Final Report

COMP 3900 - W18A Let's Chat

—

Matthew Kurniawan Scrum master
z5163128@ad.unsw.edu.au

Daniel Gordos Developer
z5208992@ad.unsw.edu.au

Yip Jeremy Chung Lum Developer
z5098112@ad.unsw.edu.au

Well Son Tan Developer
z5183987@ad.unsw.edu.au

Submitted on April 23, 2021

Table of contents

Overview

Introduction

Petiverse is an Ecommerce platform that specialises in selling pets and pet products. It strives to keep the process of purchasing and selling pet products simple and accessible for all. Our novel features for this project were an integrated chatbot application that can help users interact with and navigate the site as well as an auction feature that allows users to bid on listings competitively.

System Architecture

Backend

Framework

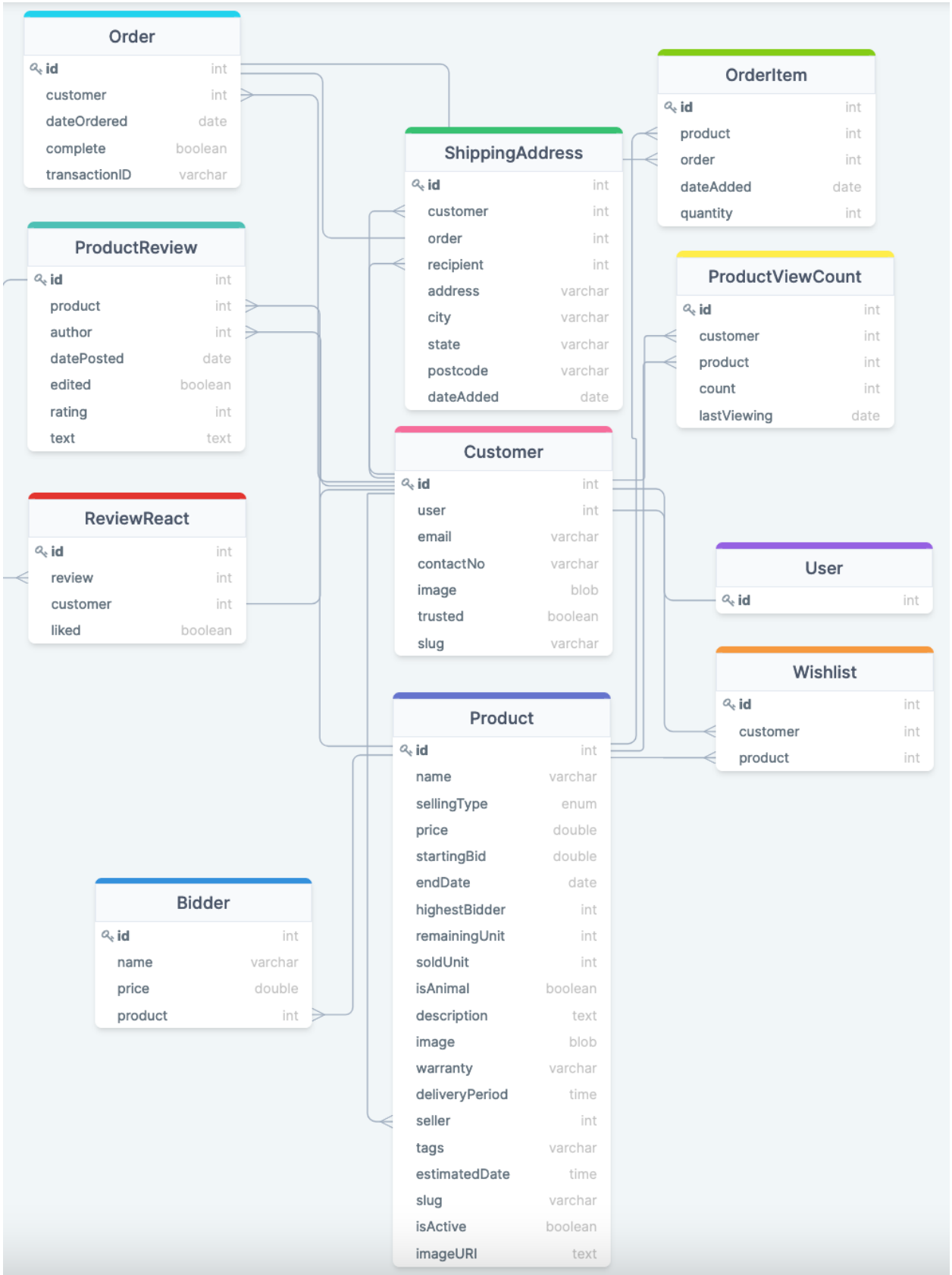
Petiverse's backend was written in Python 3 and was built on the open source Django Web Framework, which handles many aspects of its back end architecture. In particular, we utilise Django for:

- ❏ URL routing
- ❏ Session based user authentication
- ❏ Function based views, which handle server requests
- ❏ Backend HTML template rendering, using Django's template language
- ❏ Object-Relational database mapping (ORM)
- ❏ Admin / moderation functionality

Database

The system uses an embedded SQLite3 database to store user, product and sales data. Django includes an ORM system compatible SQLite3, allowing simple and safe manipulation of database records within Python code.

The following SQL diagram shows the system's database schema:


Order
id int
customer int
dateOrdered date
complete boolean
transactionID varchar
OrderItem
id int
product int
order int
dateAdded date
quantity int
ShippingAddress
id int
customer int
order int
recipient int
address varchar
city varchar
state varchar
postcode varchar
dateAdded date
ProductReview
id int
product int
author int
datePosted date
edited boolean
rating int
text text
ProductViewCount
id int
customer int
product int
count int
lastViewing date
Customer
id int
user int
email varchar
contactNo varchar
image blob
trusted boolean
slug varchar
ReviewReact
id int
review int
customer int
liked boolean
User
id int
Wishlist
id int
customer int
product int
Product
id int
name varchar
sellingType enum
price double
startingBid double
endDate date
highestBidder int
remainingUnit int
soldUnit int
isAnimal boolean
description text
image blob
warranty varchar
deliveryPeriod time
seller int
tags varchar
estimatedDate time
slug varchar
isActive boolean
imageURl text
Bidder
id int
name varchar
price double
product int

Note that the User table is left empty on the diagram as this is an authentication class provided by Django and was not implemented by us.

Messaging System

Petiverse utilises Pinax Messages, an open source messaging project for Django, to allow site users to communicate through message threads. Pinax handles the storing and retrieval of message threads from the system database, handles URL routing with respect to messaging functionality, and provides backend rendered HTML templates for the messaging interface.

Chatbot

Petiverse utilizes the Google DialogFlow service for handling chatbot queries. The natural language processing capabilities of DialogFlow agents is used to process user queries and extract the users 'intent. If the intent of a query is found then the agent advances through a dialog tree, allowing users to ask for help with tasks such as making a purchase, placing a bid, or searching for a product.

Frontend

Since Petiverse utilizes backend page rendering, frontend software consists only of elements that cannot be achieved via server rendering, such as dynamically updating UI elements in response to user interaction.

Requests

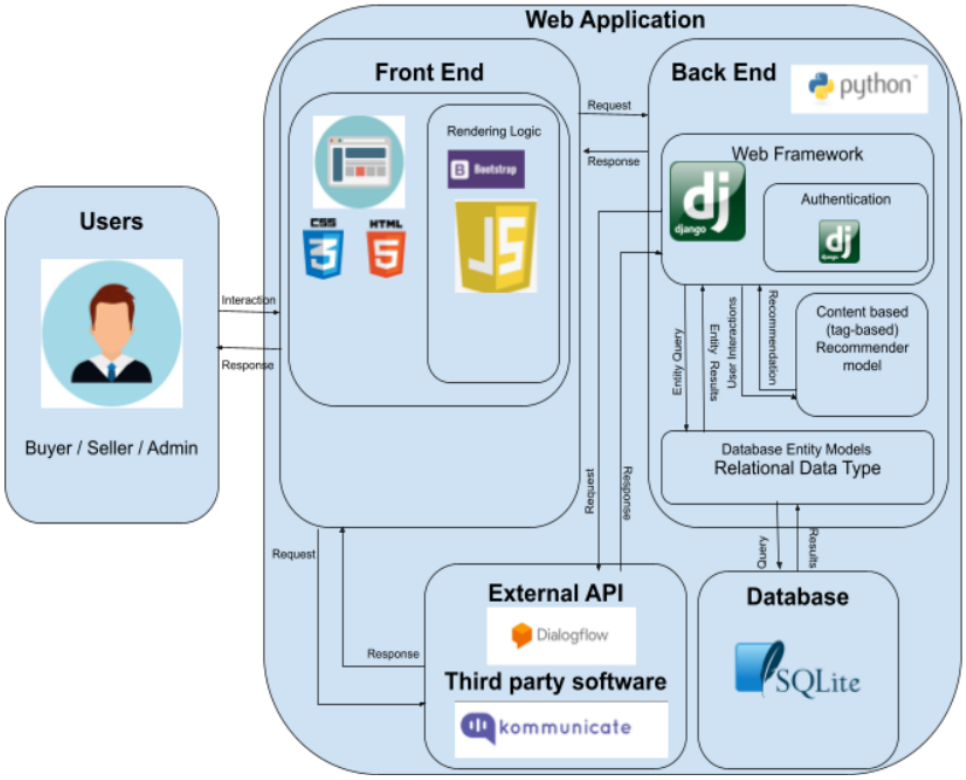
Jquery's AJAX request functionality is used to asynchronously post updates to the server in response to user page interactions such as liking product reviews and unlisting products.

Interface

The open source Bootstrap library is used for rendering page elements such as buttons, and dialogs, as well as for its collection of general purpose vector icons. These elements use lightweight, event-driven JS code to respond to user interaction.

Chatbot

Kommunicate, a proprietary chat support engine, is used as the interface for users to interact with the site chatbot.



The diagram above illustrates the system architecture described.

Functionality

Fulfillment of objectives / Implementation challenges

The following is the list of our original 10 project objectives (from the project proposal), with a detailed description of the site features that attempt to address these objectives.

1. *"Users are able to list products for sale with information such as a name, text description, images, tags and shipping information."*

Adding new listings

This objective was fully implemented by our system. Users have the ability to create new listings through a 'new listing' page accessible by all logged in users. This page contains a form for users to fill in the details of a new product, namely:

- ❏ Product name
- ❏ Text description
- ❏ Unit price
- ❏ Available units
- ❏ Warranty details
- ❏ Estimated delivery dispatch time
- ❏ Image
- ❏ Product tags - as a list of text strings

Once a user has successfully filled in and submitted the form, the product is added to the system database and the user is redirected to the store page of their new listing. It then can be reached by other users through the store recommendation system, through the 'similar products' scroll on store pages, through a search query, or by directly accessing the product page url (with the product's url slug).

2. *"Users can search for, and view the page of products that have been listed for sale. Searching can be done by text (name and description), or tags, and can be ordered by rating, popularity and date."*

Product Page

All products listed on Petiverse have a product page which is accessible through the path */product/<product slug>*, where *product slug* is a unique slug string generated from the product name whenever the product details are updated. The product page displays all details of the product relevant to a prospective buyer. In particular it shows:

- ❏ Product name
- ❏ Seller name (with link to their profile, and with a trusted badge if they have been marked as trusted by an administrator)
- ❏ Average review score and number of reviews on the product
- ❏ Text description
- ❏ Tags applied to the listing
- ❏ Unit price

The page also has an 'add to cart' button, with a quantity input box. This allows the user to purchase the product by adding it to their cart, and then checking out that cart through the purchase interface. Clicking 'add to cart' when the user has set an invalid quantity (eg. when they have selected more units than the product has in stock) will not update their cart.

Further down the product page is a horizontal scroll of 'similar listings' in a card format. This displays a small number of products that have tags in common with the current product, since they may be of interest to the current user. This was implemented using the Django-Taggable library, which handles all product tags.

Finally, the page contains an input form for users to submit a product review, by entering a text description and choosing a star rating. This is only visible if the user is logged in and not the seller of the product. The existing reviews for the product are shown under this element in a list format, with the review text, star rating, review likes, review date and seller name displayed. Reviews can be reordered through a dropdown by the following criteria:

- ❏ Most recent
- ❏ Most liked
- ❏ Highest review score
- ❏ Lowest criteria

The dynamic sorting of reviews was implemented using frontend Jquery.

Search

Petiverse's search system can be accessed from the search bar at the top of each page, or directly through the search results page, once an initial search has been made. An entered query is matched against the name, seller name, and tags of each product in the database. Any matching results are returned in a card layout similar to the main store page.

Once on the search results page, users can reorder the results alphabetically, by highest rating, or by price (ascending or descending), accessible through a dropdown menu. Similarly to sorting product reviews, the dynamic sorting was implemented with Jquery.

Advanced search filtering options can be accessed through a toggle button on the search page. This allows users to filter by:

- ❏ Listing type (regular sales or auction)
- ❏ Whether the listing is for a live animal
- ❏ Price range - by entering a minimum and maximum price

The next search search query submitted will then apply these filters.

3. *"Users can add products to a cart, which can have its contents edited and then be checked out to complete a purchase"*

Adding cart items

Store cart functionality was successfully implemented and works as expected for an Ecommerce platform. Every logged in user has a cart, stored on the system backend, which tracks a set of items that they are interested in purchasing. Each cart item also has an associated quantity, for purchasing more than a single unit at a time.

The most common way to add a product to a user's cart is through the 'add to cart' button on product pages. The associated quantity input can be used to change the desired number of units - otherwise it defaults to zero. When a user adds products to their cart, the page is reloaded and the badge next to the cart icon on the top navbar is updated to reflect the number of products currently in their cart.

Users can also add products to their cart through the inline 'add to cart' button on the product cards that are present in several parts of the site. These cards allow users to add products to their cart directly from:

- ❏ The main store page
- ❏ The similar products scroll on product pages
- ❏ Search results

Cart

Logged in users can access their cart through the path */cart*, or by clicking the cart icon on the navbar. The cart page shows each product in the user's cart, with the associated quantity and cost. The total cost of the cart is also shown. Buttons next to each cart item allow users to quickly change the quantity of an item. Reducing the quantity to zero removes the item from the cart entirely. By clicking the 'checkout' button, the user is redirected to the checkout page.

Checkout page

This page allows users to finalize a purchase by entering shipping details. Since we are using a dummy purchase / payment system, no actual payment is required, and the shipping address fields are not validated or used. By clicking continue, the user is prompted to choose a payment type. This is also a dummy / unimplemented feature.

Once a payment type has been chosen a pop notifies the user that the purchase has been completed. The user, and all of the product sellers are then notified of the transaction by email and the user is redirected to the home page.

A completed purchase results in the following database updates:

- ❏ The user's cart is emptied.
- ❏ The order is marked as 'complete', and thus its items can now be viewed by the user from the purchase history page. Additionally the relevant sellers can now view the order on their product from the relevant products 'view orders' page.
- ❏ Each purchased product has its remaining unit attribute reduced by the purchased amount. Products that have no units remaining are unlisted

4. *"Site users have a dashboard page, which displays recently viewed products, manually curated categories, and automatically recommended products to the user in a visually appealing layout."*

Store page / Dashboard

The majority of dashboard page functionality was implemented. The manual curation feature was cut due to a lack of time by the second sprint.

The main store page / dashboard page displays cards of a user's 10 most recently viewed products in a horizontal scroll element.

Under the recently viewed section, products recommended to the user are displayed in a grid layout. The recommended products are paginated, using Django's builtin pagination functionality, to reduce loading times and web page size. The recommended products are chosen and ordered by the recommender system, described in the next section

5. *"The dashboard utilizes a recommender system, which uses the product tags on previously purchased and viewed products to automatically show relevant listings to the user."*

Recommendation system

The recommender system takes a user, and returns an ordered database queryset of recommended products. The system uses two metrics to generate a recommendation score for each product:

1. Product tag similarity to user
2. Review score

For product similarity, the number of occurrences of tags on products the user has viewed are counted. The tags on products the user has purchased are similarly counted, but are weighted more heavily. By default, 1 purchase is equivalent to 2 pageviews, although this weighting can be easily adjusted. These counts form a 'profile vector' of the user, describing the attributes of products they may be interested in.

The cosine similarity metric, defined as:

$$\frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}},$$

where A, B are vectors, is used to generate a similarity score between the user's profile vector and the tags on each active product listing on the site.

The review score of a product is simply the average of the star ratings given to the product in reviews, normalised to between 0, 1.

Finally these two metrics are combined in a weighted average to form a product's final recommendation score. The review score is weighted more heavily when there are more reviews present on a product, up to a maximum possible fraction of the score (0.3 by default). Thus, the final recommendation score can be expressed as:

$$s(1-w)+rw,$$

where **s** is the similarity score, **r** is review score, and **w** is the review weighting such that:

$$w = maxRatingWeight \times \left(\frac{n}{maxReviews}\right).$$

maxReviews is a set number of reviews which when reached, results in the maximum rating weight, while **n** is the number of reviews on the product clamped between 0 and **maxReviews**.

If a user is not logged in (or hasn't viewed or purchased any products), their similarity scores cannot be determined and thus their recommendations are based entirely on review scores.

Finally, a queryset of the site's products is sorted by similarity score and sliced to return a fixed maximum number of results (100 by default).

6. *"A registration and login system allows users to retain their listings, payment details, purchase history, and personalized product recommendations. Unregistered users are able to search and view listings, but aren't able to buy or list products."*

Registration and login were implemented using the Django framework's session based authentication system.

Registration

Users are able to register a new account through the signup form at the path */signup*. This form prompts users to enter an email address, (unique) username, and a password (whose quality is evaluated by Django's signup system). Successfully registering a new account redirects the user to a success page which prompts them to log in with their new account. A welcome email is also sent to the new user. The signup page can also be reached via a button on the top navbar, or a link on the page footer when the user is not logged in.

Login

Users can log in to an existing account through the login form at the path */login*, which is also reachable through a button on the top navbar , and via a link on the page footer when not logged in. Guest users are automatically redirected to the login form when attempting to perform many actions that require authentication, such as accessing the 'manage listings page', the 'new listing' page, or viewing purchase history.

The login form prompts users to enter their username and password, and if successfully validated, redirects them to the main store page and creates a session for the user on the Django backend. All requests performed by authenticated users must contain a CSRF token in their body in order to be accepted by Django's view system.

Users can also reset their password through a form accessible from a link on the login form. This link redirects users to a page that prompts them to enter an email address, to which reset instruction will be sent. If a user matching the given email is found, an email containing a link to a reset form for their account will be sent. On this form users are prompted to enter a new valid email, which if accepted, updates the user's stored password.

7. *"Users who have purchased a product are able to leave reviews composed of a star rating and a text review on the product's page. These reviews can be viewed and filtered by*

other users viewing the product page. Users selling a product are able to reply to reviews on their product."

The majority of review functionality described in the proposal was implemented. The ability for users to reply to existing reviews was not implemented due to lack of time, and of having limited importance to the goals of the project.

Adding reviews

Logged in users are able to leave a review on a product through a form on the product's page. If the user is the product's seller the form will not be rendered, and if the user is not logged in then it will have no effect.

The review form requires users to enter a text review (of length 1-1000 characters), and select a star rating from 1-5 (implemented by radio buttons styled as stars). The form's 'review submit' button verifies that the necessary details were filled out and sends the POST request for the new review using AJAX. If the review submitted is invalid (rating not selected or text is of wrong length), alert messages above the review form notify the user of the actions they need to take to successfully submit. Once a review is successfully submitted the page is reloaded to display the new review.

Modifying reviews

Once a user has posted a review on a product, the review form is replaced by a card showing the user's review. The card contains a delete button, which when pressed removes the review from the site database and reloads the page, and an edit button. The edit button replaces the review card with a form allowing the user to modify the review's text and star rating, and save the changes, reloading the page again. Reviews that have been modified are marked as such on the backend.

Viewing reviews

As described underneath objective 2, underneath the review form is a list layout of existing reviews on the product. These reviews can be sorted through a dropdown by posting date, review score and by like count.

The Petiverse's review system implements a like / dislike system which allows users to vote on the usefulness of other's reviews. Each review's card displays the review's like score, which is simply the sum of all dislikes on the review subtracted from the sum of all likes. For example, a review with no likes or dislikes will have a score of 0, while a review with 3 likes and 1 dislike will have a score of 2. When users post a new review it is automatically

liked by them. When a user clicks either the like or dislike button on a review card, an AJAX request is sent to update the state of the user's reaction to the review. The server response contains the new score and like status of the review, allowing the review's score to update dynamically on the page.

8. *"Admin users are able to remove product listings and reviews, as well as set some default categories displayed on user dashboards."*

Admin moderation

Django allows users to be elevated to having admin status, either by other admins, or directly through the backend server command line interface. Admin users are able to access an 'admin site' reachable through the path */admin*. This site allows admins to directly modify database tables through a simple GUI. Thus, site admins are able to remove or modify listings, reviews and user accounts that are deemed not to comply with the site's terms of service.

Admin curation

The functionality allowing admins to set default display categories on the store page was not implemented, due to it causing unnecessary clutter on the main store page. The scroll of recently viewed products, followed by any products recommended to the user allows the page to have a much visually cleaner and more intuitive layout. Additionally, users may have been able to exploit a system utilizing curated categories by editing their product tags to fit all of the currently curated categories even if they are irrelevant to the category, thus greatly increasing the likelihood of their product being seen. The chosen recommender system reduces this problem as excessively tagged products inherently have their score 'diluted' as they are less likely to have a high similarity to any user's profile.

9. *"Sellers are able to list a product as an auction rather than regular listing. When starting an auction, the seller can specify the starting bid, minimum increment, and autobuy price."*

Most auction functionality was implemented as expressed in the proposal. When users are creating a new product listing, they have the option to choose between a regular sales listing, and an auction listing. The form then allows the user to set a starting bid price, and an end date for the auction. When created, the auction listing displays its current bid price, rather than a regular price, and gives users the option to add a bid on the listing, rather than purchase units.

When a user submits a bid from the listing's product page, the current bid is updated and any future bids less than or equal to the current bid are rejected. Additionally, when a user successfully bids on a listing, they are added to a list of bidders on the product. Recent bidders and bid amounts can be accessed through a button displaying the total number of bids on the listing, on its product page. This button opens a modal window listing recent bidders and bid amounts. The page for an auction product also displays the time remaining before the auction ends through a dynamic timer widget.

When an auction reaches its end date, it is automatically marked as complete by a background server thread that periodically checks the status of active auctions. The auction is unlisted (so it is not visible to future users), and the product is automatically added to the cart of the winning bidder. An email is sent notifying both the winner and the seller of the auction outcome.

10. *"A chatbot can be utilized by users to easily make purchases, listings, search, ask for product recommendations, and ask for general help."*

Chatbot

Petiverse implements a chatbot that can handle general questions from users such as "How to make a listing?", "I want to raise a complaint.", "I want to know warranty of ***." etc. More advanced users are able to ask for product recommendations and request assistance in participating in auctions. All the features are processed through DialogFlow to detect the user intention and match with stored entities. The matched entities are parsed to the website database to retrieve the wanted data and packaged in a fulfilment, after which they are finally sent back to DialogFlow as a JsonResponse. DialogFlow passes the fulfilment to Kommunicate which renders it through its user chat interface.

Selling help

Chatbot will reply with a complete procedure on creating a listing and the further on features for product management.

Product enquiry

Product enquiry feature allows users to ask the information they want to know about a product such as warranty, delivery period and description.

Product searching

DialogFlow will detect intent and check whether the keyword has matched any of the existing entities and webhook to the website backend and find all products that have the tag that matched with the keyword. After finding the products, return them in fulfilmentMessages as a JsonResponse and the payload format will be differentiated by Kommunicate and become an Actionable Message of Kommunicate then render out as a list format on the chatbot.

Complaint helper

During the complaint process, chatbot will ask for two values which are reason and order ID. The reason value will be parsed to DialogFlow agent to match with the existing entity called complaint_reason that is stored in the agent and order ID value needs to be an integer value. If any of them can not be matched the complaint process will not be achieved. Otherwise, chatbot will reply with a success message that tells users their requests have been received and awaiting to be handled.

