also have a quality tag that is a QualityTag object
- Each product will also have a category tag that is a CategoryTag object

<u>Notification</u>

- The Notification object is created when a user shows interest in a product and tries to contact buyer
- The Notification object is also created when there is a new product posting in a specific category or if a previously bookmarked object has been sold or deleted
- The Notification object is also created if a product's price has changed from its original price
- The Notification object includes an product, sender, receiver, and a message depending on the reason for the notification
- The Notification object is used to send a notification to the seller from the Notification API
    - This notification will consist of the interested buyer's email address, as well as their handle for their preferred method of contact if the seller also has that method of contact uploaded to their profile

<u>Rating</u>

- The Rating object is created when a SellerProfile object is created
- Users who are buyers can leave ratings for sellers who they have purchased a product from
- The rating is computed as the average of all rating received
- Maintained as a separate class to maintain modularity and separation of responsibilities
    - User and seller profile classes are only handling user and product functionality, respectively, for simplicity and maintaining organization

<u>QualityTag</u>

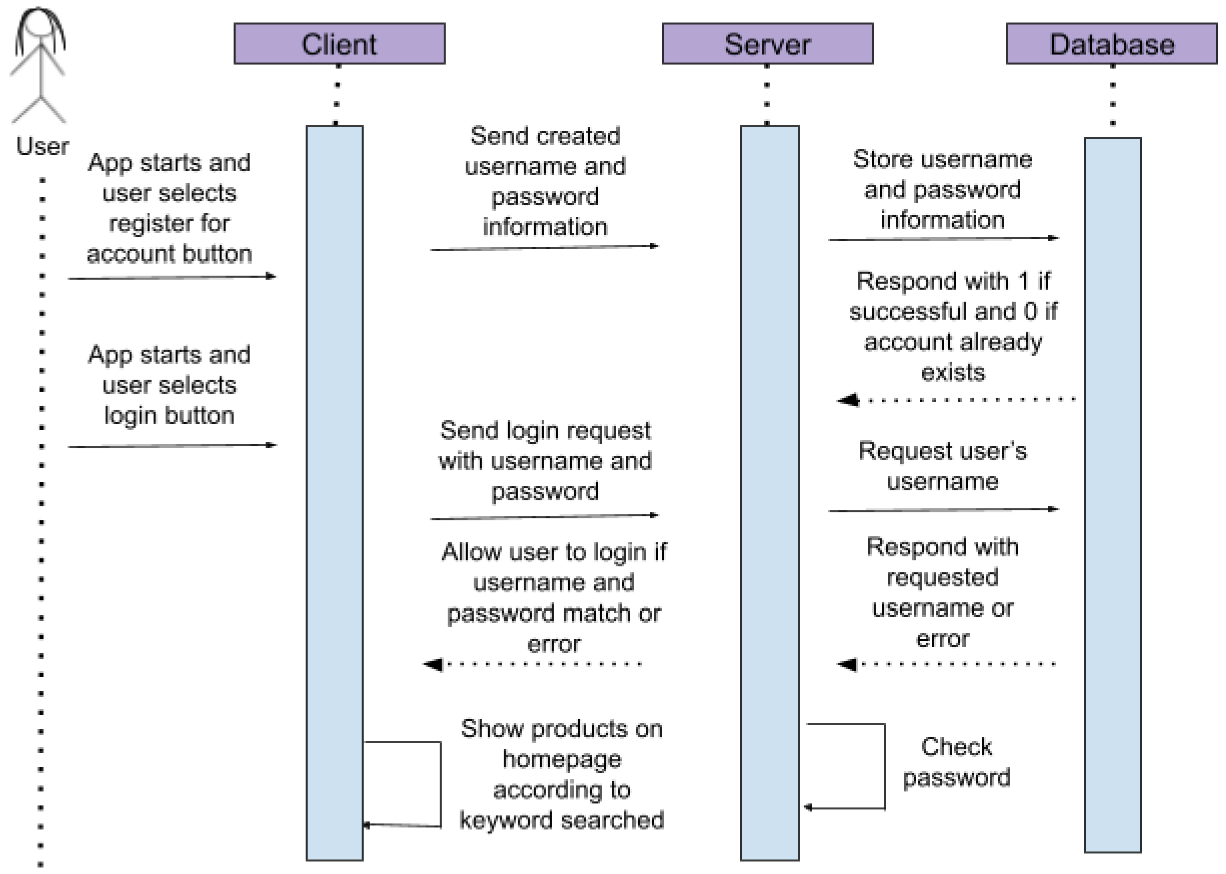
- The QualityTag object has information about the product's condition including if it is "Like New", "Good", and "Acceptable"
- The QualityTag is being implemented into the filter feature
    - Decided to make this a separate class so the code is more readable and the organization is easier to maintain