Place bid

To successfully place a bid through chatbot will require three key values which are the correct name of an auction product, a price that is higher than the current highest bid and an username that excluded the seller of the product. Different correction responses will be replied to when any invalid parameter has been detected. If all values obtained from the user were correct then chatbot will answer with the success message.

Third party functionality & Impacts of licensing terms

DialogFlow ES

Google has provided a list of free APIs for Google users to utilise and DialogFlow ES is one of them. DialogFlow ES means to be a starter kit for beginner and experienced web developers to build their own chatbot so they are the powerful and friendly platform for anyone to get familiar with the chatbot architecture. Our team uses a free edition to build our interactive chatbot by only having basic knowledge about the chatbot system.

We use DialogFlow webhook functionality to perform natural language processing (NLP), detect user intention and integrate with our system database which allow our DialogFlow chatbot agent (Petiverse) to retrieve the user requested data and parse it to the third party chatbot support application (Kommunicate) to render different type of fulfilment payloads.

While in the setting of this project, Google's API licensing system does not have any major impact on the project. However, in a real world scenario, Google's APIs licensing term would mean that Google has the right of non-exclusivity meaning that they can develop a similar product to our platform. Google also expects its clients to implement a commercially reasonable effort to protect user information collected by our platform.

Kommunicate

Kommunicate provided us with different types of frontend UI like normal conversation chat bubbles, list of recommended products, rating system for the conversation etc. It also allowed us to review, resolve and reopen the conversation through their website kommunicate.io. By using the tools they provided, we can easily handle each conversation as we desired. Kommunicate allows a new user to have a free trial of 30 days which gives us sufficient time to implement and test our chatbot for the project. Meanwhile, the 30 days trial also constrained our usage time and we had to recreate a new account to utilise their services in a long run.

Pinax

Pinax is an open source project released under the MIT license. It contains open source projects mostly created using the Django framework. Which was perfect for our project considering that our platform was also heavily reliant on Django. One of the open source projects inside pinax is a messaging system named *pinax-messages.* It is a user to user messaging system that allows users to send messages with a subject heading and a content field to others. It also has a simple inbox that displays all incoming messages from other users. By using this, we cut down on a lot of developing time and prevented "reinventing" a message system. Since the project was also released under the MIT license, any modifications to the code can be released with no further confirmation to the creators. It also allows the code to be commercially released without any risk of copyright infringement.

Installation

Prerequisites

- ❏ Windows 10 or MacOS Big Sur - Other operating systems are untested
- ❏ Google Chrome or Safari Browser - Some functionality may not work correctly with Firefox
- ❏ Git - https://git-scm.com/downloads
- ❏ Python 3 - https://www.python.org/downloads/
- ❏ Pip 3 - https://pip.pypa.io/en/stable/installing/
- ❏ Ngrok - https://ngrok.com/

Downloading project and packages

1. Clone the project repository into the desired directory with the command:

   ```
   git clone
   git@github.com:unsw-cse-comp3900-9900-21T1/capstone-project-3900-w18a-let-s-chat.git
   ```

2. Next you must install the project's required packages. Navigate to the project's root directory and run the command:

   ```
   pip3 install -r requirements.txt
   ```

   This should install the python packages required by the system, such as Django, Pillow and Pinax messages.

Chatbot Setup

Our team has set "ali.darejeh@unsw.edu.au" as one of the developers for Petiverse's DialogFlow agent. Please contact him or wellsontan@hotmail.co.uk to grant permission for the further action.

The chatbot with all the working functionalities will require a public url generator, the corresponding DialogFlow agent (Petiverse) and Kommunicate web script. Currently, the github code has included a working chatbot which has a valid Kommunicate appID that can last for 30 days (expires in 21 May 2021). If the chatbot is missing from the web pages that could mean the appID is no longer working, please refer to "Kommunicate free trial expired do this" section. Or else just follow "Get a public url to enable DialogFlow webhook functionality" to start the chatbot.

Get a public url to enable DialogFlow webhook functionality

Download

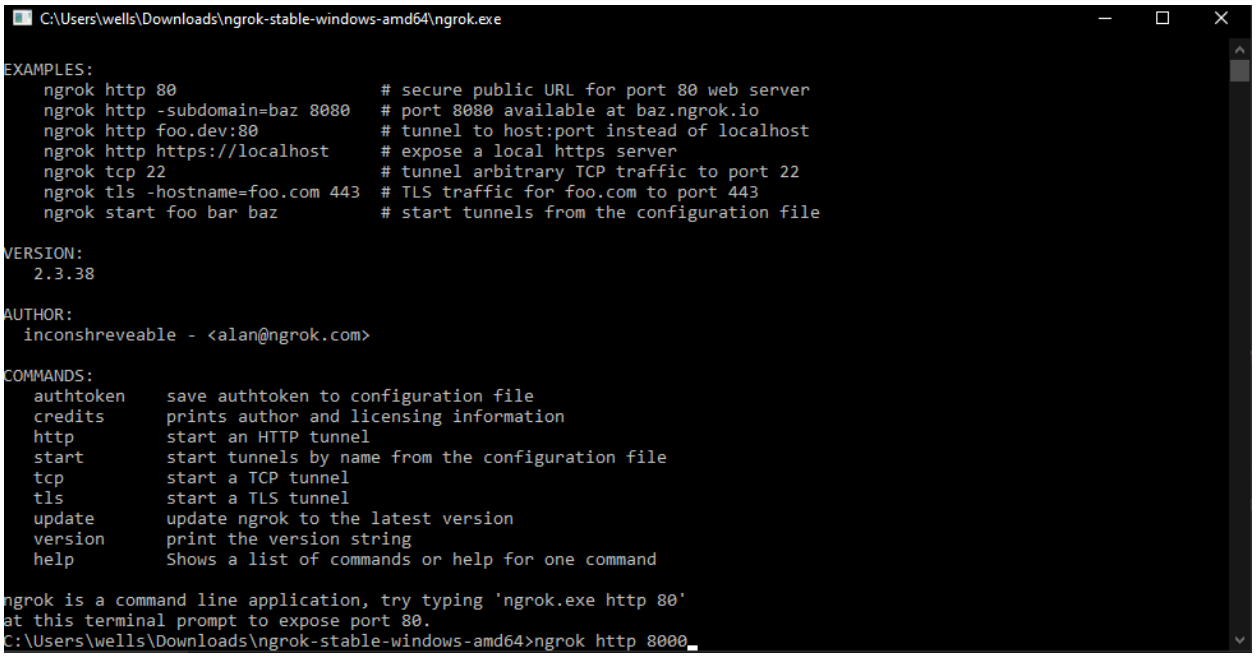
Please download ngrok and generate a temporary public url (2 hours long):

https://ngrok.com/

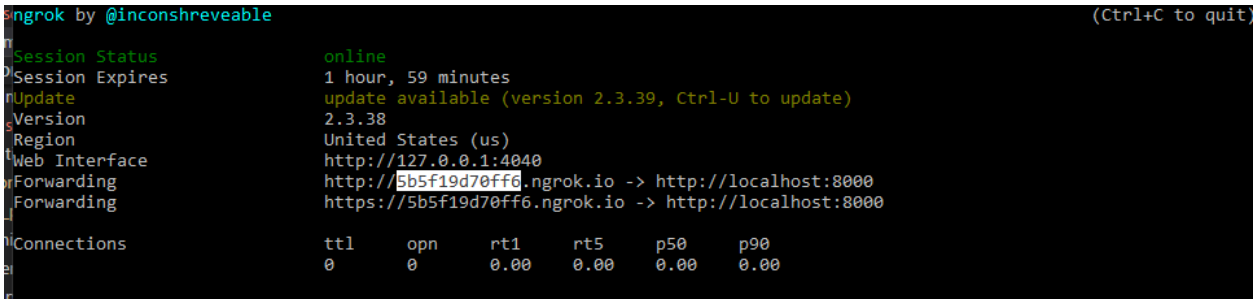
After downloading the zipped file, please 'Extract all' and run the ngrok.exe file and the ngrok command prompt will pop up.

Replacement

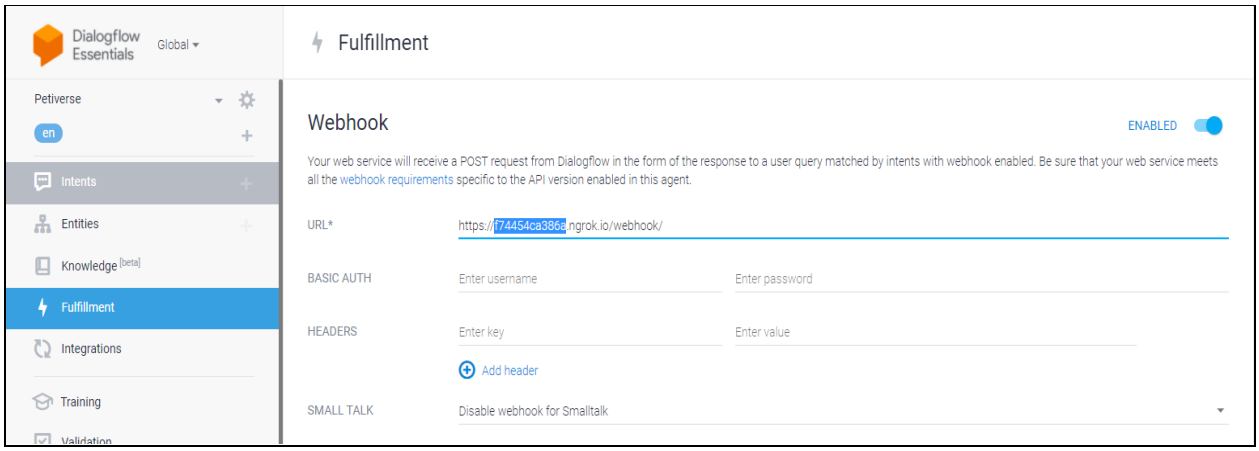
Please enter 'ngrok http 8000' in the command prompt to trigger the public url generation:

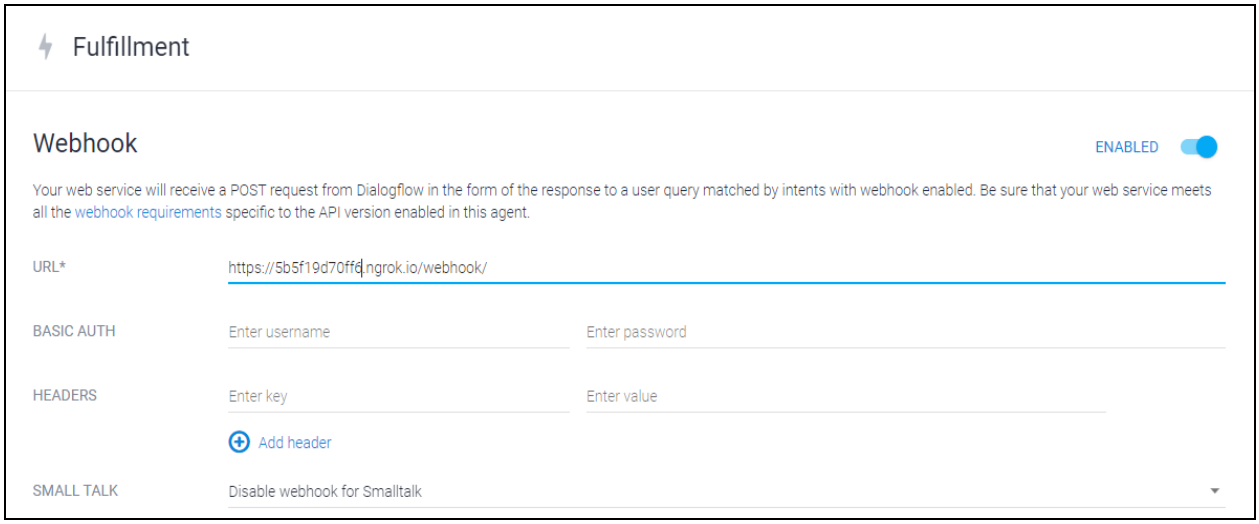


After the public url has been generated, copy the highlighted part as shown below:

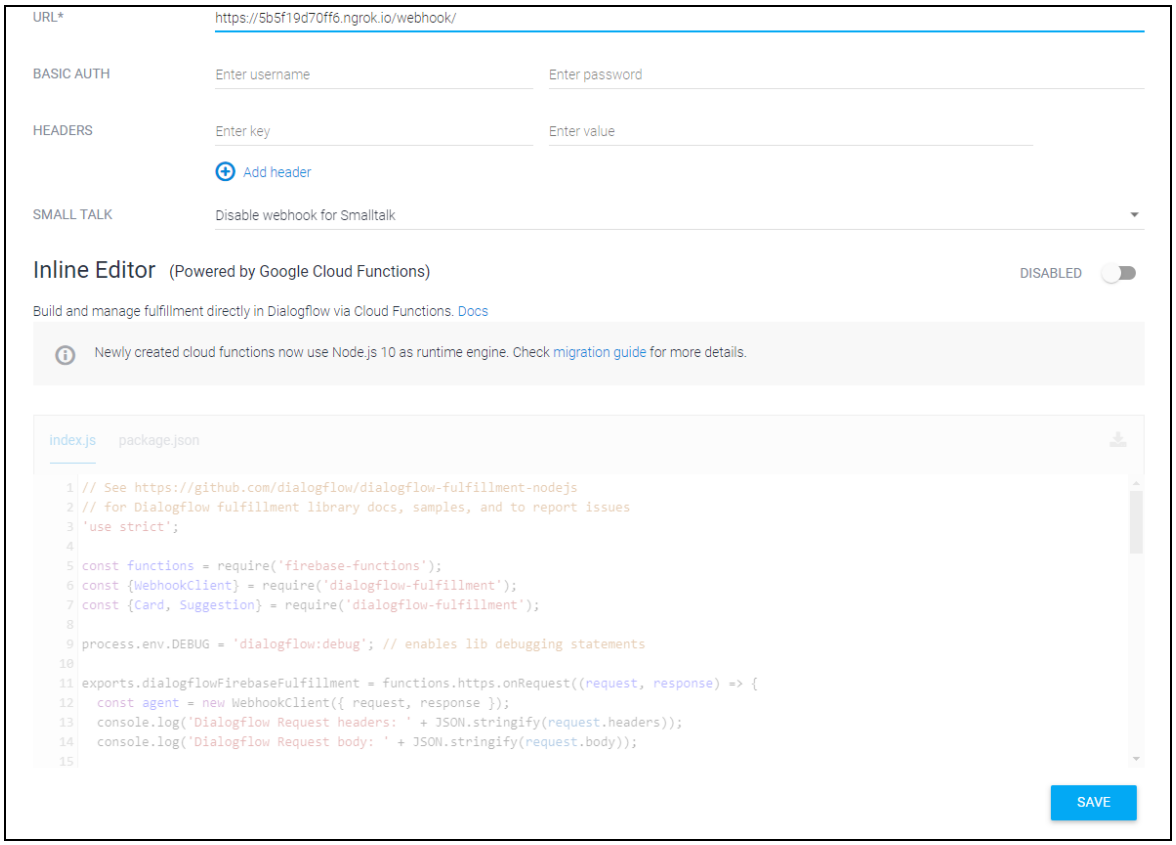


Now, go to the DialogFlow fulfilment section and replace the highlighted part with the copied texts and save it. Then, 'Save' button will become 'Done'. The complete processes will like the following screenshots:



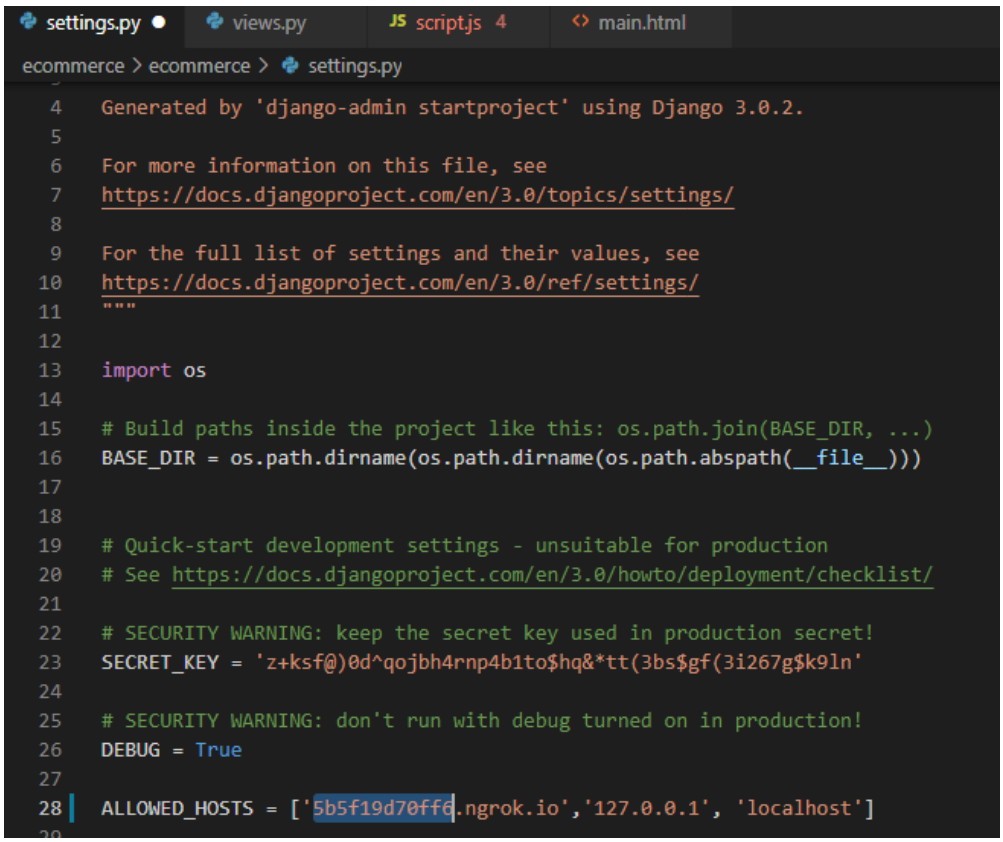


Note: Always include '/webhook/' in the URL section. Do not remove that. Also the webhook requires a https url.

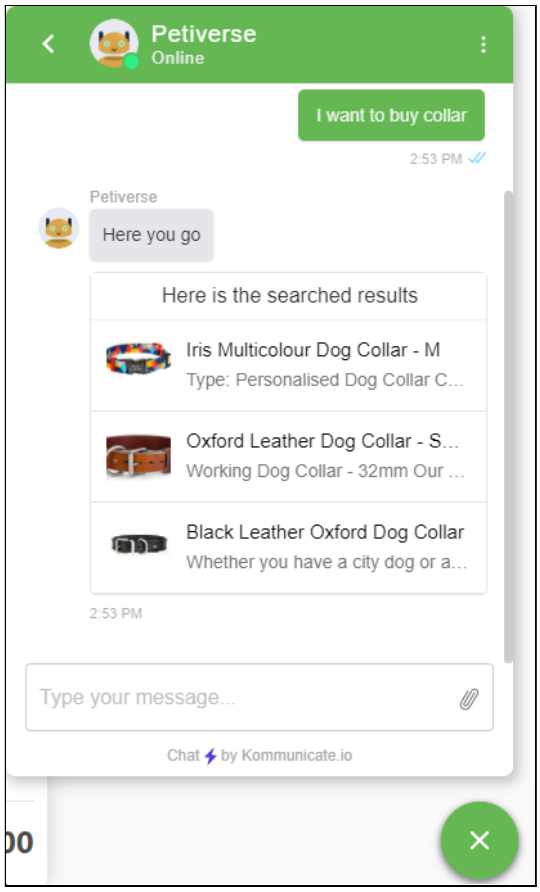

URL*
https://5b5f19d70ff6.ngrok.io/webhook/
BASIC AUTH
Enter username
Enter password
HEADERS
Enter key
Enter value
Add header
SMALL TALK
Disable webhook for Smalltalk
Inline Editor (Powered by Google Cloud Functions)
DISABLED
Build and manage fulfillment directly in Dialogflow via Cloud Functions. Docs
Newly created cloud functions now use Node.js 10 as runtime engine. Check migration guide for more details.
index.js
package.json
// See https://github.com/dialogflow/dialogflow-fulfillment-nodejs
// for Dialogflow fulfillment library docs, samples, and to report issues
'use strict';
const functions = require('firebase-functions');
const {WebhookClient} = require('dialogflow-fulfillment');
const {Card, Suggestion} = require('dialogflow-fulfillment');
process.env.DEBUG = 'dialogflow:debug'; // enables lib debugging statements
exports.dialogflowFirebaseFulfillment = functions.https.onRequest((request, response) => {
const agent = new WebhookClient({ request, response });
console.log('Dialogflow Request headers: ' + JSON.stringify(request.headers));
console.log('Dialogflow Request body: ' + JSON.stringify(request.body));
SAVE

DONE

After changing the DialogFlow fulfilment part, return to ecommerce/ecommerce/settings.py and replace the expired url in ALLOWED_HOSTS with the copied text and save as below:



When the server is run, if the chatbot is functioning correctly, it will appear as below:



Remember each public url generated from ngrok will only last for 2 hours. Therefore, to extend the usage period please repeat the 'Replacement' part. The webhook fulfilment is used on product_searching, product_enquiry and place_bid features. The rest of the features will not require webhook to perform.

If the Kommunicate free trial has expired, do this:

1. Sign up Kommunicate account and retrieve web script.

Please head to the following link to sign up an account:
https://dashboard.kommunicate.io/signup?product=kommunicate

Once successfully signing up the Kommunicate account, the user will be able to obtain a 'web' javascript for html file which looks like this:

```
</script>
<script type="text/javascript">
    (function(d, m){
        var kommunicateSettings =
            {"appId":"adcce7b6296ec3fcc77de67b1e9ce45d","popupWidget":true,"automaticChatOpenOnNavigation":true};
        var s = document.createElement("script"); s.type = "text/javascript"; s.async = true;
        s.src = "https://widget.kommunicate.io/v2/kommunicate.app";
        var h = document.getElementsByTagName("head")[0]; h.appendChild(s);
        window.kommunicate = m; m._globals = kommunicateSettings;
    })(document, window.kommunicate || {});
/* NOTE: Use webserver to view HTML files as the real-time update will not work if you directly open the HTML file in the browser. */
</script>
```

Please copy and paste your own code to store/templates/store/main.html and save the changes like below:

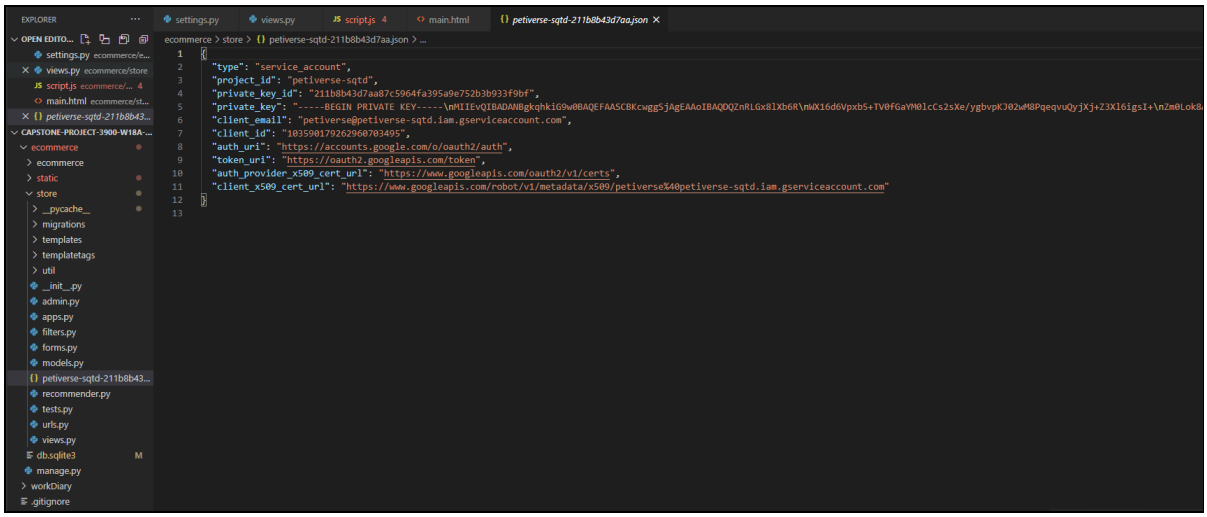
```
settings.py  views.py  script.js 4  main.html
ecommerce > store > templates > store > main.html > html > body > script
    /* When the user clicks on the button,
    toggle between hiding and showing the dropdown content */
    function toggleDropdown() {
        document.getElementById("myDropdown").classList.toggle("show");
    }

    // Close the dropdown if the user clicks outside of it
    window.onclick = function(event) {
        if (!event.target.matches('#user-icon')) {
            var dropdowns = document.getElementsByClassName("dropdown-content");
            var i;
            for (i = 0; i < dropdowns.length; i++) {
                var openDropdown = dropdowns[i];
                if (openDropdown.classList.contains('show')) {
                    openDropdown.classList.remove('show');
                }
            }
        }
    }
    </script>
    <script type="text/javascript">
        (function(d, m){
            var kommunicateSettings =
                {"appId":"adcce7b6296ec3fcc77de67b1e9ce45d","popupWidget":true,"automaticChatOpenOnNavigation":true};
            var s = document.createElement("script"); s.type = "text/javascript"; s.async = true;
            s.src = "https://widget.kommunicate.io/v2/kommunicate.app";
            var h = document.getElementsByTagName("head")[0]; h.appendChild(s);
            window.kommunicate = m; m._globals = kommunicateSettings;
        })(document, window.kommunicate || {});
    /* NOTE: Use webserver to view HTML files as the real-time update will not work if you directly open the HTML file in the browser. */
    </script>

</body>
</html>
```

2. Integrating Kommunicate with DialogFlow

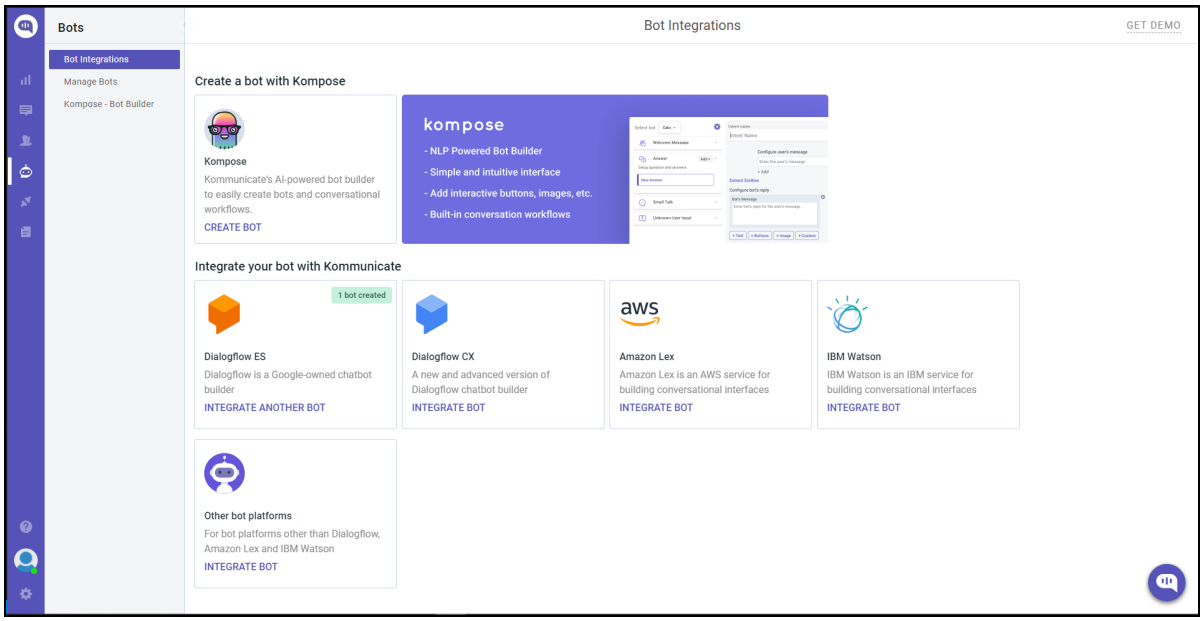
You can find the Petiverse agent private key json file is in the following location:

ecommerce/store/petiverse-sqtd-211b8b43d7aa.json

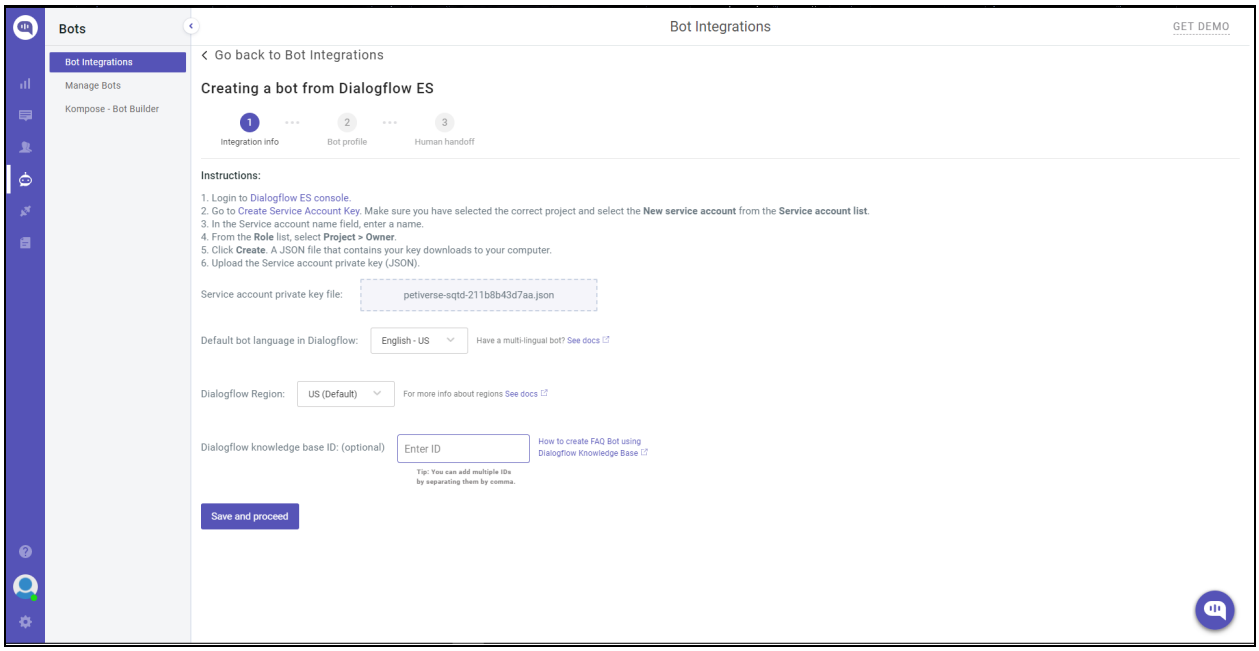
The correct file looks like:



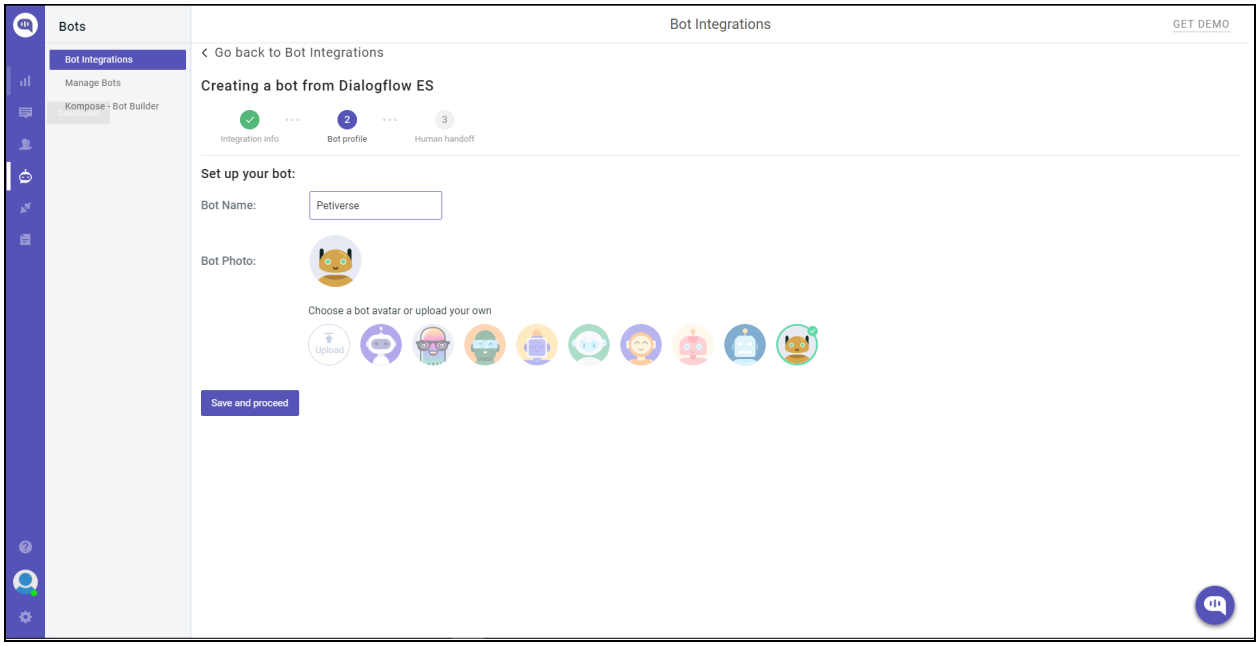
Now, return to the Kommunicate web page and under the ‘Integrate your bot with Kommunicate’ section choose ‘DialogFlow ES’. The correct web page is shown below:



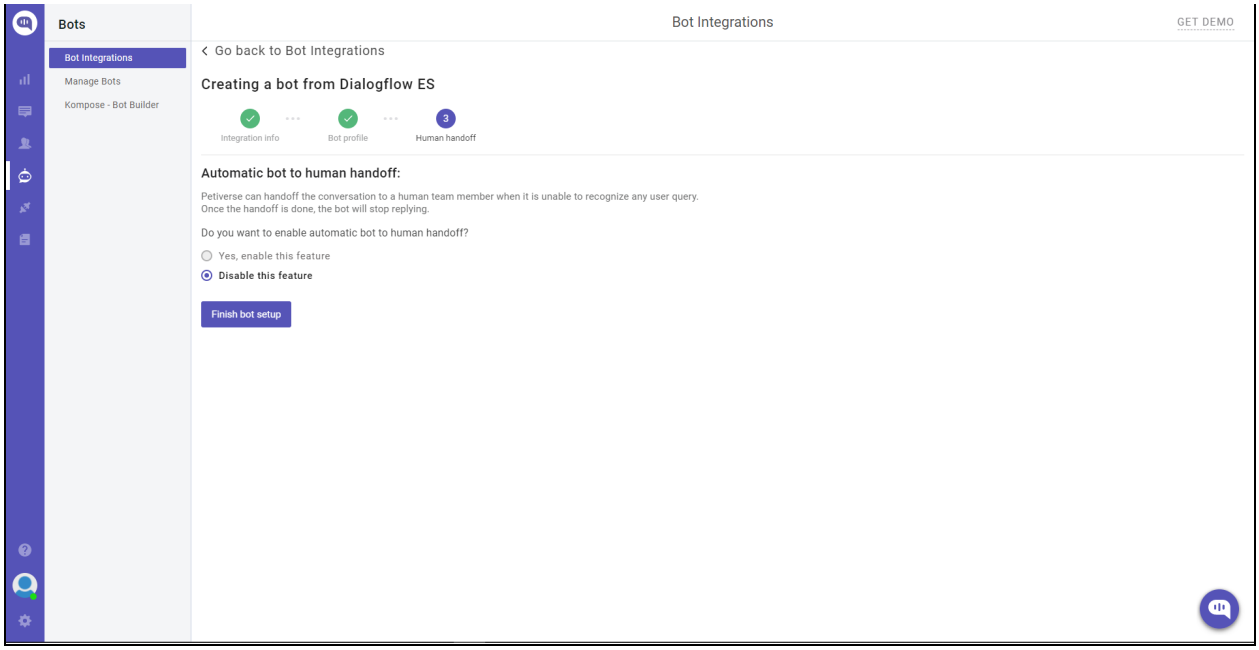
Drag the file from the folder and place it in the private key section. Please leave the rest of the fields as below shown and press ‘Save and proceed’.



After that, the user will be navigated to next section to customise the appearance of the chatbot and press ‘Save and proceed’ like below:



Now, the user has reached the final step for the bot integration. Please choose 'Disable this feature' and press 'Finish bot setup' to complete the bot integration like below:



Running the server

Once all setup steps have been completed, the site server should be ready to run. To run the server, ensure you are in the project root directory and run the command:

```
python3 ecommerce/manage.py runserver
```

This will start the project development server, and the website can now be accessed locally on your browser through the localhost address *127.0.0.1:8000*. The website will use our existing test database, stored in the file *ecommerce/db.sqlite3*. This database contains some test users and products, to demonstrate site functionality such as store pages, purchasing, and the recommendation system.

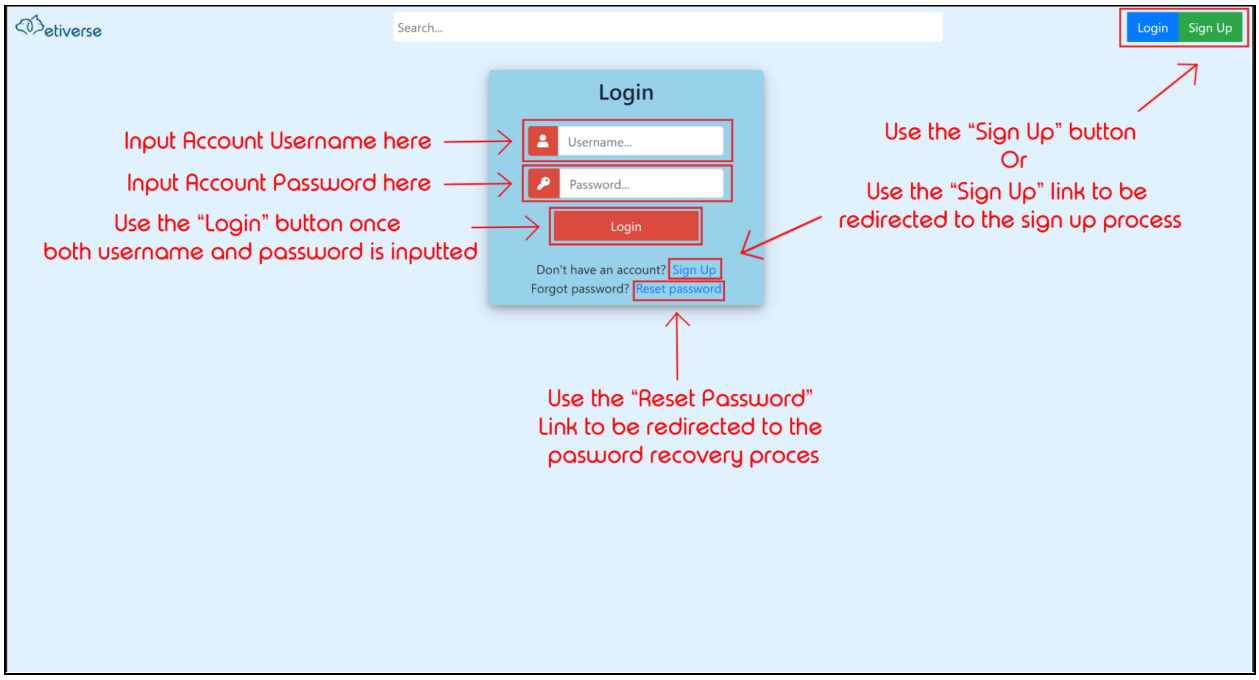
An existing admin account that can be used to inspect the site has the username *danny*, and the password *unsw2021*. If you wish to create your own account you can do so using the signup page on the website. This account can then be promoted to admin status using the admin site at the path */admin* while logged in to an existing admin user. The admin site allows existing admins to freely view and modify the site database, and take actions such as

deleting or modifying users, or even clearing all records if you wish to experiment with a fresh database.

Note: If you wish to modify the database schema by running a migration, the final line of the file *ecommerce/store/views.py* (check_auction_time()) will have to first be commented out. Failing to do so may cause an infinite loop during the migration process. The line can be uncommented after the migration has completed to restore site functionality. This is a development note and is not required to test existing site functionality.

Usage

Login Page



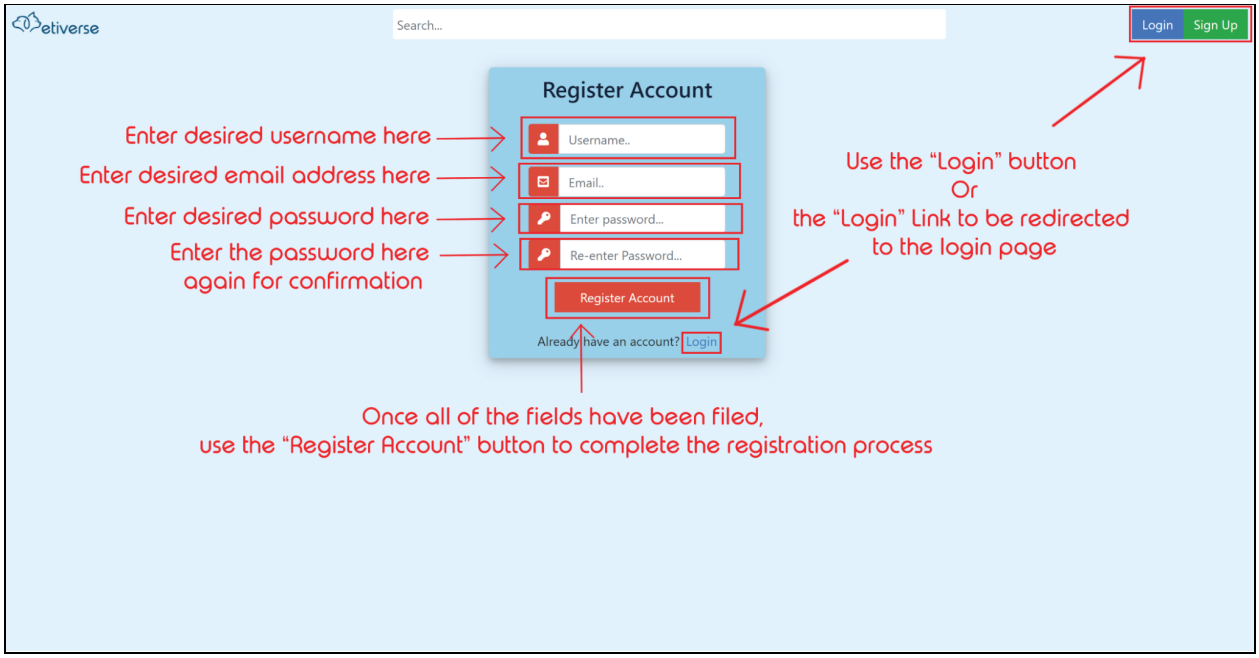
On the Login page, a user can register an account by pressing the *Sign Up* button, which will send the user to the *Registration* page.

Existing users can reset their password by pressing the *Reset Password* button when they have forgotten their password.

An anonymous user can also access the *Home* page by pressing the Petiverse logo on the top left corner of the page, which will show up a list of existing products.

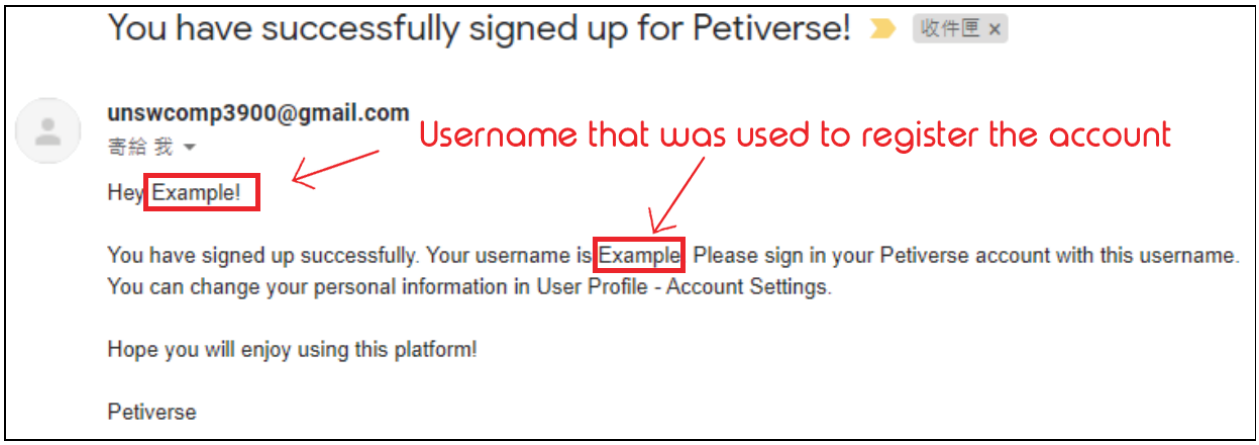
The search bar on the top allows a user to search for products based on product name, seller name and tags.

Registration Page

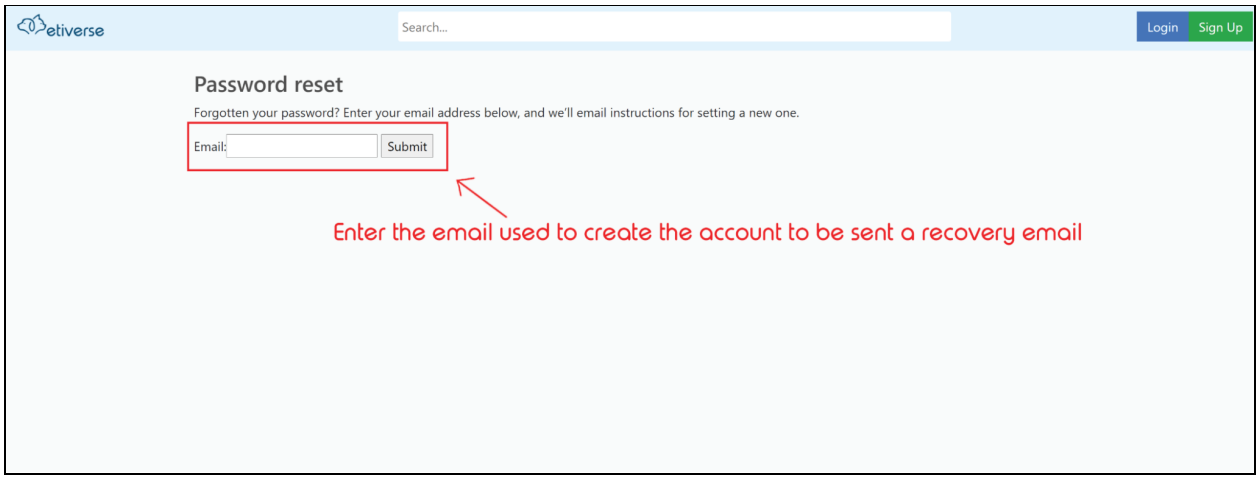


To register an account, the user must fill in all the required fields and satisfy the criteria we have for each field. Once all fields are inputted satisfactorily, the user can press the *Register Account* button to complete the registration process. Once pressed, the user will receive a confirmation email as shown below that would be sent to the email that was used to register the account. The email has a gentle reminder of what the username for the account is and a simple instruction on how to update the user's personal detail.