<u>CategoryTag</u>

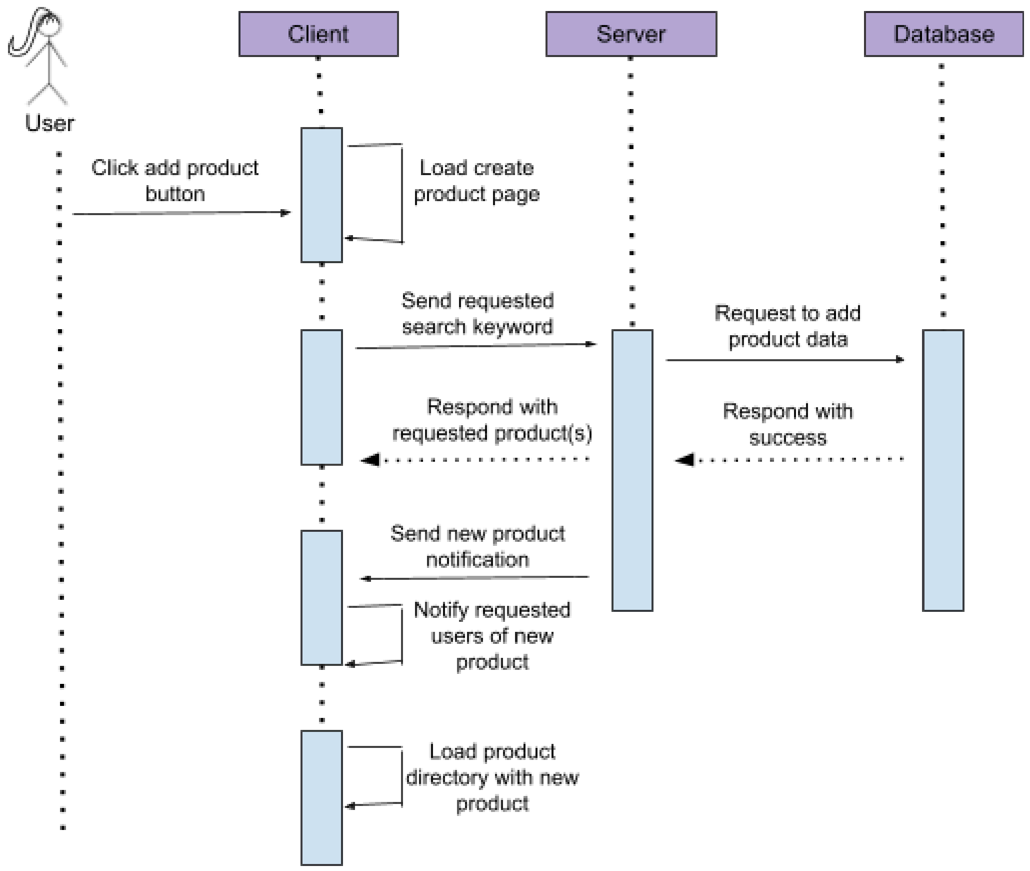
- The CategoryTag object has information about the product's category stating which category certain products should be under
    - This tag exists so users can filter by the different category items they want to view (i.e. "Furniture", "Lab Materials", etc.)
- Decided to make this a separate class so the code is more readable and the organization is easier to maintain since this tag is also being implemented into the filter feature

Sequence Diagrams:

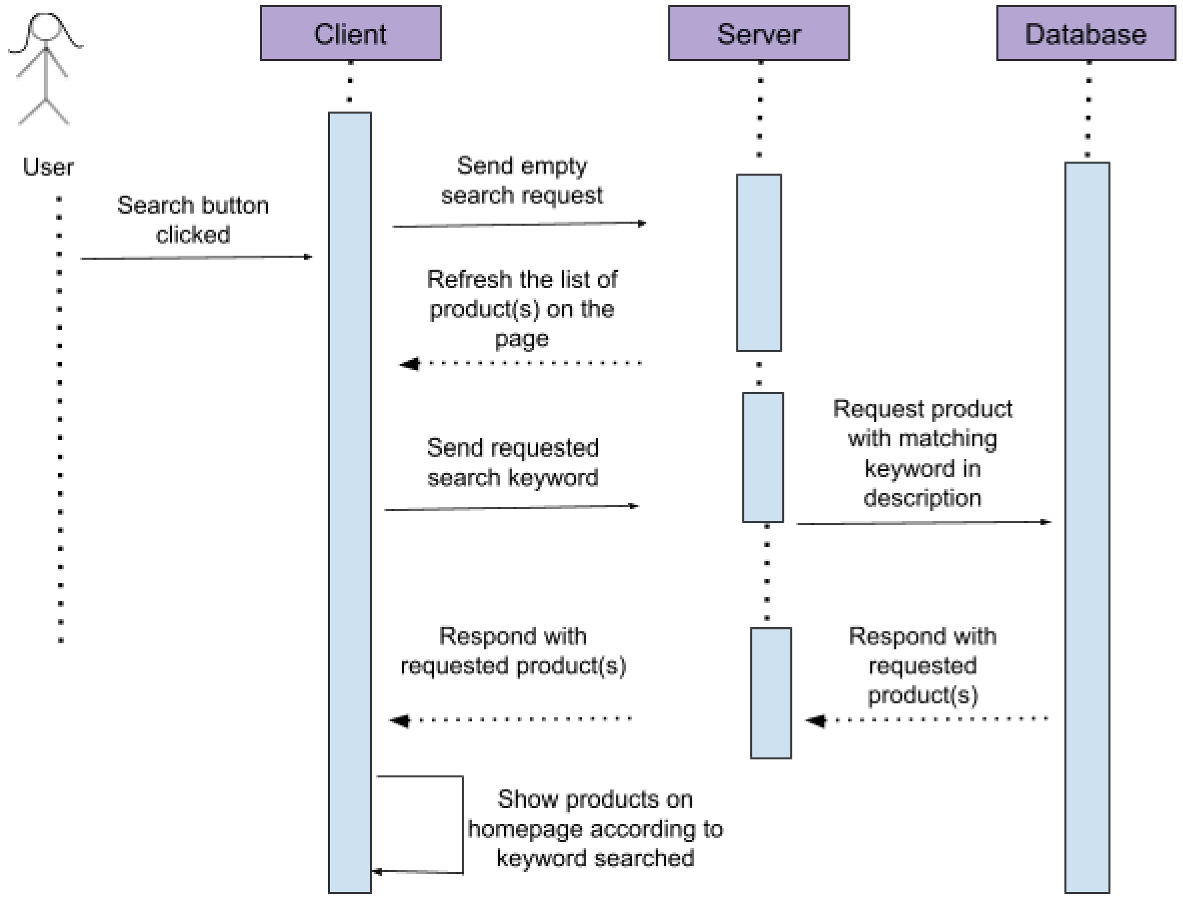
Sequence diagram when users login:


Client
Server
Database
User
App starts and user selects register for account button
Send created username and password information
Store username and password information
Respond with 1 if successful and 0 if account already exists
App starts and user selects login button
Send login request with username and password
Request user's username
Allow user to login if username and password match or error
Respond with requested username or error
Show products on homepage according to keyword searched
Check password