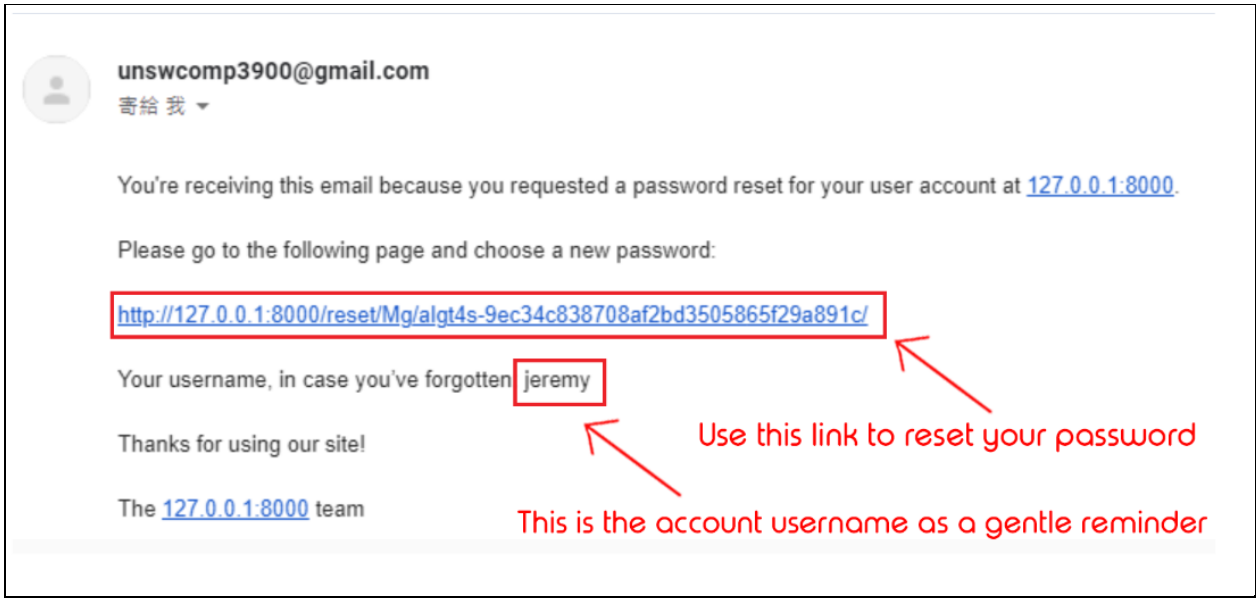
If the user happens to have already have an account, they can press on either the *Login* link underneath the *Register Account* button or on the top right next to the *Sign Up* button



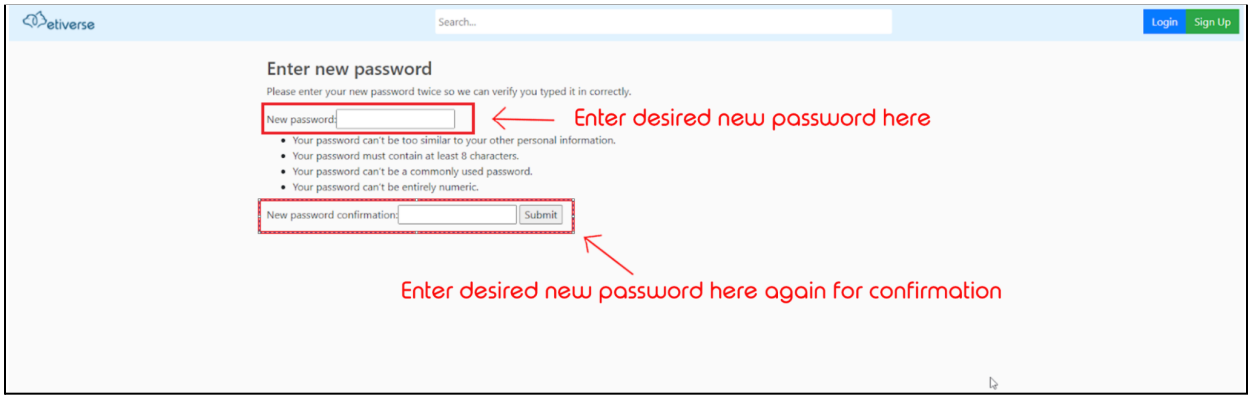
Password Reset Page



In order for a user to reset their password, the user has to fill in the email address that they have used to register his account and submit the form. If the email entered is in the database and is associated with an account, the user can expect a password recovery email to be sent as shown below that includes a link and a gentle reminder of the account username associated with the email.

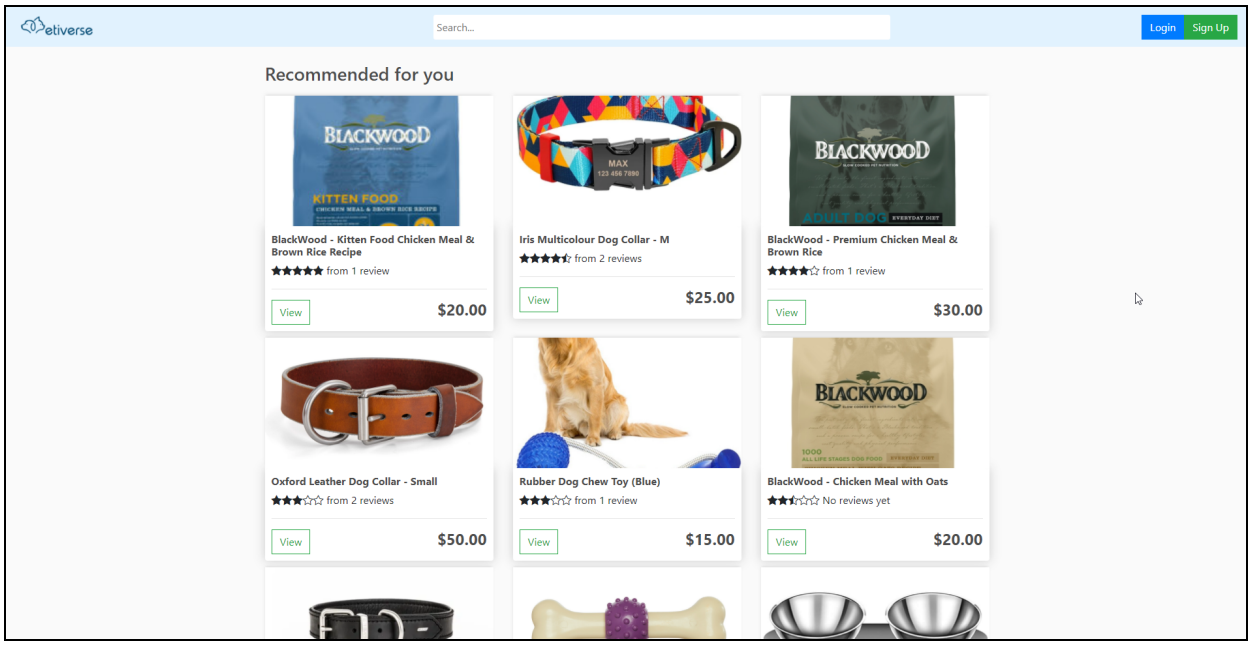


If the user presses the link, they will be redirected to a new page as shown below where they will be prompted to enter in a new password that satisfies the criteria listed under it. If the password is unsatisfactory, a warning pop-up will appear to show that it is wrong with the given password and it will prompt the user to enter a new password. Once a new password is decided, the user will have to enter that password again as a confirmation.



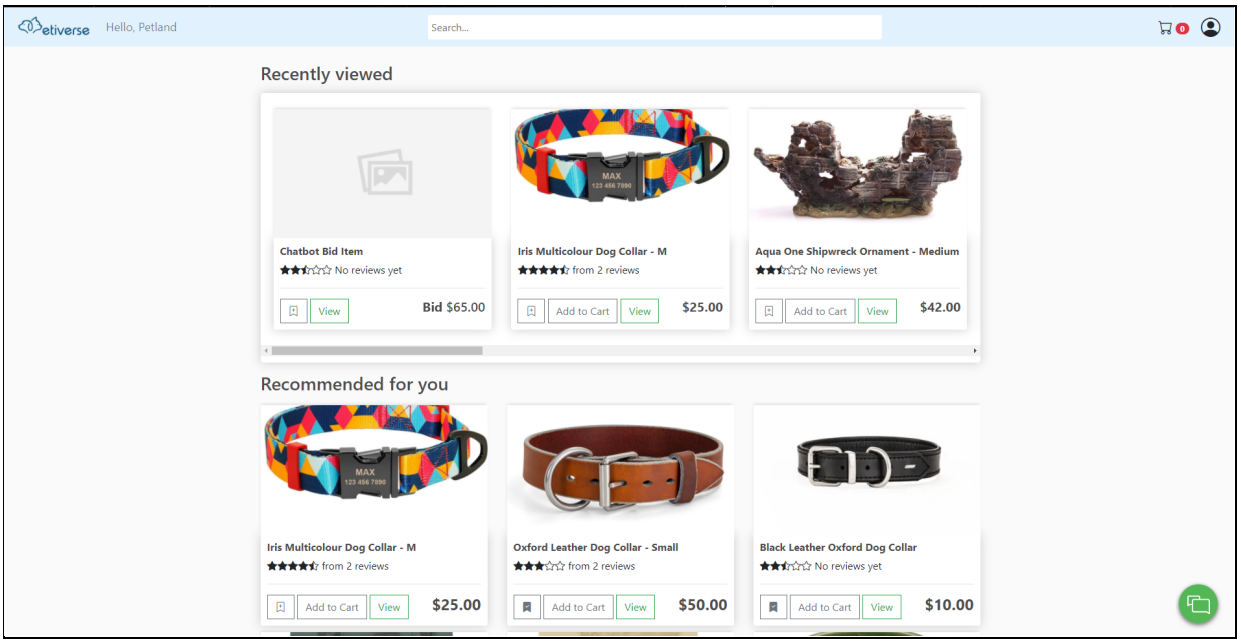
Once the password recovery process is complete, the user will be automatically redirected to the login page where they can log in to their account with their newly made password.

Home Page (Anonymous User)



An anonymous user is only allowed to view products and prohibited to make any purchase. In order to obtain complete user experience, users are required to register and login to the website.

Home Page (Logged in User)

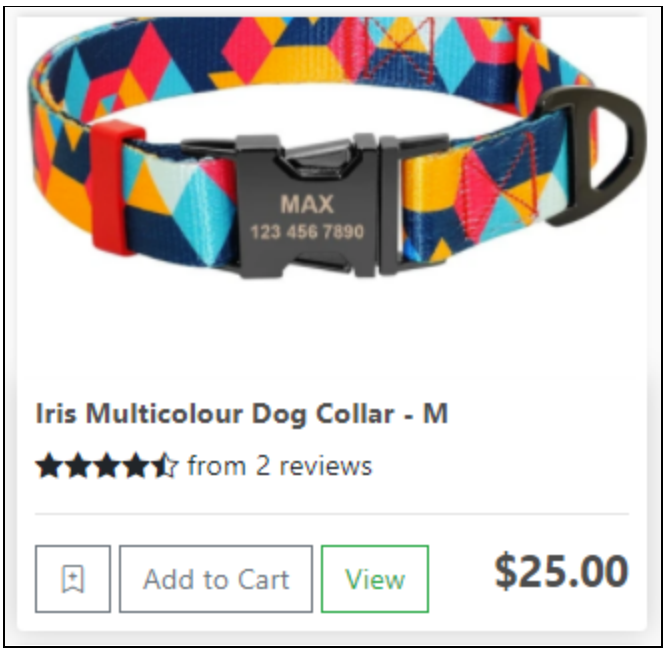


Signed in users will see a similar page like the picture above. The only difference is the display sequence of the products which are based on their personal viewed pages and purchases.

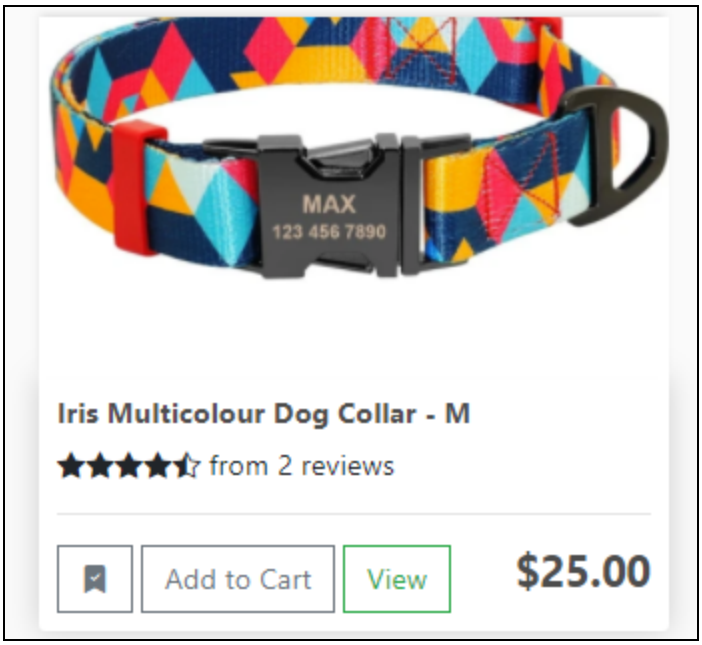
Users can hit the *Add to cart* button to add a product into their personal shopping cart and the red circled number icon on the top right corner will update the number based on the number of items in cart. Shown as the following picture below:

Users can also view the information of any product by clicking the *View* button or the image of the product. This will redirect them to the specific product page.

Users are able to add any product to their wishlist by pressing the bookmark icon beside the 'Add to cart' button. After the product successfully is added to the user's wishlist, the bookmark icon will have a tick within it instead of + . The status will be shown as below:

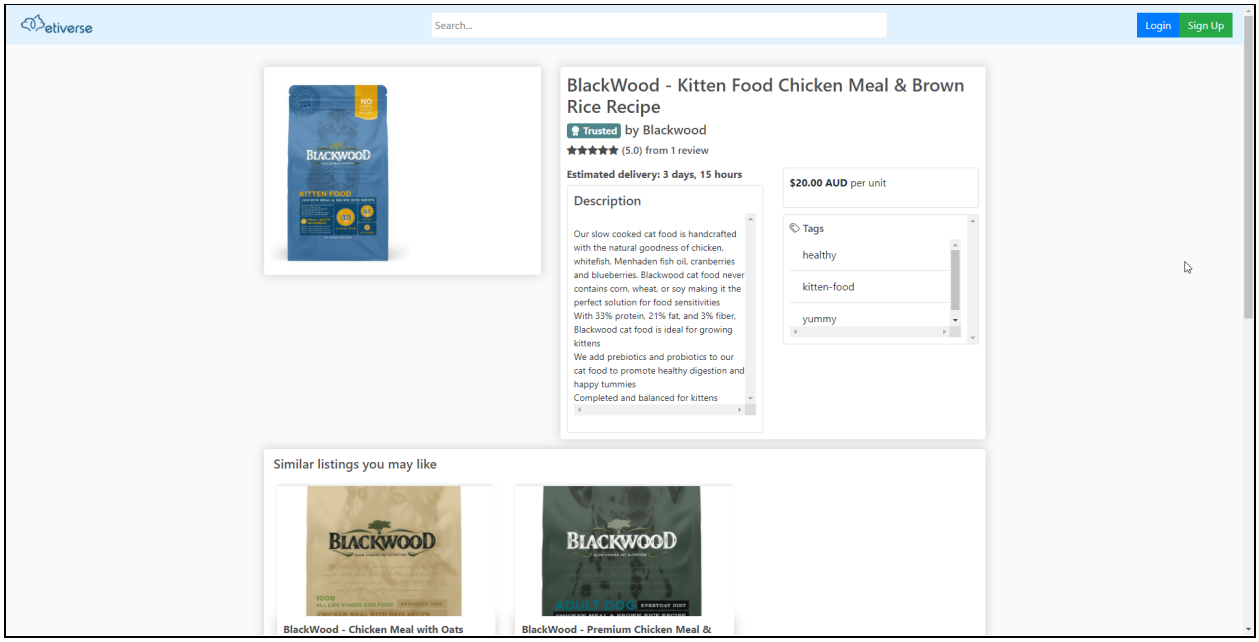


Before



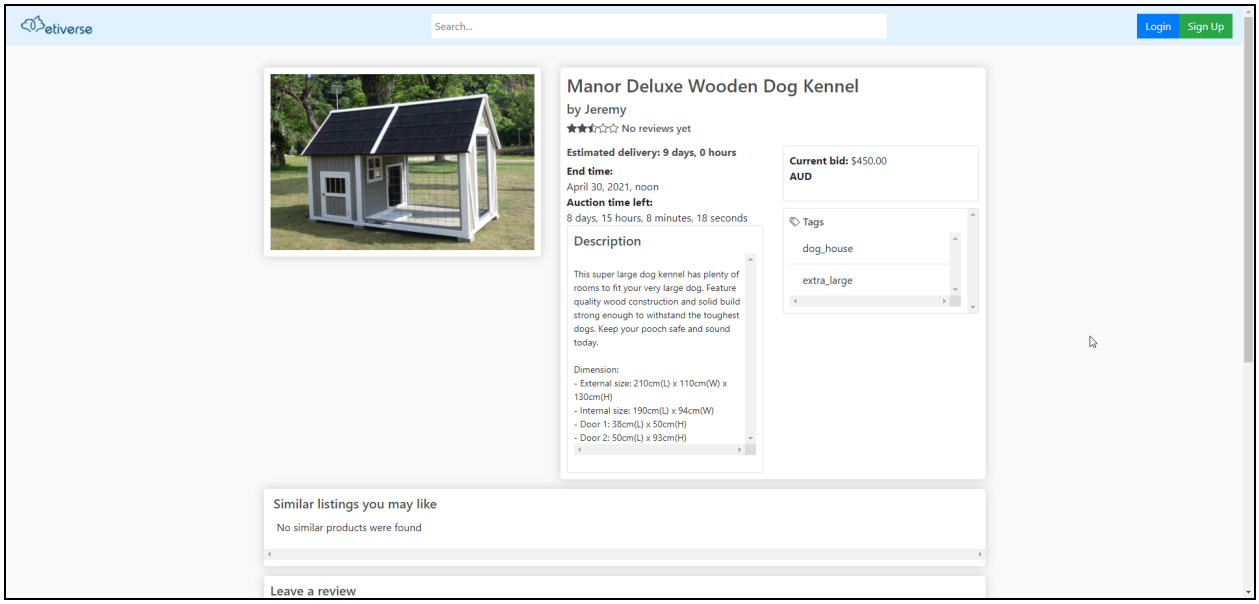
After

Product Page (Anonymous User) (Sales Type)



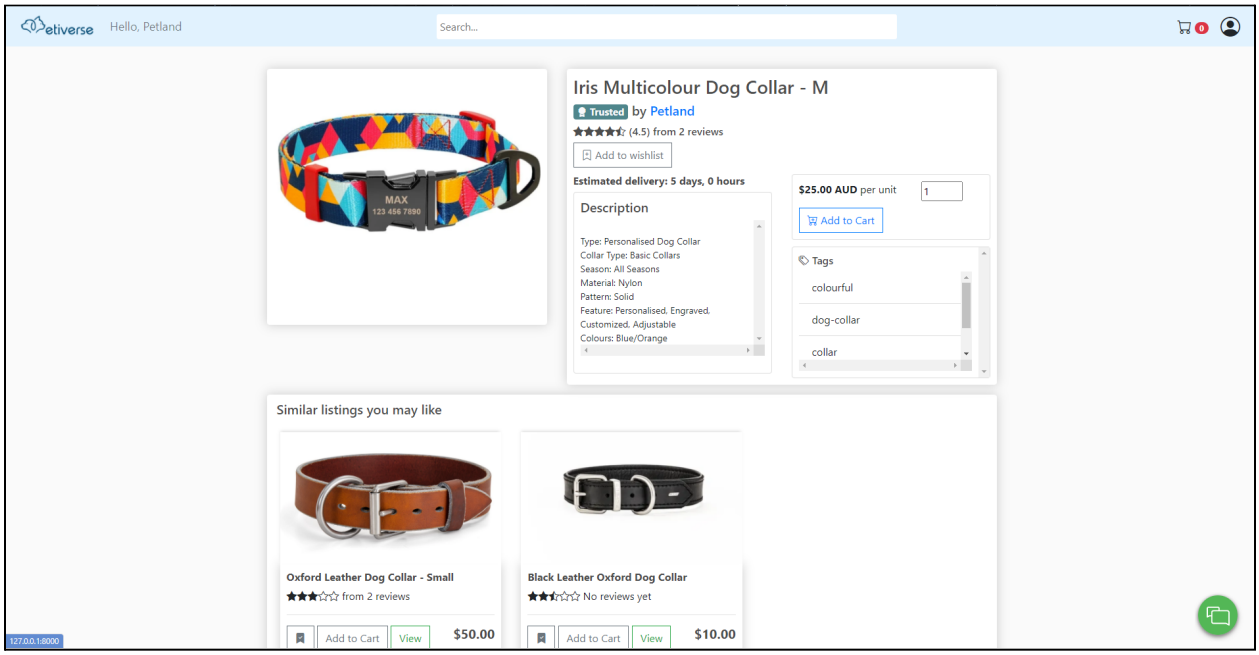
An anonymous user will be able to access the *Product* page for a sales listing from the *Home* page, but the user will not be able to add a product to the cart, and access to other user's profiles.

Product Page (Anonymous User) (Auction Type)



An anonymous user will be able to access the *Product* page for an auction from the *Home* page. The user can see the current bid, end date and the timer for the auction product. However, an anonymous user is not allowed to place a bid for an auction product.

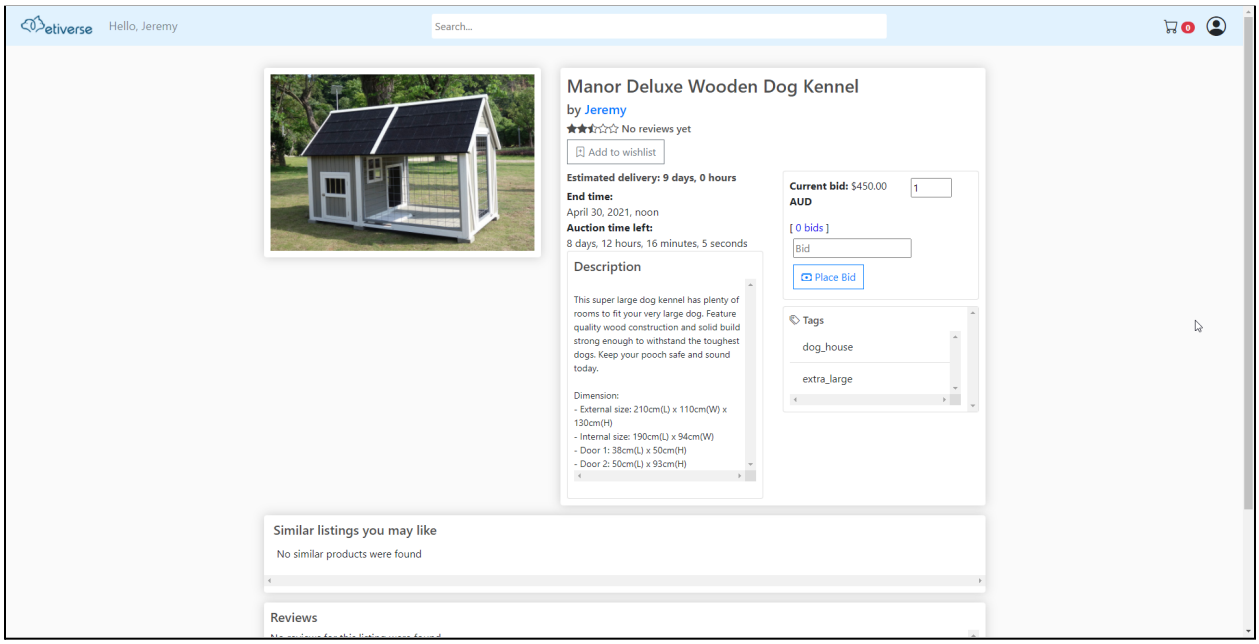
Product Page (Information section and similar listing) (Sales Type)



The product page shows the user detailed information about the product. Such as the average review score, estimated delivery days, price, seller, description and tags.

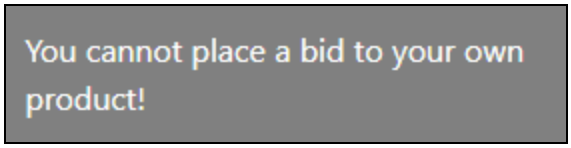
If signed in, the user has a range of extra features that they have access to compared to a guest user. Firstly, they can access the seller's profile by pressing the seller name which is indicated by the blue text. The user can also add the product into their wishlist by pressing the *Add to Wishlist* button, so that they can keep track of the products that they are interested in. By pressing the *Add to Cart* button, users can add the product into their shopping cart ready for checkout. Below the product description, the user has access to products that are similar to the current product that the user is observing, based on the same product tags.

Product Page (Information section and similar listing) (Auction Type)

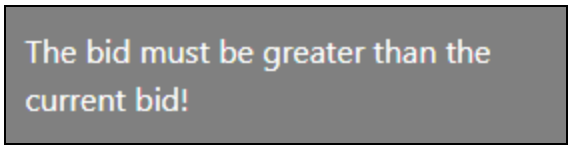


Same as the sales listing product page, the auction product page shows almost the same detailed information about the product. The end date of the auction is displayed on the *product* page with a timer below it. If signed in, the user can also add the product into their wishlist by pressing the *Add to Wishlist* button, so that he can keep track of the products that he is interested in. By pressing on the *Place Bid* button, the user will be able to place a new bid to the product. There are certain rules in order to place a valid bid.

Firstly, a seller cannot place a bid to his own product in order to prevent the seller artificially raising the price of the auction. The image to the right shows the error message when a seller is trying to place a bid to his own product.

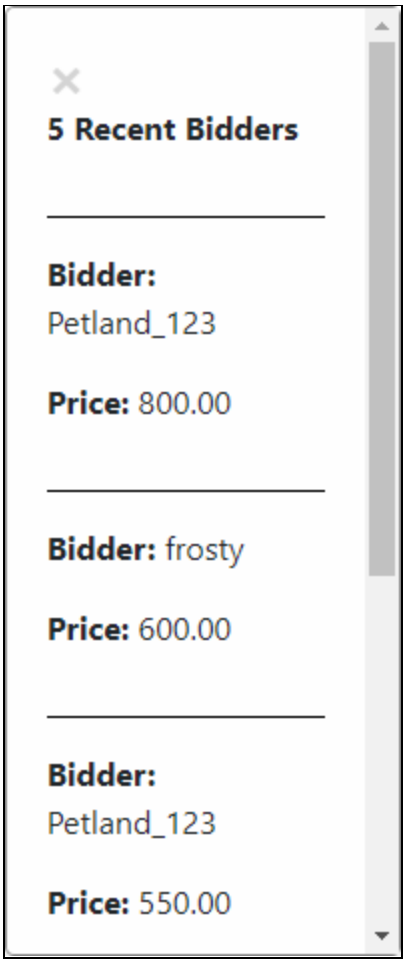


Secondly, a user cannot place a bid which is lower or equal to the current bid price because the price of the product is meant to keep increasing along the auction. The figure to the right shows the error message when a user is attempting this invalid action.



The *"[0 bids]"* under the current bid price shows the number of bids has been placed for the product. The number will keep increasing when there are users placing bids during the

auction time. A user can check the five recent bidders of the product by pressing on the number of placed bids.



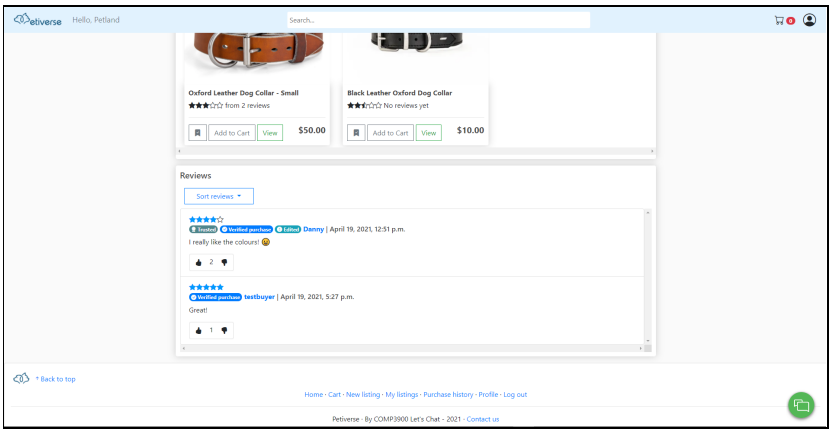
The figure on the right shows the list of the five recent bidders of the auction product. It shows the name of the bidder and the bid price they have placed.

A user can scroll down to see the rest of the bidders by using the scroll bar on the right.

Once the auction has ended, the product will be automatically added to the winner's shopping cart and an email will be sent to both seller and buyer to notify them about the result of the auction.

If the buyer did not proceed the payment within 3 days, the seller can contact the admin and the buyer will receive a penalty about it. Which the buyer's account may get banned from Petiverse or even have to pay an indemnity to the seller.

Product Page (Reviews section)



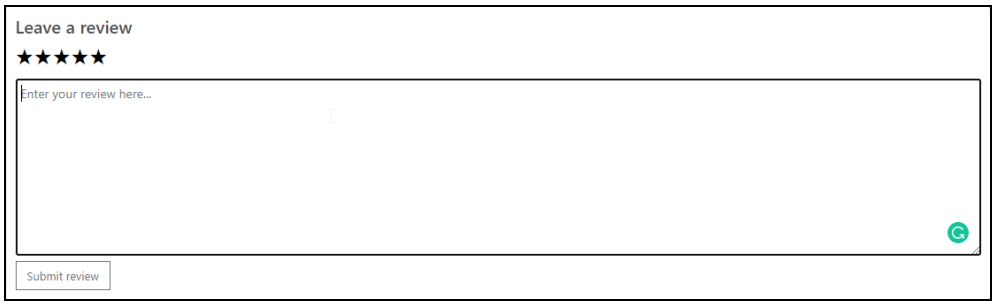
The reviews section for a product lies in the bottom of the *Product* page. A user can write a review for the product using a star system with the minimum of 1 star and a maximum of 5

stars accompanied by a text field that allows users to attach comments to their star reviews. On publishing the review, other users can see the reviewer's username, their badges, the time the review was published and the amount of likes or dislikes the review drew.

On the topic of likes and dislikes, other users can upvote or downvote a review based on the usefulness and truthfulness of the review. If users deem the review useful, they can like the review so that it will rank higher than other reviews. If users deem the review to be fake or dishonest, they can downvote the review so that it would be placed last if sorted based on their rating.

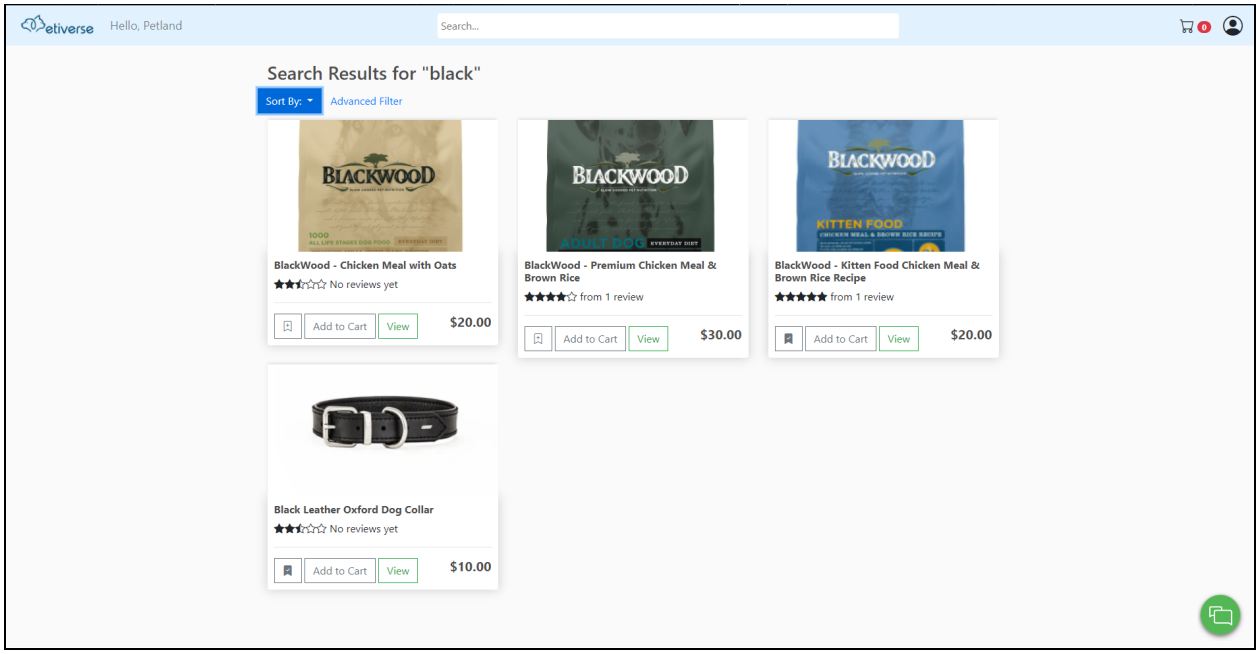
A seller is not able to leave a review on their own products to avoid fake reviews with the intent to increase their average rating of the product. The section to write a review is hidden as shown in the figure above.

The figure below shows the writing review section for users that are not viewing their own products.



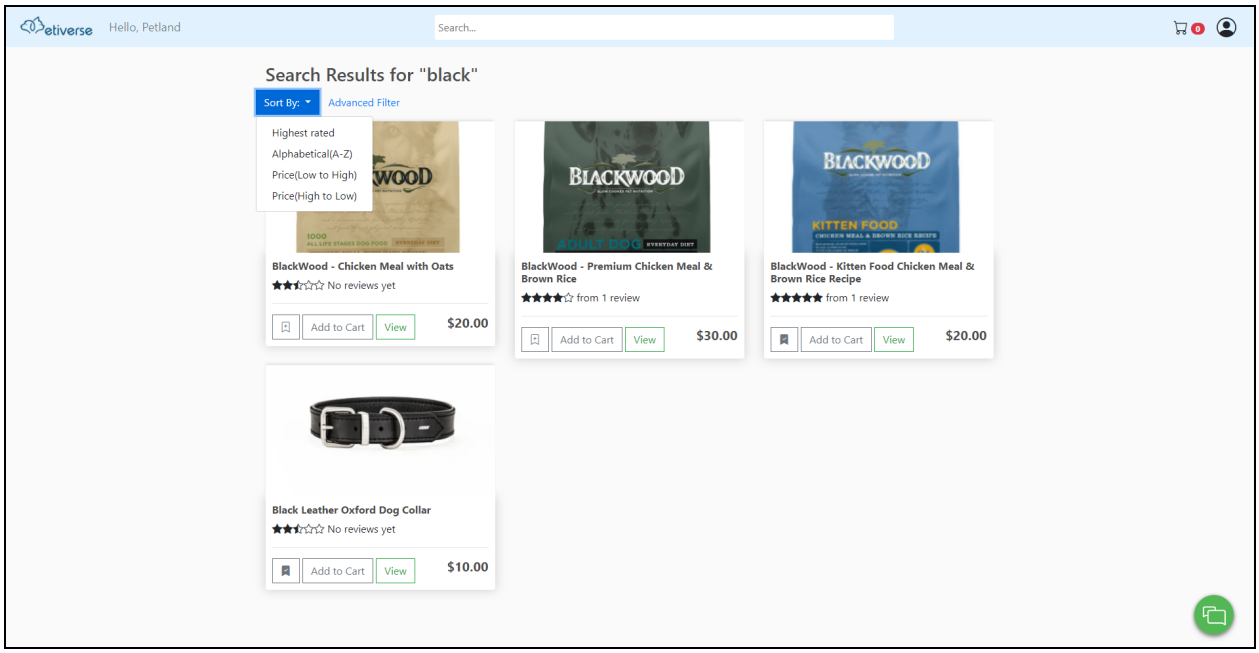
As shown in the figure above, a user can edit his review by pressing on the *Edit* button if the user wishes to make a change to the review. A user also has the freedom to remove the review by pressing the *Delete* button.

Search Page (Default)



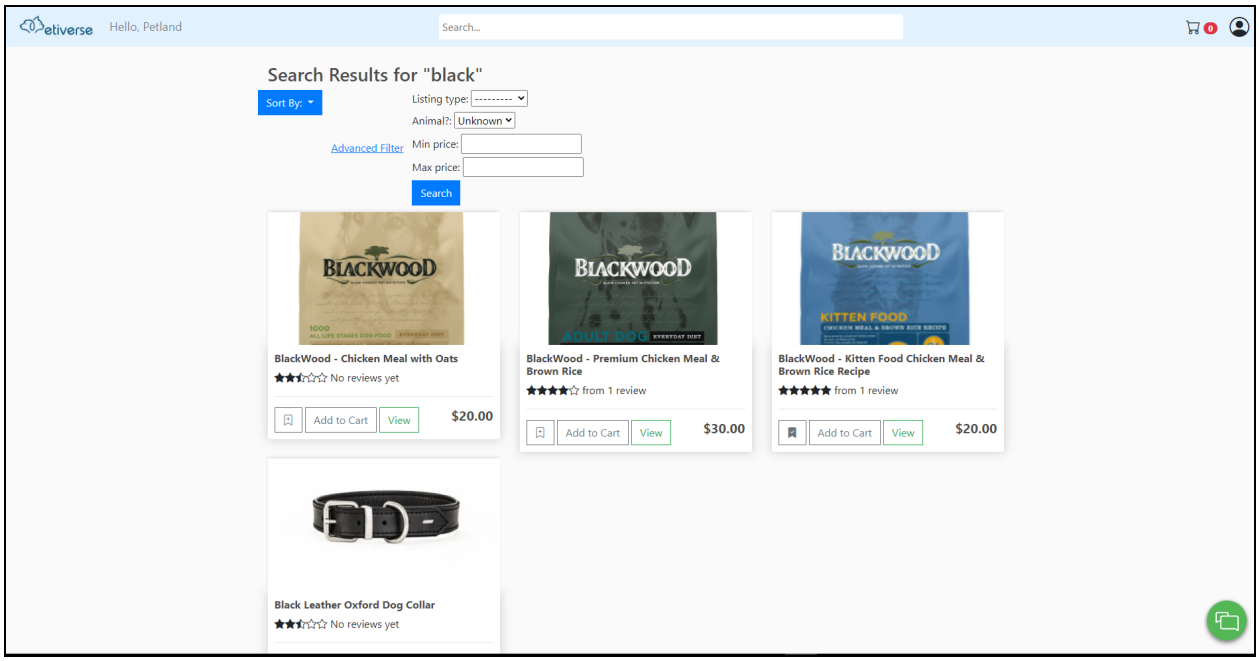
The search bar at the top of every page allows the user to input queries to search for items that the user is interested in. It exists on every page to allow for easy user accessibility as it is one of the core features of the platform. After pressing the enter key, a user will be sent to the search result page that shows up the match items with the inputted keyword(s).

Search Page (Sort options)



Once the user is redirected to the search result page, the user's query is saved and displayed next to "Search results for" and is in quotations. The search results displayed on this page is a result of the query being matched to 3 parameters of each product. The query is matched against the product name, the seller name and all the tags associated with the product. So products on the page shown above either have the word *"black"* in it's product name, seller name or in one of the tags. From here, the user can further refine their search by further accessing the sorting and filtering option. By pressing on the *Sort by* button, it will list a few sorting options. Sorting by *Highest Rated* means that products will be rated based on their review score. This takes into account how many reviews a product has. Products that have 0 review are given 2.5 stars by default and will have a lower rating count. Next the user also has access to some simple sorting options such as alphabetically and by price.

Search Page (Advanced Filter)

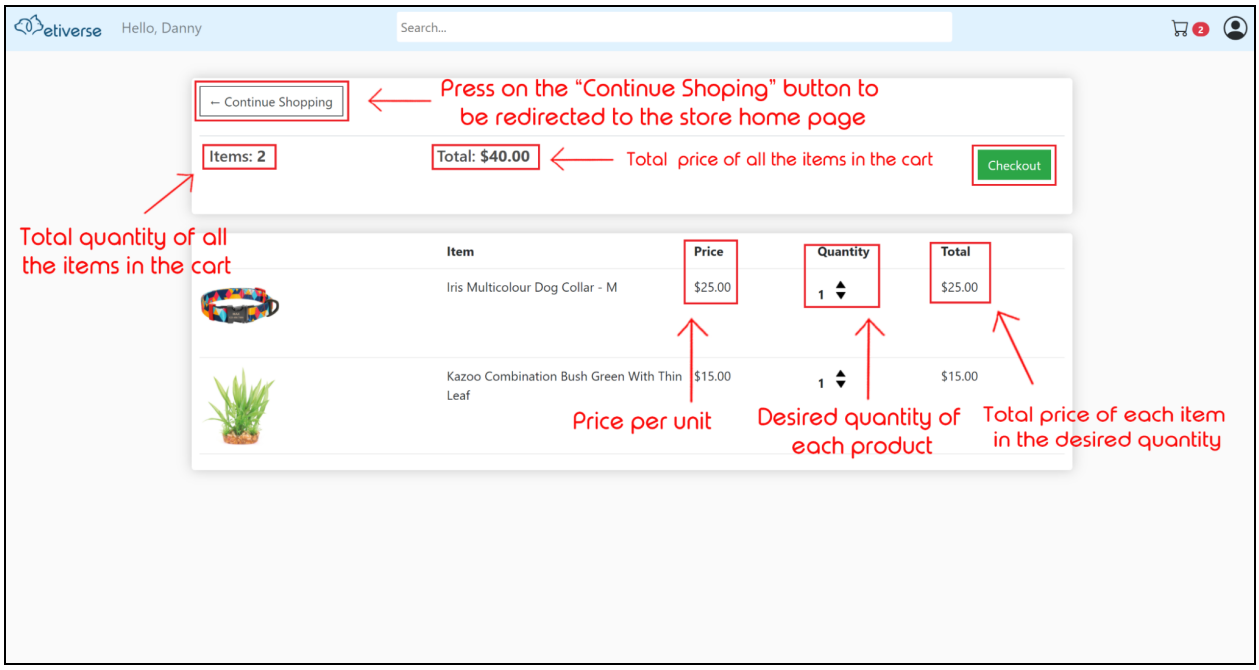


To access the advanced filter, users can click on the "Advanced Filter" link next to the *"Sort By"* button. The *Advanced Filter* button allows a user to view specific products with a list of options. Users have several options on which to filter. To start with, users can sort products based on their listing type. The dotted line as shown in the *"Listing type"* field in the figure above means that the products listed in the search are both of auction and sales type. If users choose the auction only option then only products that are of the type auction will appear and similarly if users choose the sales only option, then only products that are of the type sales will appear.

Next in the filter list is *"Animal?"*. This filter allows users to refine their search based on the product type. By using this filter, users can either choose between "*pets only*" or "*pet products only*" . If the user chooses "*Unknown*" as shown in the figure above, the search result will list listings that are either of pet or pet products.

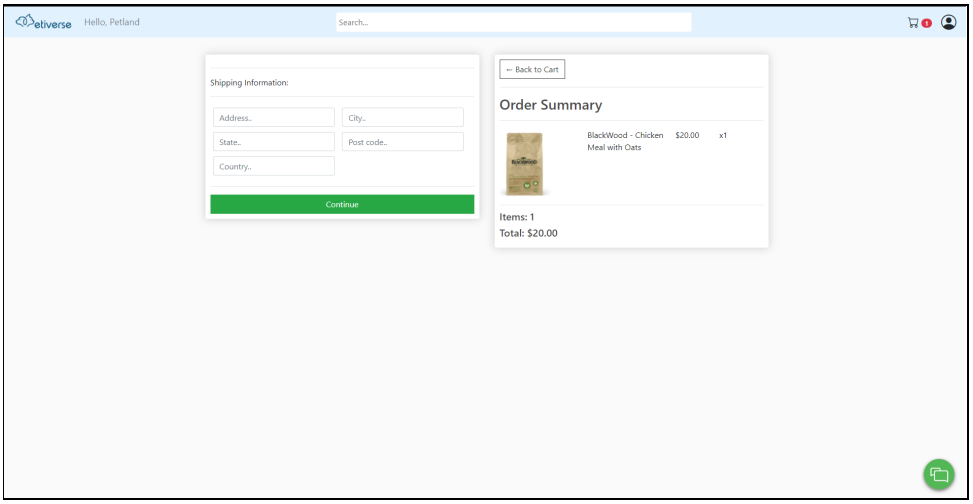
Last in the filter is a price range. This allows users to set a price range that they are looking for in the products search result. By leaving either of the fields empty, only the inputted field will be affected. For example, if the user only input a value into the minimum price, then the search result will only use the minimum price filter and vice versa. If the user inputs a value on both, it will use both filters to filter the search result. If the filters counteract each other, e.g. a minimum value of $30 but a maximum value of $25. This will return an empty search result as expected.

Cart Page

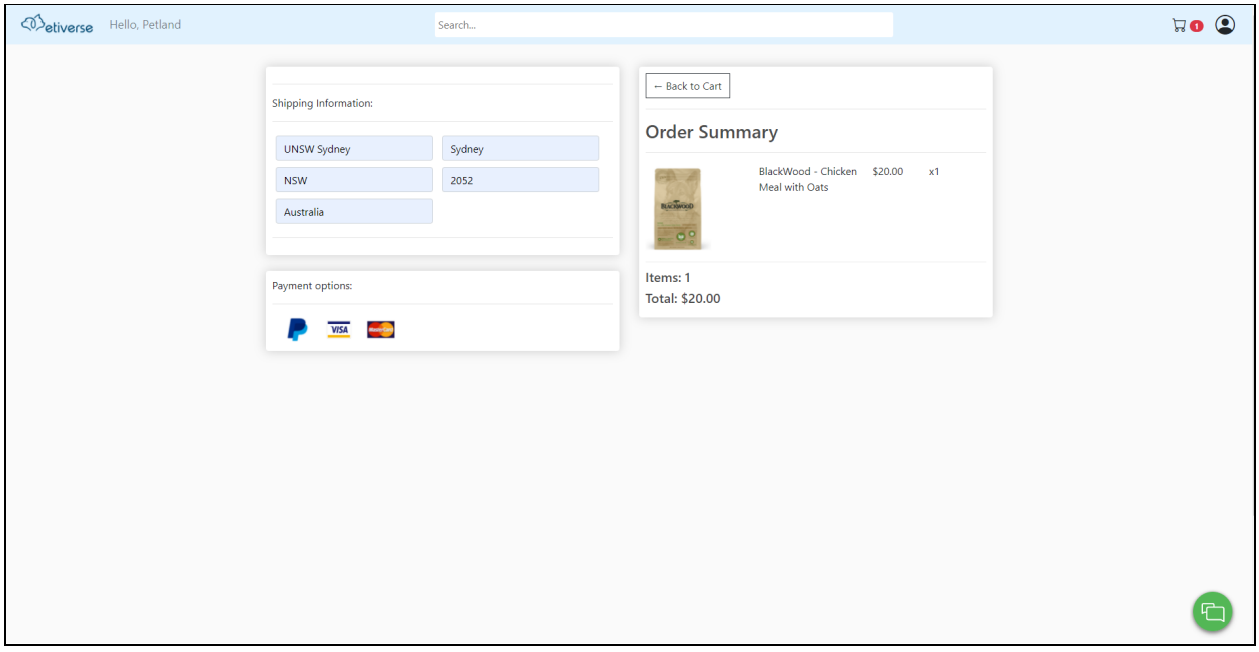


Once the user has decided on all the products they want to purchase, by clicking on the cart icon on the top right will they be redirected to the *"Cart"* page. As shown above, the cart page outlines all the products that the users have added to the cart. Users can also further modify the amount of each product they would like to purchase on this page. First of all, if the user decides that they want to keep on browsing the store, they can either press on the "*Continue shopping*" button to be redirected to the store's home page or they could simply enter in another search query to be redirected to the search result page. Otherwise, the user has access to the information as shown on the figure above. Users can also modify their quantity by using the arrow buttons to increase or lower the quantity by an increment of 1. Setting a quantity of a product to 0 will automatically remove the product from the cart. Once the user decides that they are happy with the content of the cart, they can proceed to press on the green "*Checkout*" button to be redirected to the checkout page.