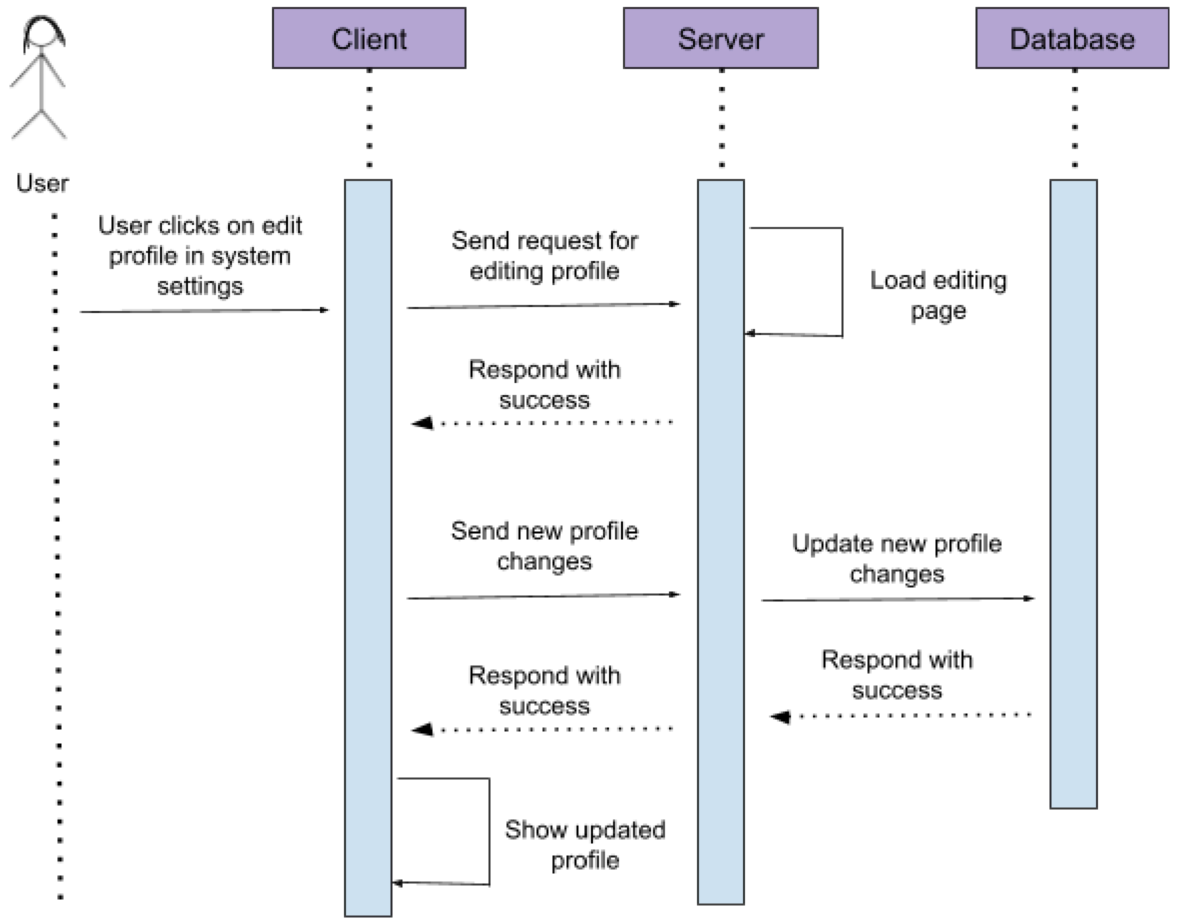
Sequence diagram when a user posts a product listing to the product directory:


User
Client
Server
Database
Click add product button
Load create product page
Send requested search keyword
Request to add product data
Respond with requested product(s)
Respond with success
Send new product notification
Notify requested users of new product
Load product directory with new product

Sequence diagram when a user searches for a product:

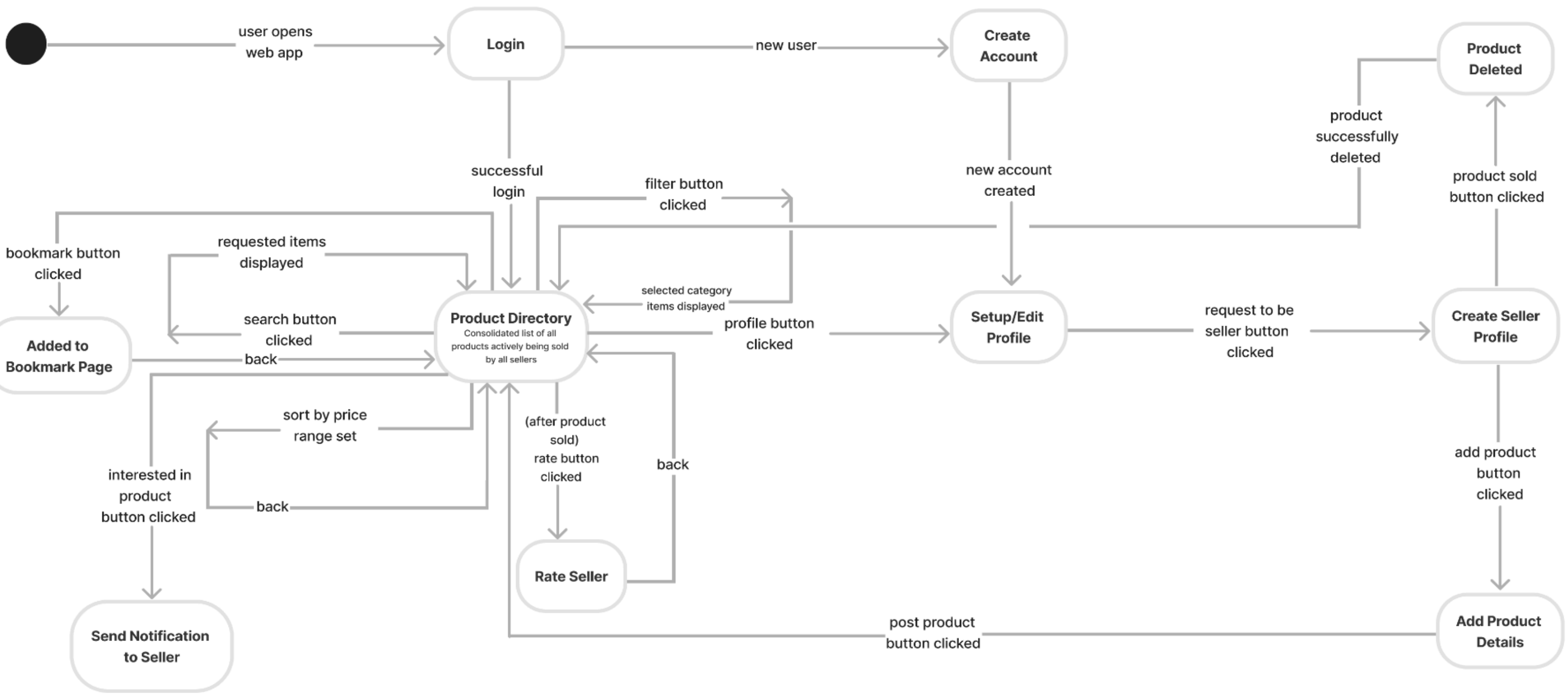

Client
Server
Database
User
Search button clicked
Send empty search request
Refresh the list of product(s) on the page
Send requested search keyword
Request product with matching keyword in description
Respond with requested product(s)
Respond with requested product(s)
Show products on homepage according to keyword searched

Sequence diagram when a user wants to edit their profile:

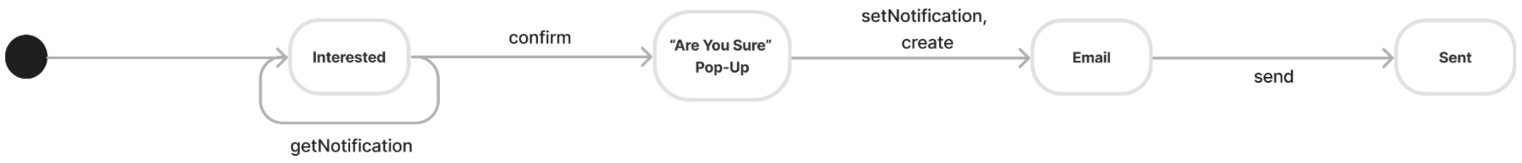


Activity/State Diagrams:

Overarching State Diagram

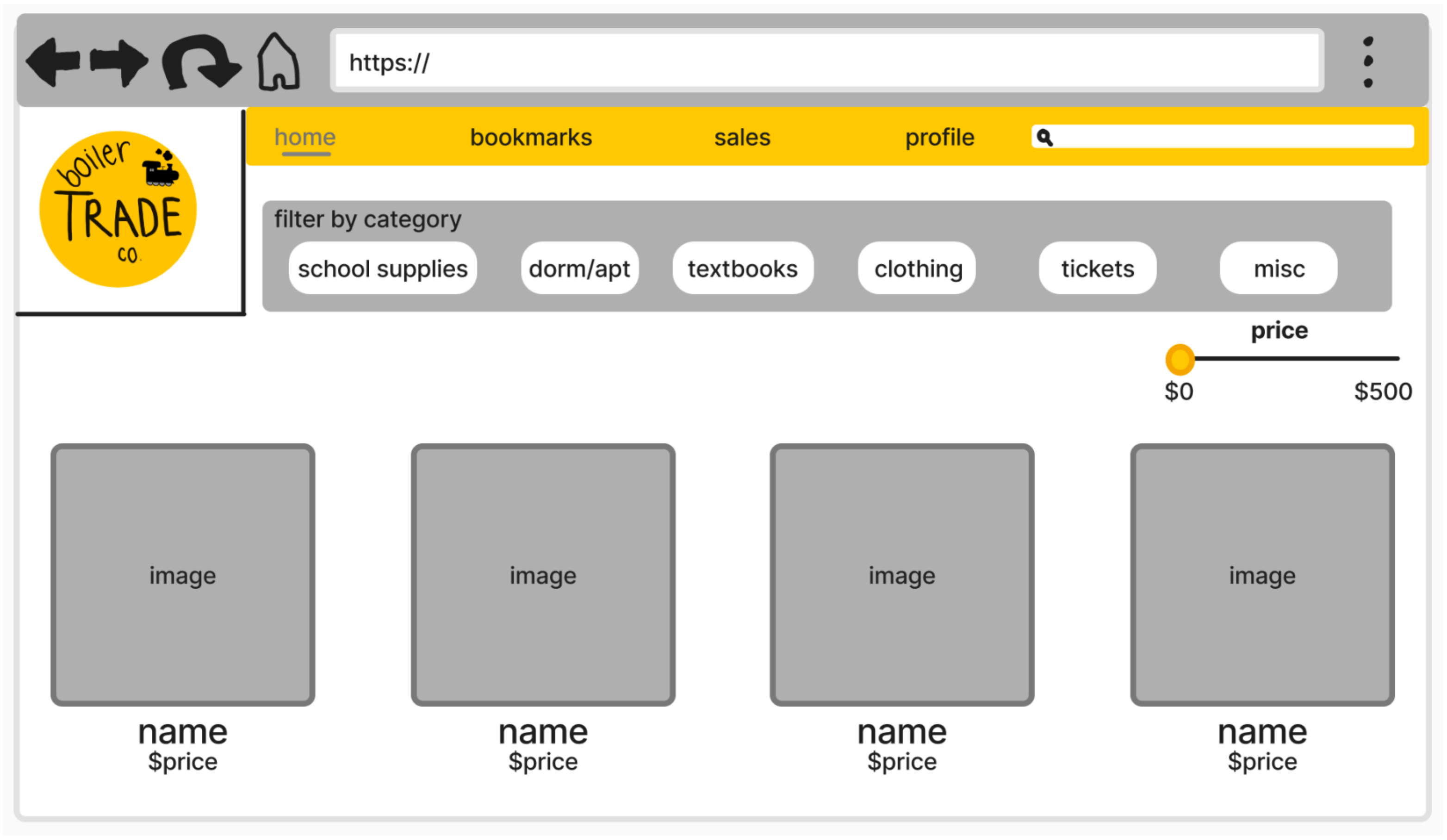

user opens web app
Login
new user
Create Account
Product Deleted
product successfully deleted
successful login
filter button clicked
new account created
product sold button clicked
bookmark button clicked
requested items displayed
selected category items displayed
Product Directory
Consolidated list of all products actively being sold by all sellers
search button clicked
profile button clicked
Setup/Edit Profile
request to be seller button clicked
Create Seller Profile
Added to Bookmark Page
back
sort by price range set
(after product sold) rate button clicked
back
add product button clicked
interested in product button clicked
back
Rate Seller
post product button clicked
Add Product Details
Send Notification to Seller

Notifications State Diagram (Notifying Seller of Buyer Interest)