Checkout Page

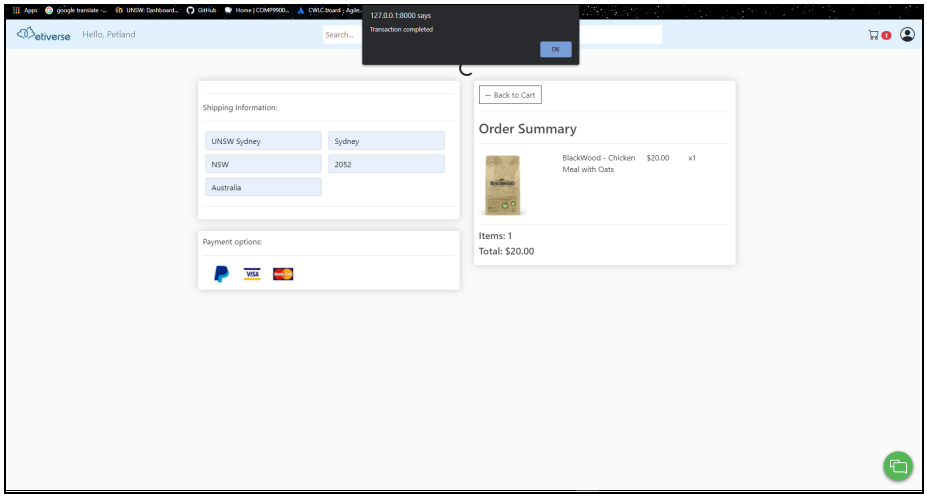


Here the user has access to the order summary which basically summarises and lists all the items the user has in the cart previously. Here the user is also asked to input in their shipping information which once payment is completed will be sent to the seller to complete the transaction.



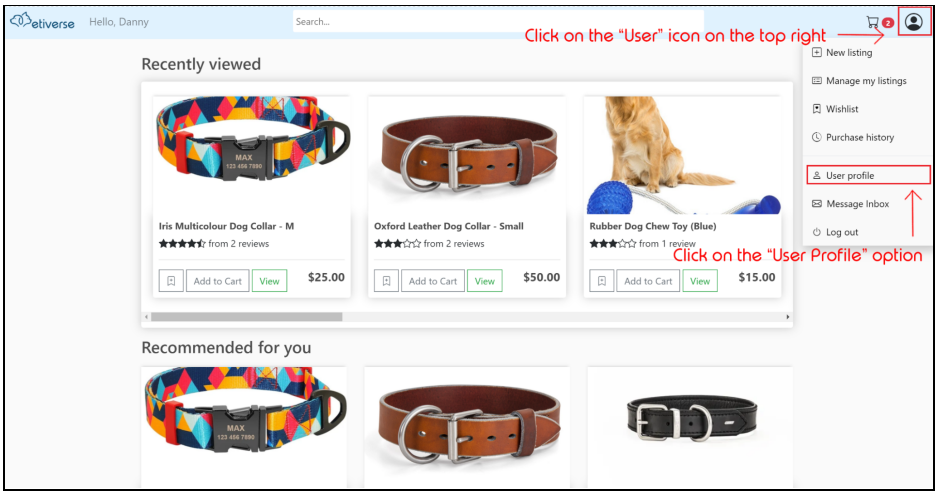
Once completed, the user presses the “Continue” button and then they will be asked to choose their payment option. the user will then be able to choose to pay with Paypal, Visa or MasterCard. For this project, we implemented a dummy payment system where by pressing on one of the payment options will always return a success.

Checkout Page (Transaction success alert)

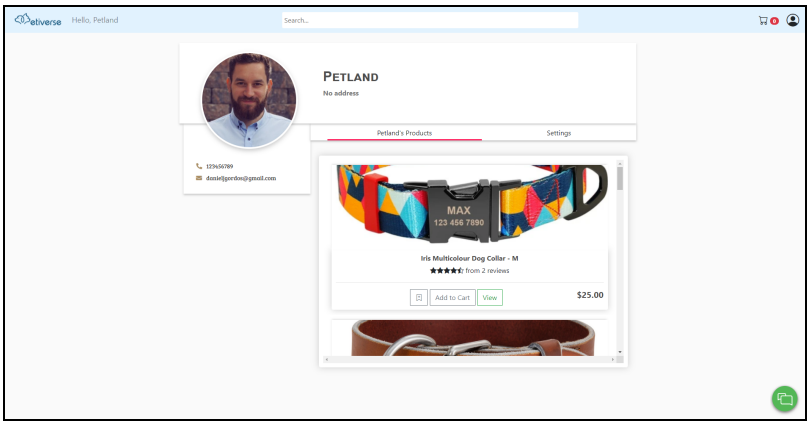


As shown above, after any of the payment options has been clicked, it will take a few seconds to send out the information and receive the confirmation back. Once everything is completed, A transaction success alert will pop up to alert the user that the transaction is complete.

User Profile Page

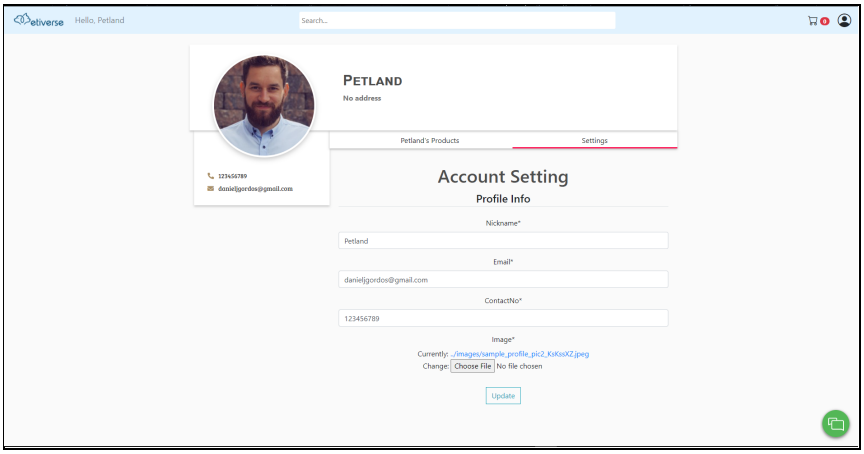


Each user on the platform has a user profile. They can access this by pressing on the user icon on the top right corner of the website and then on the "*User Profile*" button on the menu.



Once in the profile page, the user's username, profile picture and contact details appear. Under the *"Petland's Products"* tab, all active listings that the user has on the platform is listed. Users can use this feature to find out more about other products that the seller is selling.

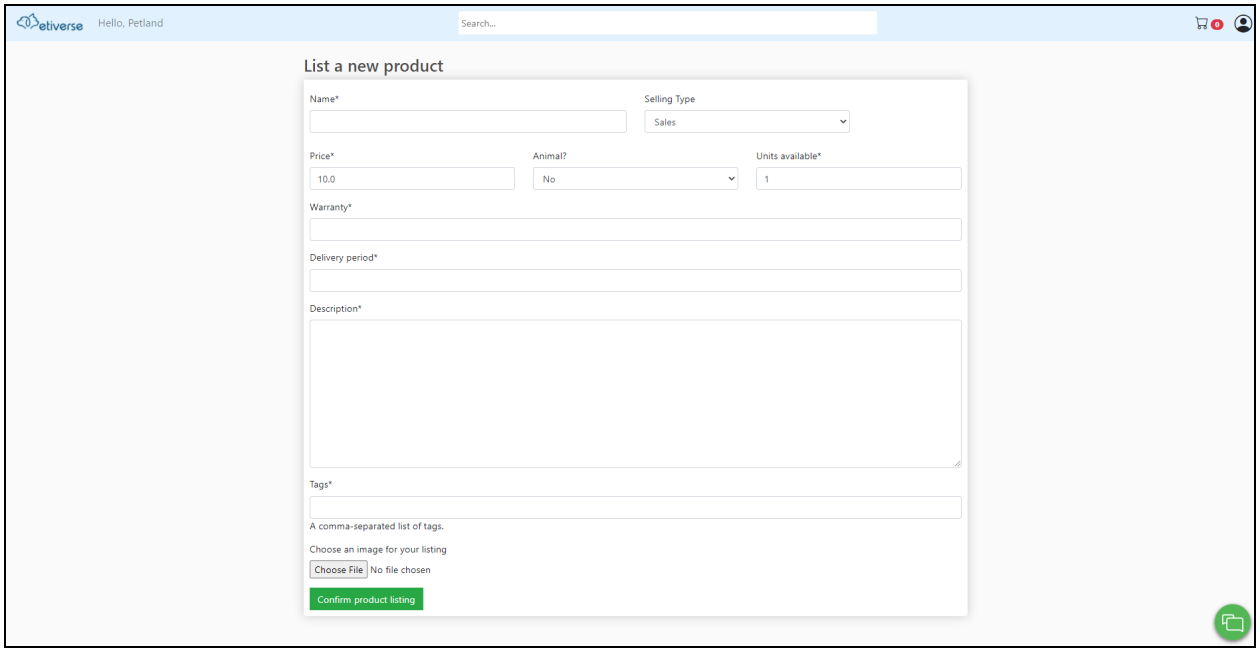
User Profile Page (Setting tab)



Under the settings tab, the user can modify their account details. Editable fields of the profile name include the user's nickname, email, contact number and their profile picture. The "Nickname" field at default is the account's username. When changed, it alters the name that the user is known by on this platform. This is shown by the change in the "Hello <Name>" greeting which appears on the top left next to our logo. It also changes this particular user's name in any of their product listing and their profile page.

This change however does not affect the username. Hence users that change their nickname should be reminded to log in using their original username that was used to register the account. Changes to email and contact number should be accurate as that is one of the ways another user can contact you by. To change the user's profile picture, user's can choose to upload an image file using the "*Choose file*" button before clicking Update when all changes are made satisfactorily.

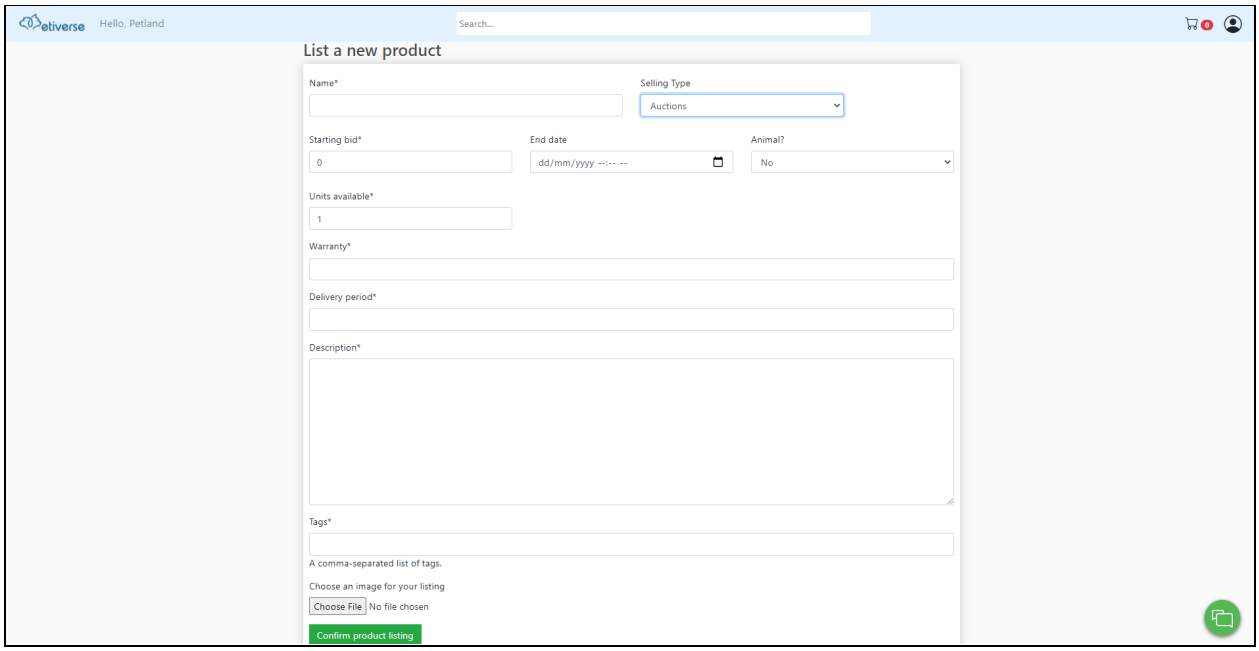
New listing Page (Sales item form)



Any user on the platform can sell a product on the platform by creating a new product listing page. The figure above shows the form which the user will encounter when choosing to sell a product. All the fields shown are required to be filled to create a new listing successfully. Starting from the top, the user must input a name for the listing. This normally is the name of the product being sold. Users then can choose a listing type. For now we will cover a normal fixed price sales listing. Next the user sets a price at which they would like the product to sell per unit. Next to it, the user chooses whether the product about to be listed is an animal or an animal product before choosing the quantity of the product they would like to be made available.

While not all products will have a warranty attached to them, sellers are required to input a value for this for record and transparency. The seller is also required to write down an estimated delivery period from which they believe the product will arrive at the buyer. Next is the product description. This is where users outline and describe any information regarding the product. Lastly, users can attach tags to the product by providing a list of tags separated by commas that will aid in the visibility of the listing on the platform. The user is also then recommended to upload an image of the product for the listing. While not necessary this may boost the number of sales of this product. Once done, the user needs to press on the "*confirm product listing*" button to complete the listing registration process.

New listing Page (Auction item form)

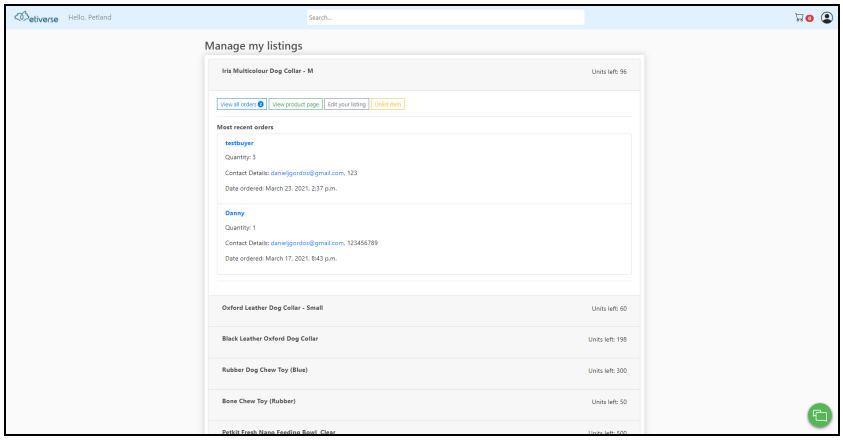


As outlined before, users have the option to make their listing an auction type. This means that other users can put bids up for the product making it a more competitive method of selling.

Most of the fields are similar to the fixed price sales listing except for the end date and the starting bid of the product. Users are required to input an end date for the listing to decide on when they want the auction to end. A longer date may mean a bigger period of competitive bids however sales of a product will take longer.

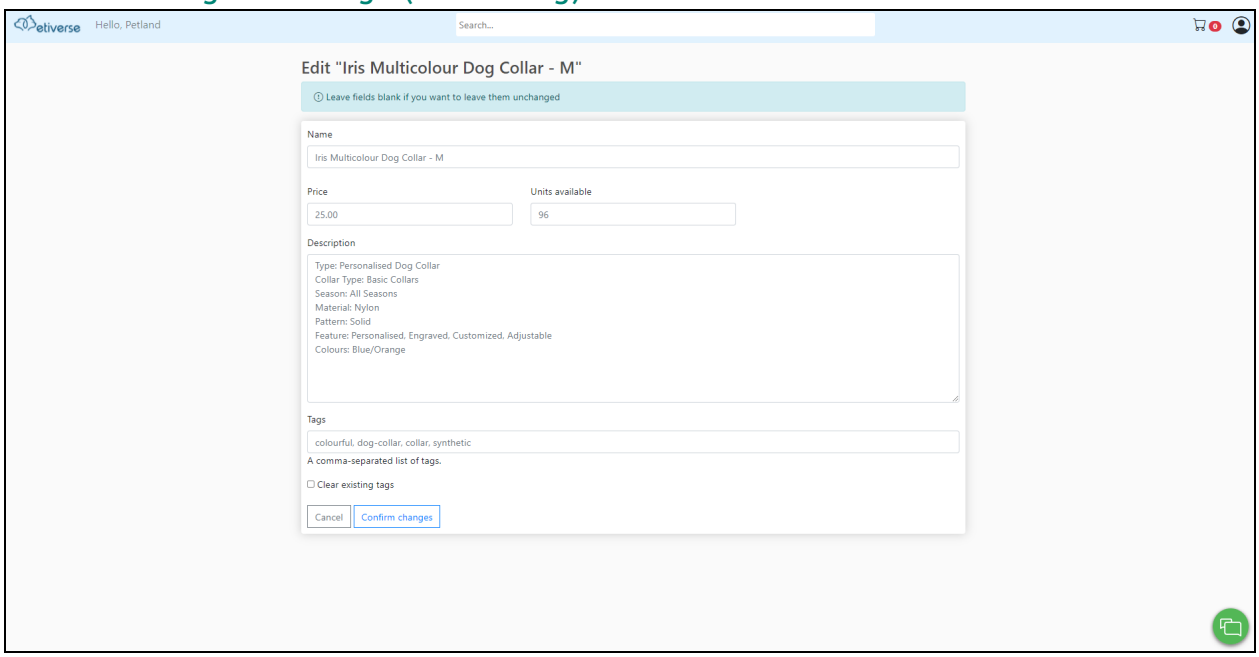
Next the user is also prompted to input in a starting bid. This is the price that bids will start rising from. Any bids lower than this bid will not be accepted and if no bids above the starting bid exist by the end date, then the product has not successfully sold. Lastly the user needs to input in a quantity for the product. Differing from the sales listing, a product sold through auction will sell all available quantities at a total of the bid price and not sell for the bid price each.

Product Management Page



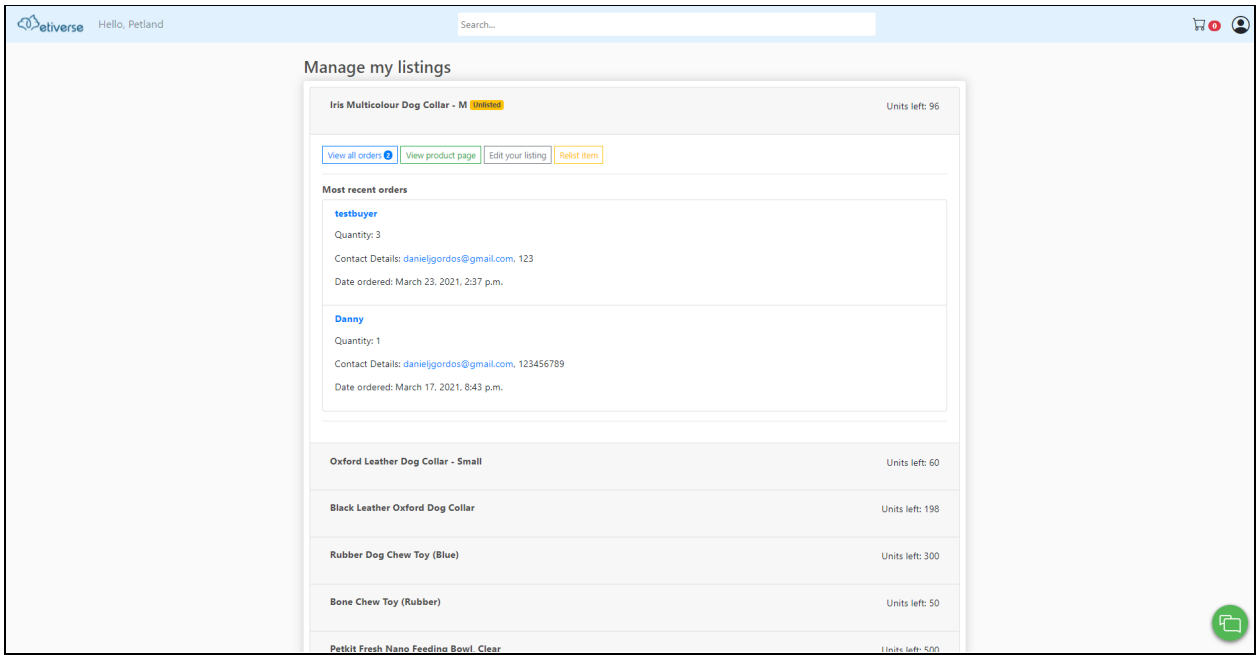
A user who has posted a listing on the platform will have access to features of the "*Manage listing*" page. Here users can track all orders made through the platform. Each order consists of the buyer's name, contact details and the date ordered. The user can also edit the listing by using the "*Edit your listing*" button and unlist the listing by using the "*Unlist item*" button if any modification is necessary.

Product Management Page (Edit listing)



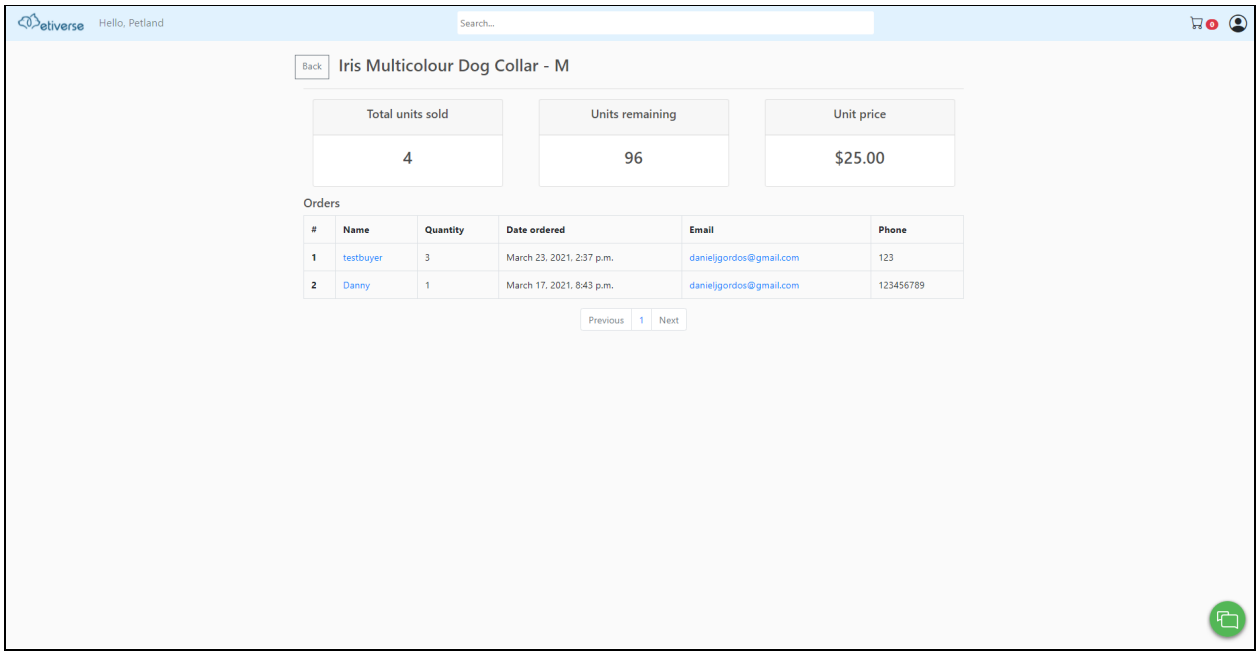
If the user decides to edit the listing, a similar form to the "*New listing*" page appears. Users can then modify any field by making direct changes to the field. Any fields that the user wants to leave unchanged should be left blank. Once all changes are made, the user should press the "*Confirm changes*" button to successfully update the listing.