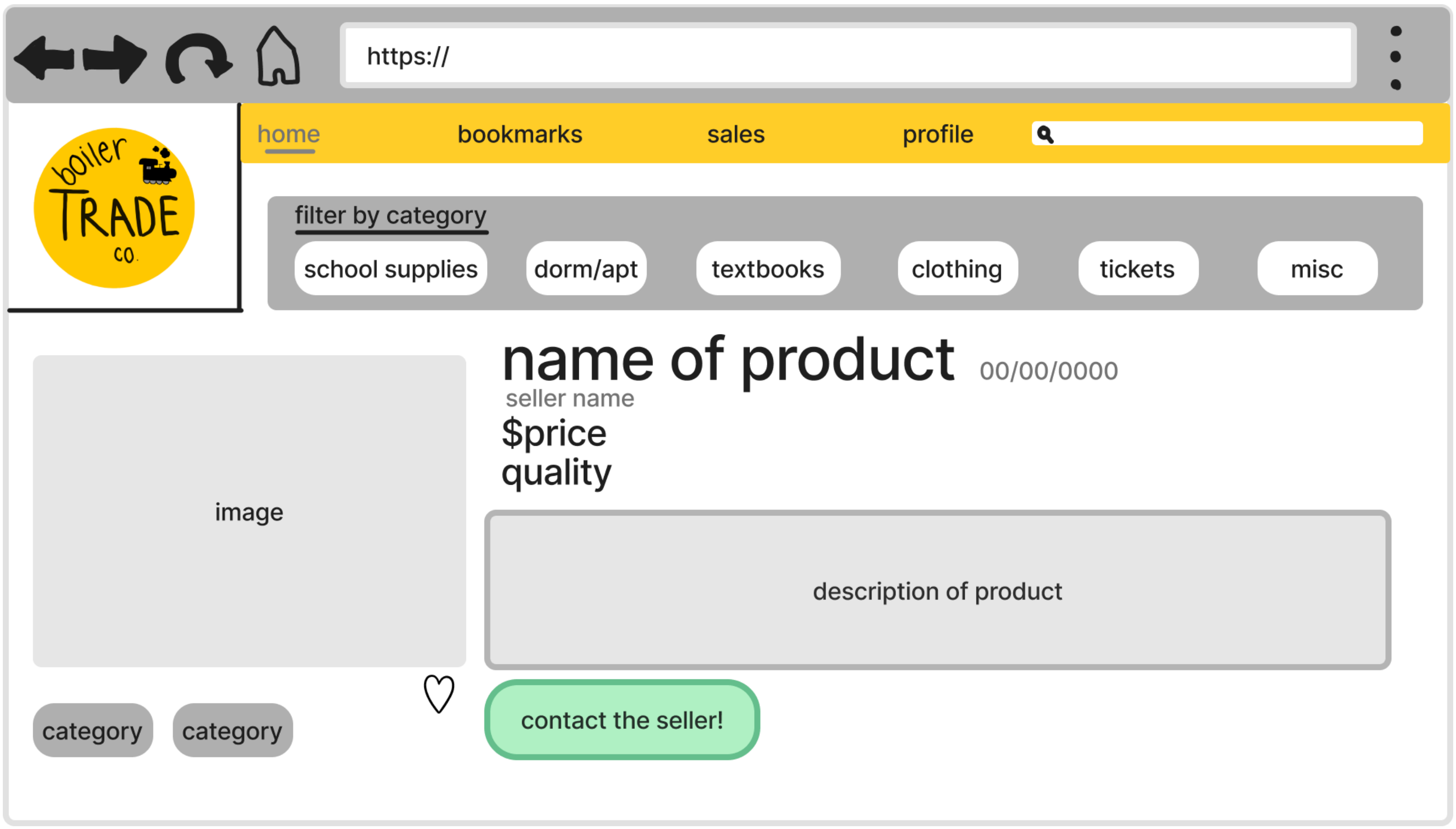
UI Mockups:

Home Page



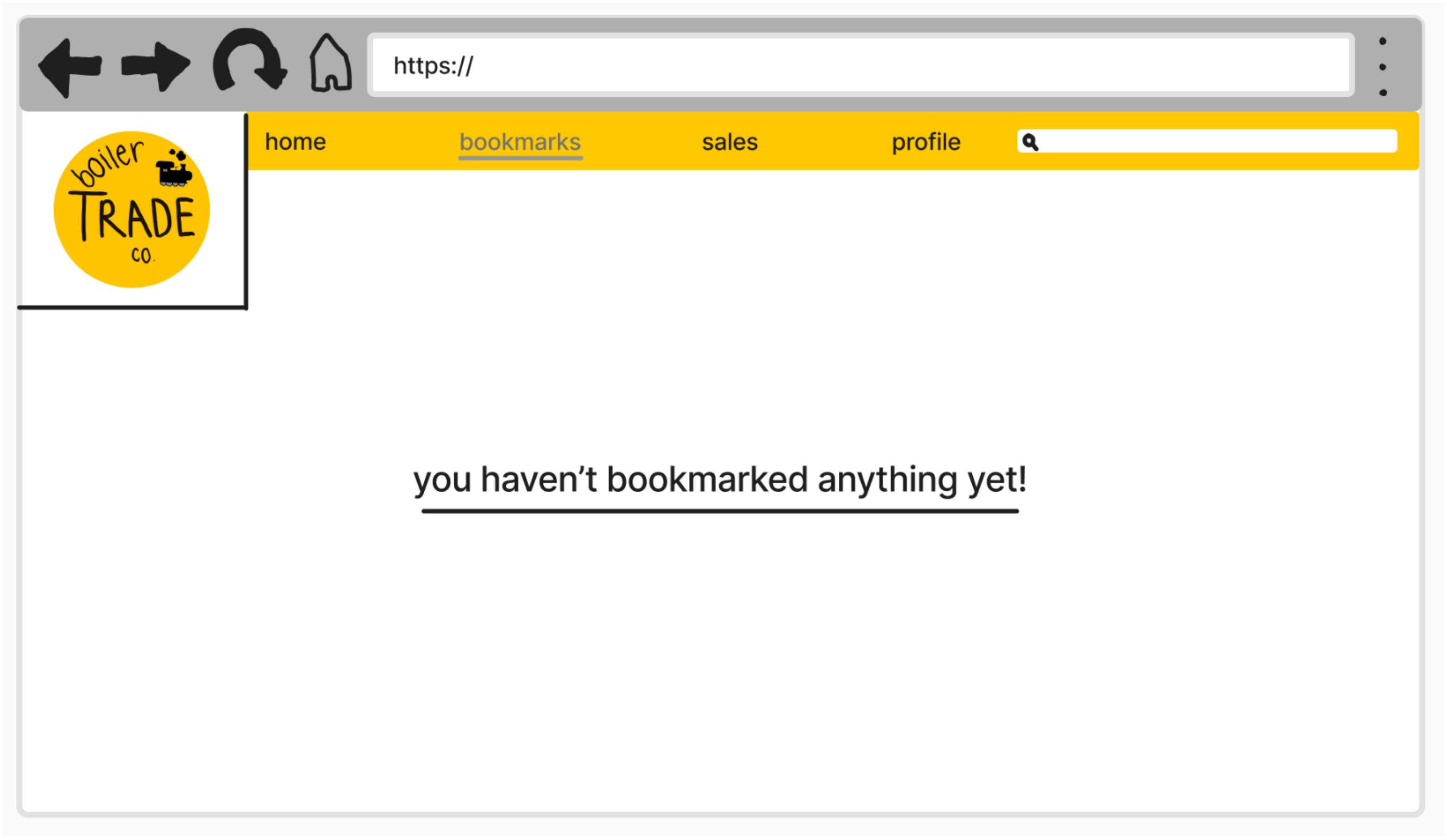
This is the homepage for BoilerTradeCo. We have our logo on the left side of our website and we are using a top navigation. Below our navigation, there is a filter option where a user can view products under a certain category. There is also a slide bar for filtering products given a certain price range. On our homepage, the products available in our product directory are listed in multiple rows with an image, name, and price of the product by date posted.

Product Listing



This is the screen when a user clicks on a specific product listing to view from the product directory. In addition to the product's name, price, condition, and image, users can also get more information about the product through the product description. When a user is interested in purchasing a specific product, they can click on the "contact the seller!" button, which will then notify the seller that a user is interested in their product.

Bookmarks Page



you haven't bookmarked anything yet!

When a user clicks on the "bookmarks" tab on the top navigation, they are led to this page. This page displays all the product listings that the user has favorited and added to their bookmarks. According to our mockup, there are currently no bookmarked products.