Product Management Page (Unlisted status)



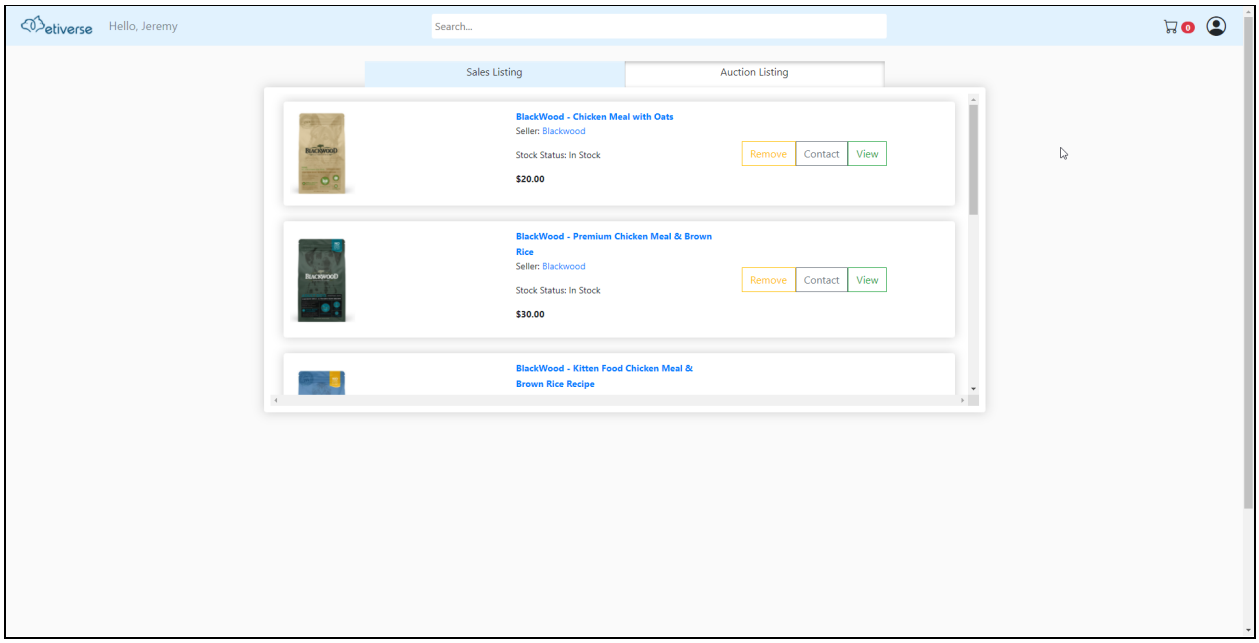
If the user decides to end the listing, they can use the "*Unlist item*" button to take the product of the store. What this does is hides the product from new search queries and prompts an unlisted warning to users that still have the product in their "*recently viewed*", "*purchase history*" and their "*wishlist*". However, by unlisting the item, buyers interested in the product can no longer add the product to their cart and can only view the item.

Product Management Page (View Orders)



To view all the orders a product has, users can use the "*view all orders*" button to access a summary of all the orders that have ever been made on the platform for a product. The page also includes a quick rundown of statistics such as the number of products sold, remaining quantity and the unit price. Under this is an order list where buyers' order details are stored including the quantity each order contains, the date the order was placed and the buyer's contact details.

Wishlist Page (Sales Listing)

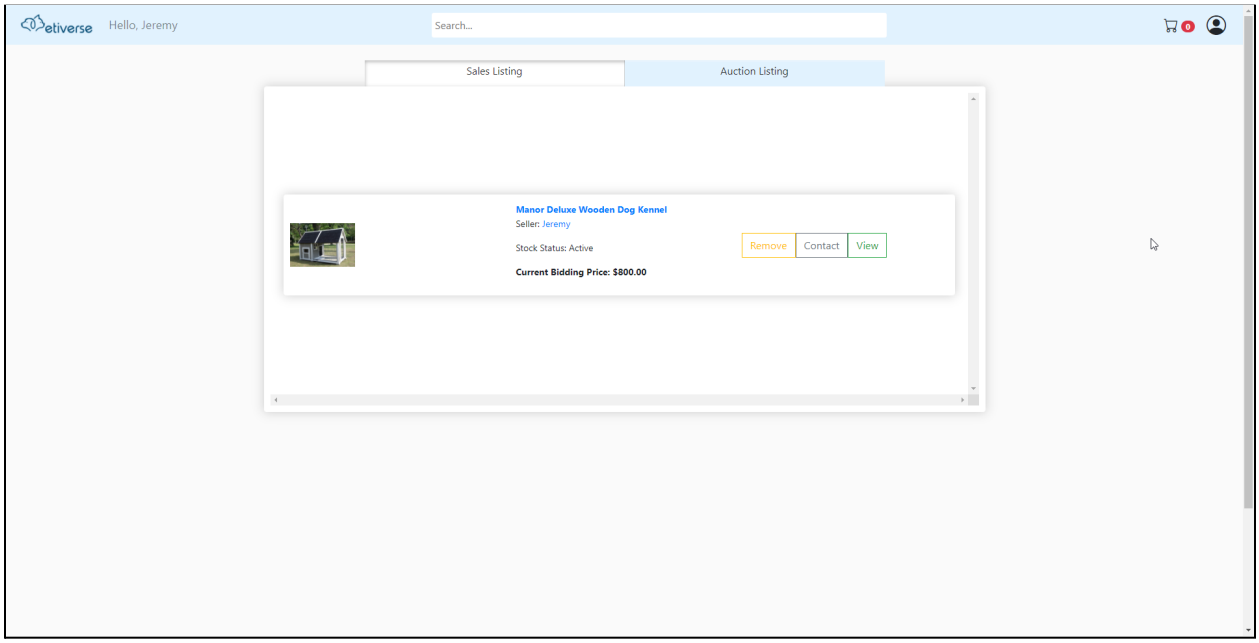


If a user wishes to keep track of products on the platform, they are able to do so by adding them into the wishlist. users can check the product's name, image, seller, price and the stock status. The stock status is a simple prompt that shows whether there are any available units left to sell of the product. If the remaining unit of the product is 0, then the status will appear as "Out of Stock" and if the remaining unit of the product is more than 0, then the status will appear as "In Stock"

If the user is no longer interested in certain items, the user can remove it from the list using the *Remove* button.

A user can also contact the seller through email with the *Contact* button, which will bring up the Mail application that the user has installed in his computer. Another way to contact the user is through our *Messaging* system which can be found in the menu on the top right corner of the website. Using the *View* button will send the user to the product page.

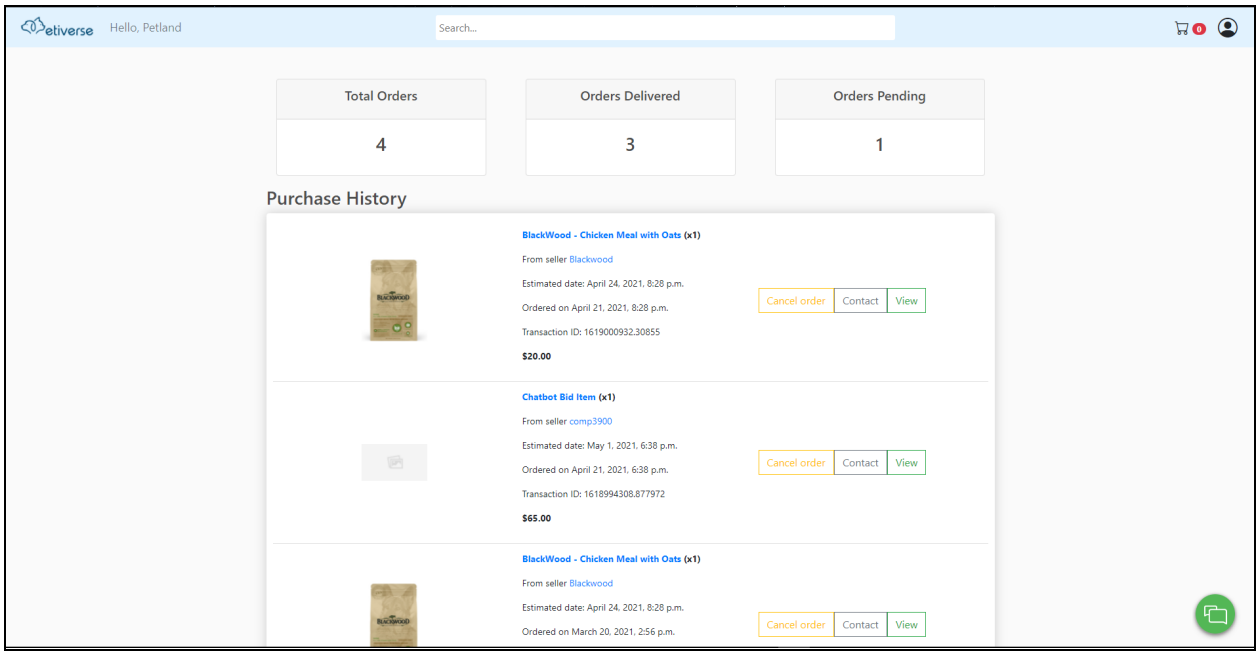
Wishlist Page (Auction Listing)



The wishlist is also separated by auction listings. A user can check the auction products in their wishlist by changing to the *Auction Listing* tab. Which shows an almost similar layout as the sales listing. The only changes being the price and stock status of the product is no longer shown.

Instead, the price has been changed to the current bidding price of the auction product. The stock status here shows the status of the auction and the status will change when the auction has reached its end date.

Purchase History Page



By pressing on the user icon on the top right corner of the website, a menu will show up.

A user can access his purchase history by choosing the *Purchase History* option.

The user can observe his total orders, delivered orders and pending orders on the top of the page.

The *Purchase History* section shows all of the user's orders. It presents the image, name, seller and price of the products. The user can also see the estimated arrival date, ordered date and transaction ID of his orders.

The user will be able to use the transaction ID to reference his order if he has an issue about it.

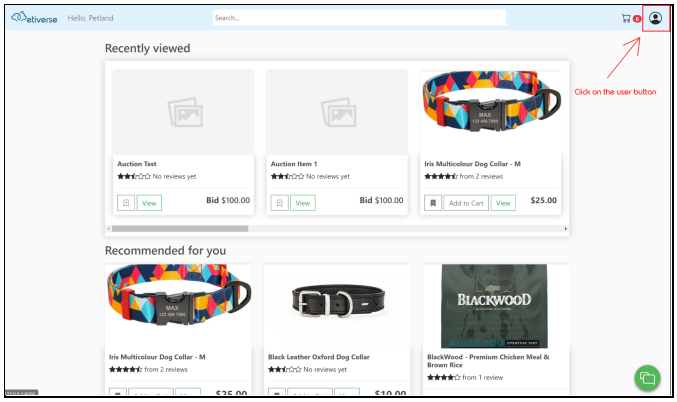
The user has been provided an option to cancel the order if he has any problem. However, if the reason is invalid, a penalty may be imposed to the Users.

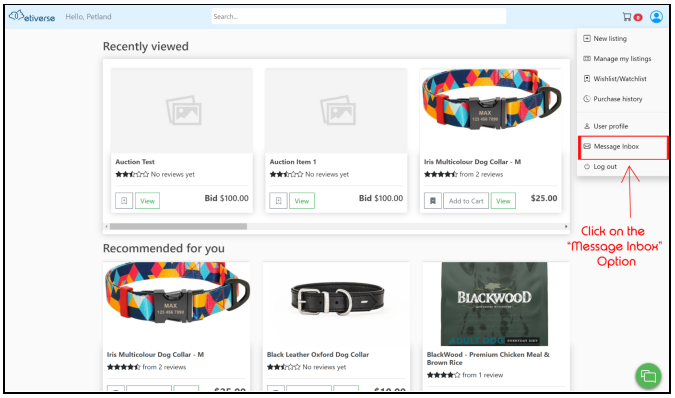
Possible punishments are the buyer's account may get banned from Petiverse or even have to pay an indemnity to the seller.

Cancelled items will disappear from the list after an order has been successfully cancelled.

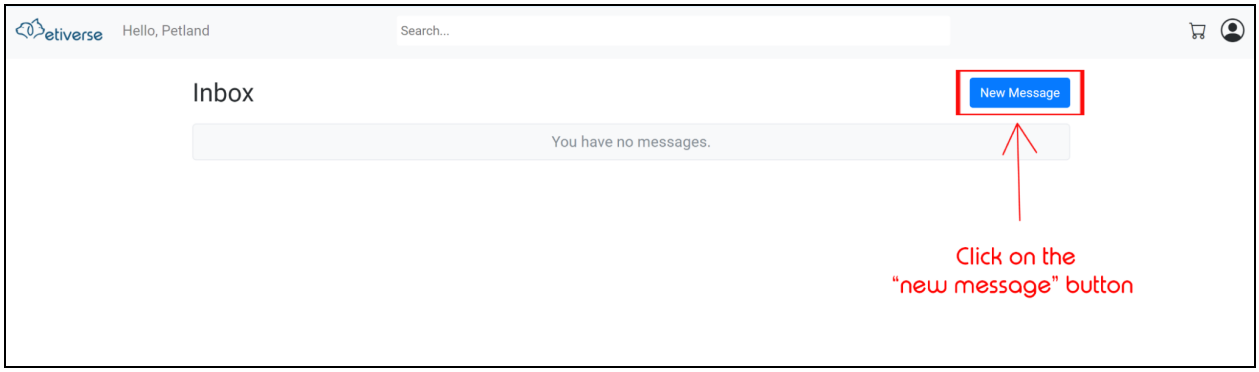
Same as the *Wishlist* page, the user can contact the seller through email with the *Contact* button, and the *View* button will send the user to the product page.

Message Inbox Page



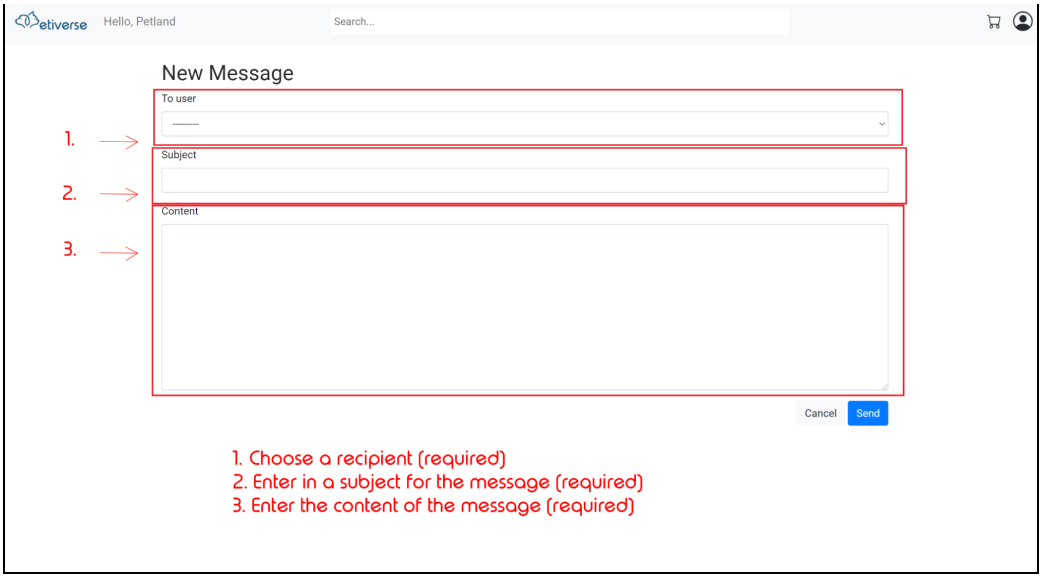


To access the Inbox, the user must go to the user button on the top right and navigate to the “Message Inbox” in the dropdown menu.



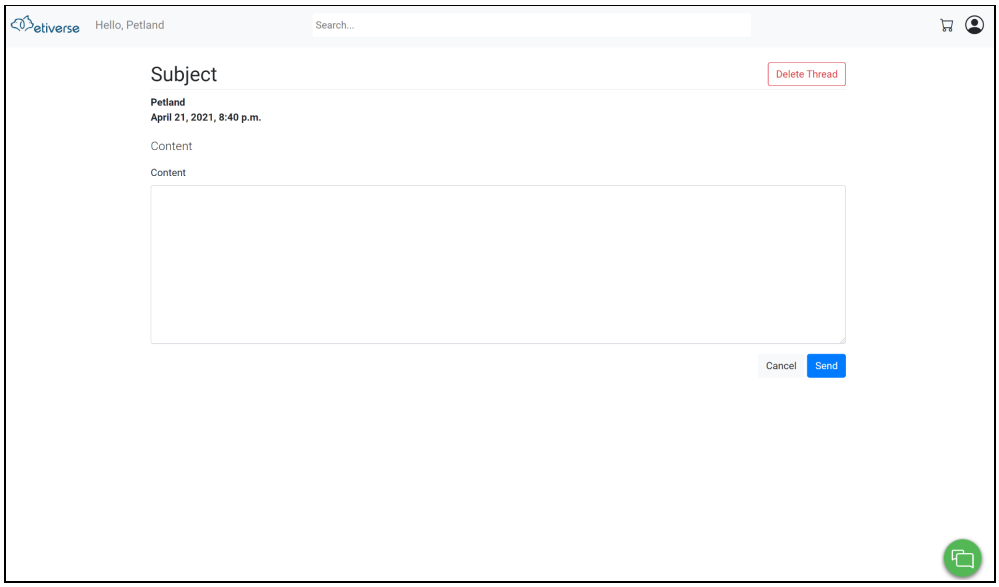
Upon being redirected to the inbox, the user will be faced by either an empty inbox (as shown above) or an inbox that has a message/messages in it. We will first go over the case whereby an inbox is empty.

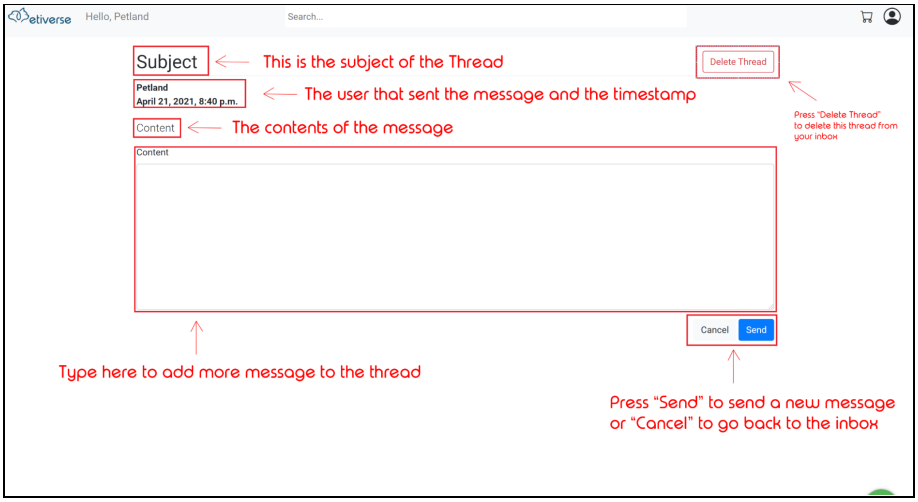
If the inbox is empty, the user will be met by a “You have no messages” prompt. In this case, there is nothing the user can do except redirect to the other pages or compose a new message. To compose a new message, the user must first press on the “New Message” button where they then will be redirected to a form where they will be allowed to compose a new message as shown below.



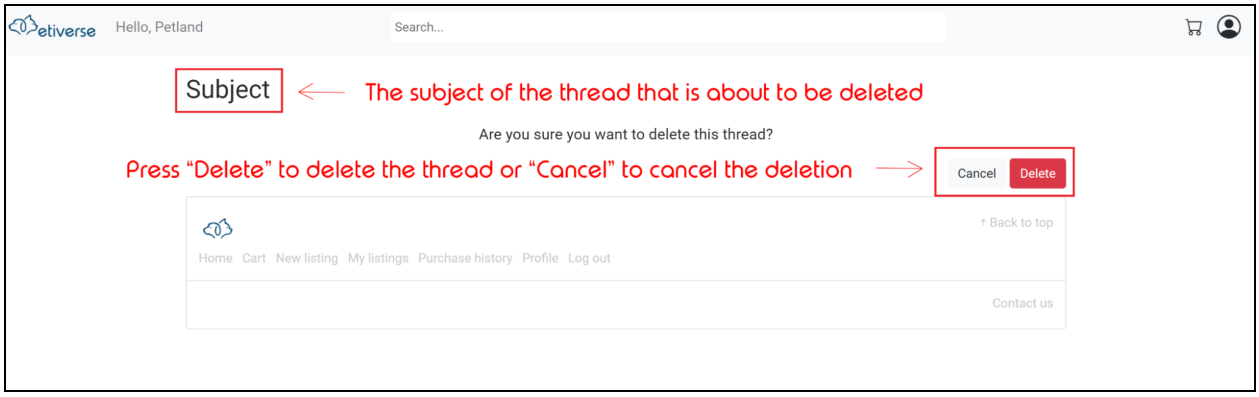
Now that the user has been redirected, to compose a new message, a user must first choose a designated recipient. Next a subject for the message and the content of the message must be entered. Once the user is satisfied with the response they have inputted, the user can press the “Send” button to send the message to the designated user or press “Cancel” whereby the message they have composed would not be sent and discarded and the user will be redirected to the inbox page.

If the user does decide to send the message, they will be redirected to the page as shown below.

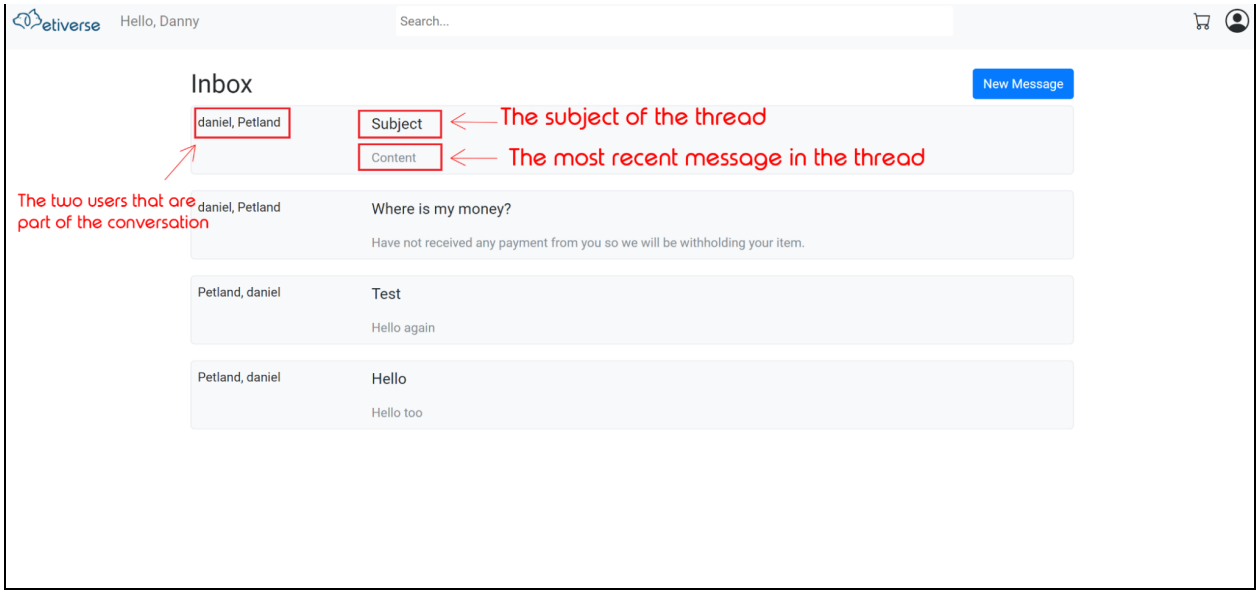




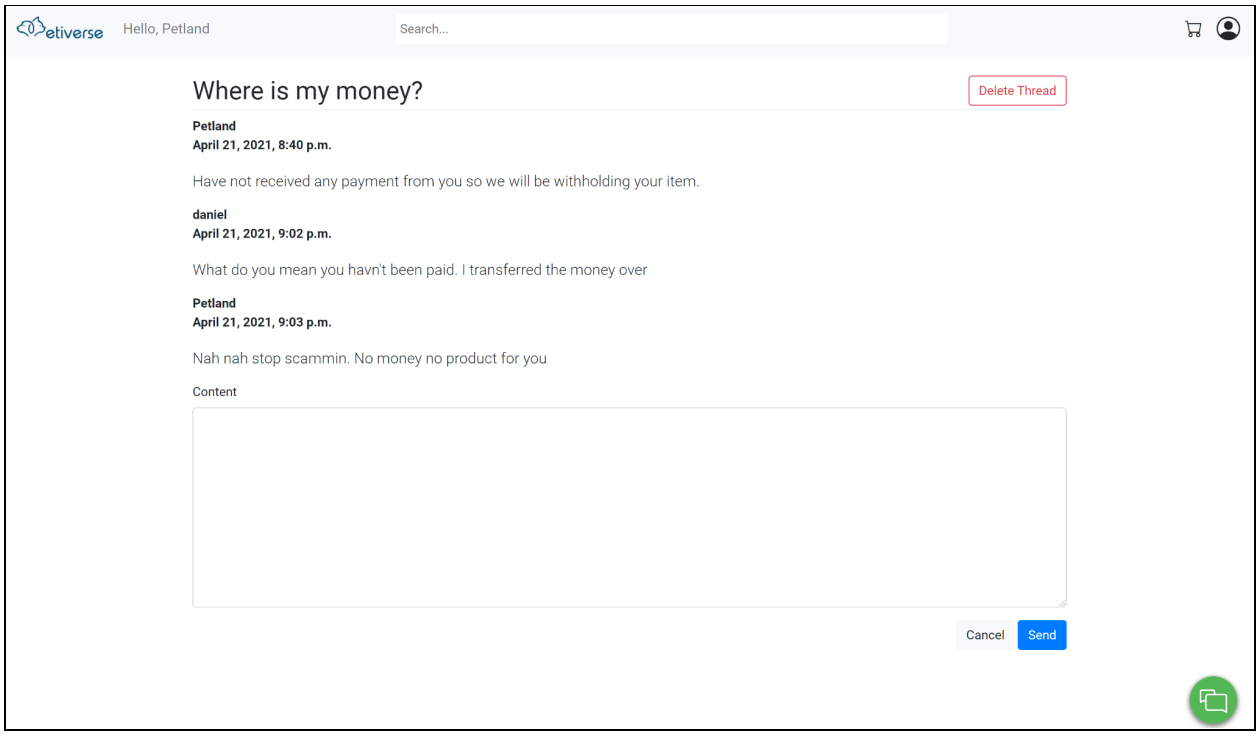
Currently, what the user is viewing is the thread that was created between the user and the designated receiver of the message that the user has composed. The layout of the page is outlined above. From here, the user has the option to add more messages to the thread by typing in the "Content" field and pressing the "Send" button. Or the user has the option to press "Cancel" which will redirect the user back to the Inbox page. If the user does go back to the Inbox page, the message that the user has written will not appear on the inbox as it only detects messages coming into the inbox and does not recognise any of the sent messages. The message still goes through to the intended recipient so this functionality works just fine.



Last but not least, the user from the thread page above also has the option to delete the whole thread by pressing on the "Delete thread" button. If pressed, the user is redirected to a confirmation page that includes the name of the thread wished to be deleted on the top left. If the user confirms the deletion of the thread, any record of the thread will be deleted from the user's inbox. This however does not delete the thread on the recipient's inbox and they will still have a record of the thread.



Next we will go over the case where the user has a message/messages in their inbox. As pictured above, the different threads will be listed and each thread bubble contains the name of the two users that are engaged in the thread. It also includes the subject of the thread and the most recent message that appears in the thread. If a user presses on one of the thread bubbles, they will be redirected to that particular thread's page whereby the functionality follows the previous explanation. An example of a more populated thread that includes more messages is shown below.

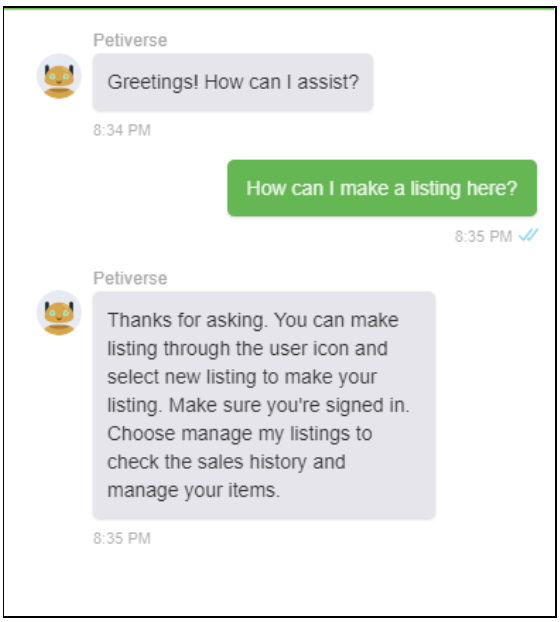


Chatbot

For some new listings to display in the chatbot, admins of the platform are required to take some additional actions in DialogFlow or the admin site. Please refer to the instruction to the “Additional Guides” section.

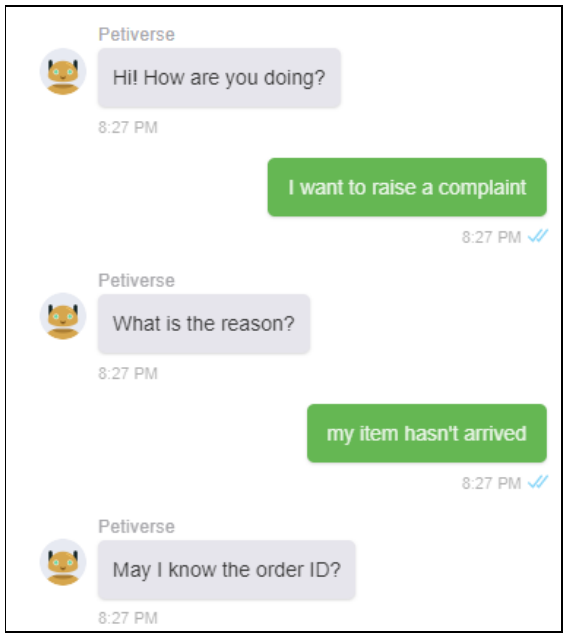
Selling instruction

As a user(Buyer/Seller), the Chatbot can help new users on the procedure to create a new listing and on how to manage their products on the platform. This can be triggered by the user by asking questions along the lines of “How can I make a listing here” on the chatbot window. A sample conversation is shown on the right.



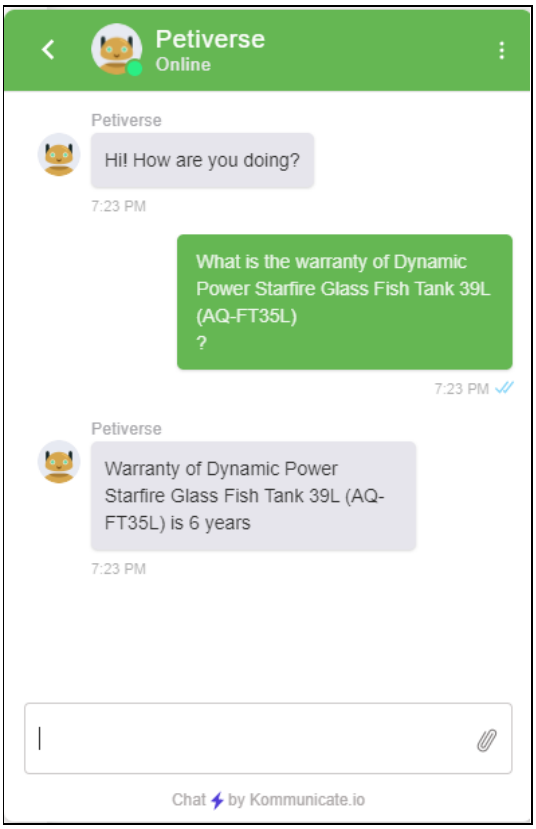
Complaint helper

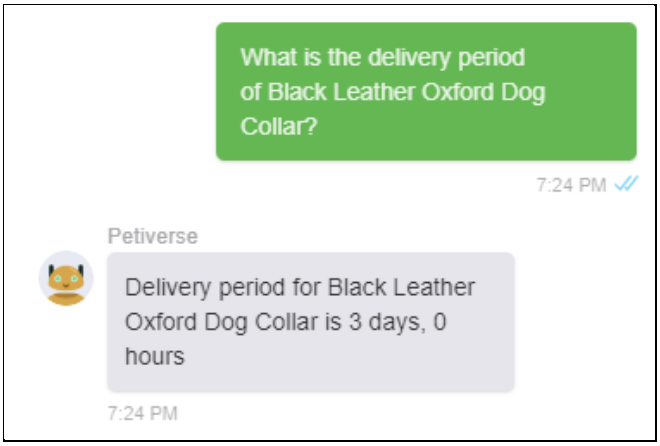
The chatbot is also able to raise a complaint for users as shown on the right. This is done by the user asking a question along the lines of “raising a complaint” in which the chatbot will reply by asking for the reason of the complaint. Once the user inputs in a proper response, the chatbot will reply by asking for the order ID. The user can input in the transaction ID. To complete the process, the chat bot will require a valid reason and an order ID. All valid reasons can be viewed in DialogFlow entity set named ‘complaint_reason’ and order ID can be any integer number.

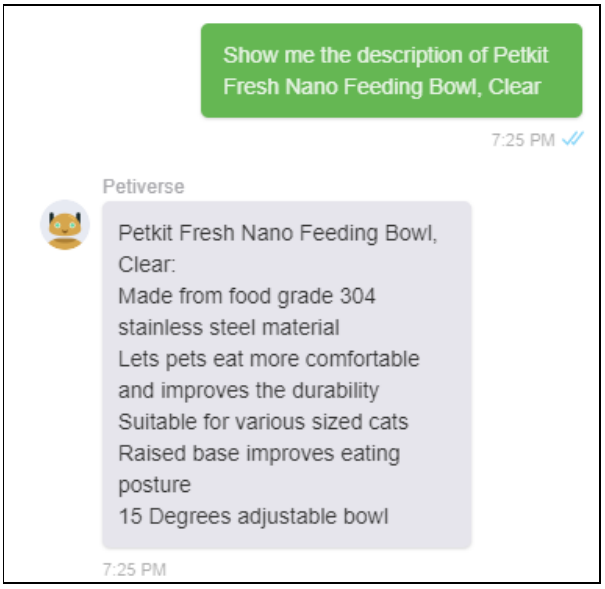


Product enquiry

If a user wishes to know more about a product, the chatbot can answer a series of questions about the product such as warranty, delivery period and description as shown by the following pictures. Users can simply ask the chatbot a question along the lines of "What is the <information that the user wants to know> of <Product name>" and the chatbot should return that section of the product information. Product names have to be exact so a simple copy and paste of the product name into the chatbot window will suffice.





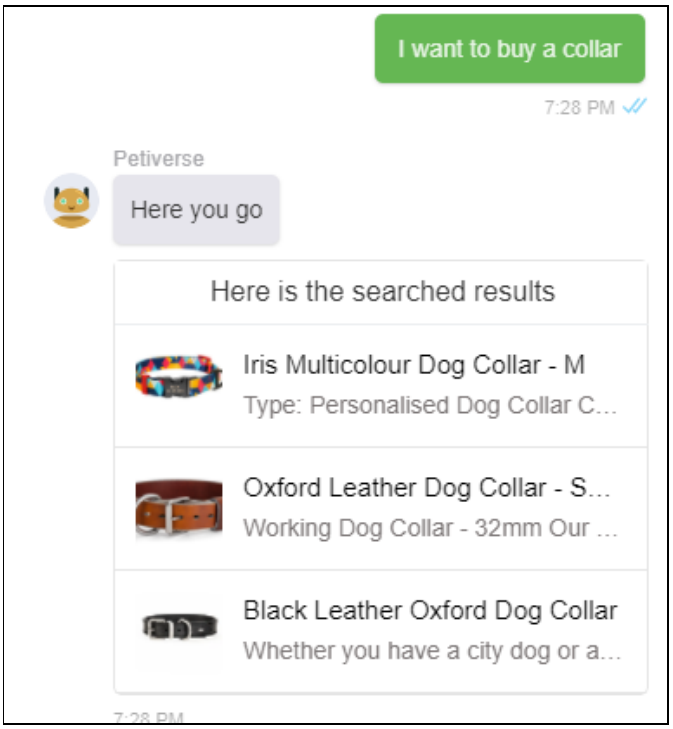


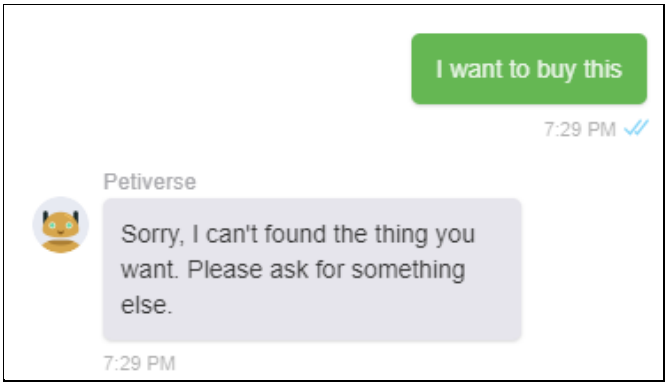
Case:

A new collar listing has been created by a seller and the website admin wants to allow other users to enquiry the information about the new listing.

Solution: Please add the full name of the new listing into the corresponding entity set (In this case will be collar). Refer to "Add a phrase into an entity set (DialogFlow)"

Product searching





A user can also search for products through the chatbot by asking a question along the lines of “I want to buy a <Item name/tag>”. This will trigger the chatbot into recommending a list of products based on the user interest as shown in the figure to the right. Users can click any of these in the list and that will open a new tab for the product page to allow users to perform the further action. The chatbot will determine the keyword and match it with all existing tags and search for all the relevant products.

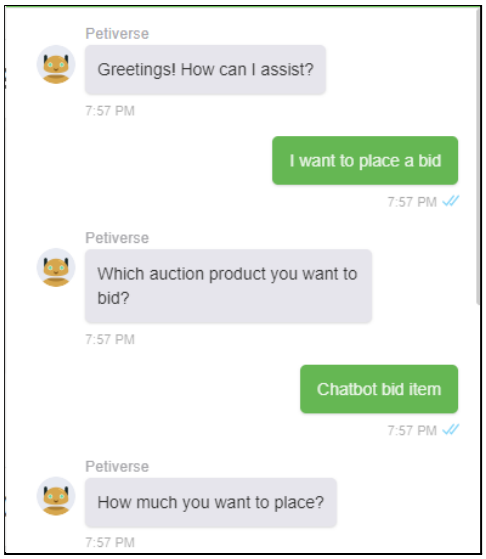
Sometimes, the chatbot is unable to determine the keyword and returns an apology message as shown to the right. User’s should repeat the question using a registered keyword to get a proper response.

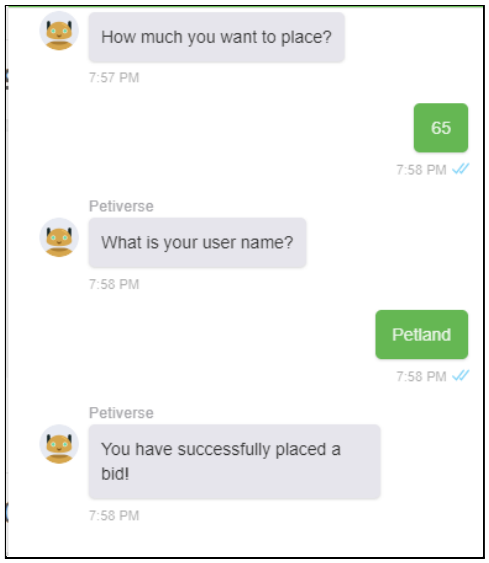
Case: A new collar has been listed out by the seller and the photo of it also included but the chatbot has not shown the image of the product in the recommended list.

Solution: This is because the data uri is required for displaying the image in the chatbot recommended list. Thus, adding a dataURI will solve the problem. Please refer to "Adding dataURI to ImageUri".

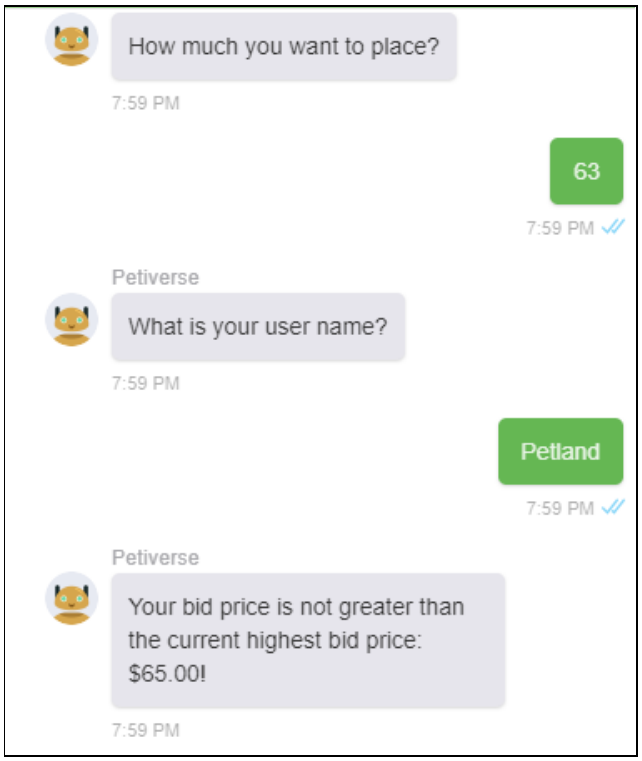
Place bid

Chatbot can also help users place bids for the auction product. This will take three values to work. The chatbot needs the name of the auction product, the bid price and the account username. Chatbot will perform further checking to ensure all the values are valid. The success example will be like below:

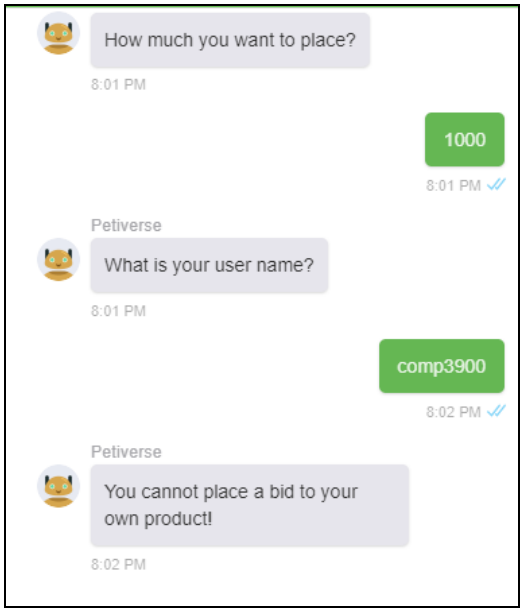




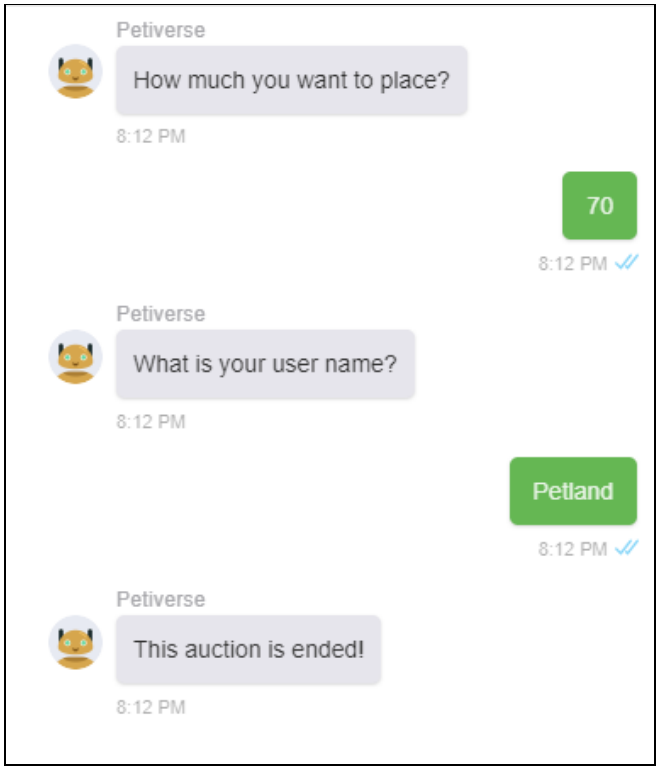
If a user enters a bid price lower than the highest bid, the chabot will register this as an invalid bid price and it will be rejected by the chatbot as below.



If a user tries to bid on a product listing of their own creation, the chatbot registers this as an invalid username. The chatbot will reject the request to bid by replying with a warning prompt as shown below.



One final case will be if the auction the user requested is ended so no more bidding will be accepted as shown below. The bid will be rejected and will not appear in the bid history of the product.



Currently, most of the existing users in the database are able to place bids through chatbot so for the new user and some excluded users need to be added to the DialogFlow entity set called 'customer_name'. Also there is only one auction product that can be bidded through chatbot at the moment which is Chatbot Bid Item, therefore for any new auction listing that wants to utilise this feature also needs to be added into the 'auction_product' entity set.

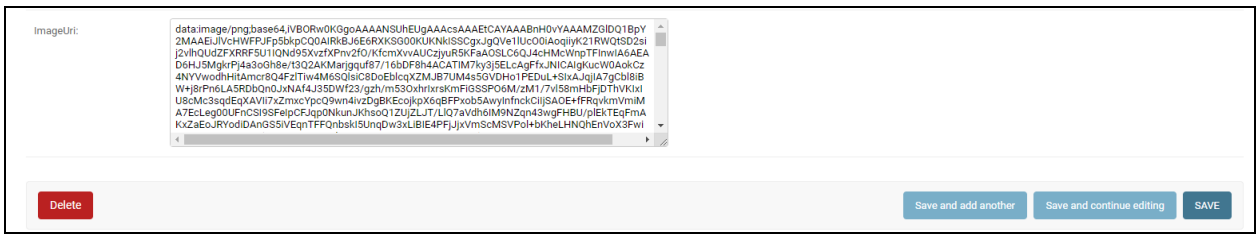
Please refer to 'Add a phrase into an entity set (DialogFlow)'.

If the chatbot bid item auction is ended, please go to the admin panel http://127.0.0.1:8000/admin/ and sign in with any admin account (e.g. username: admin ; password: Unsw2021). Go to 'Products', select 'Chatbot Bid Item', reset a new end date and save. Then, head to http://127.0.0.1:8000, click the user icon and select 'manage my listing' to relist 'Chatbot Bid Item'.

Additional guide

Adding dataURI to ImageUri

Please go to the admin panel http://127.0.0.1:8000/admin/ and sign in with any admin account (e.g. username: admin ; password: Unsw2021). After signed in, select 'Products' and select the name of the listing,then, fill up the 'ImageUri' section and like below:



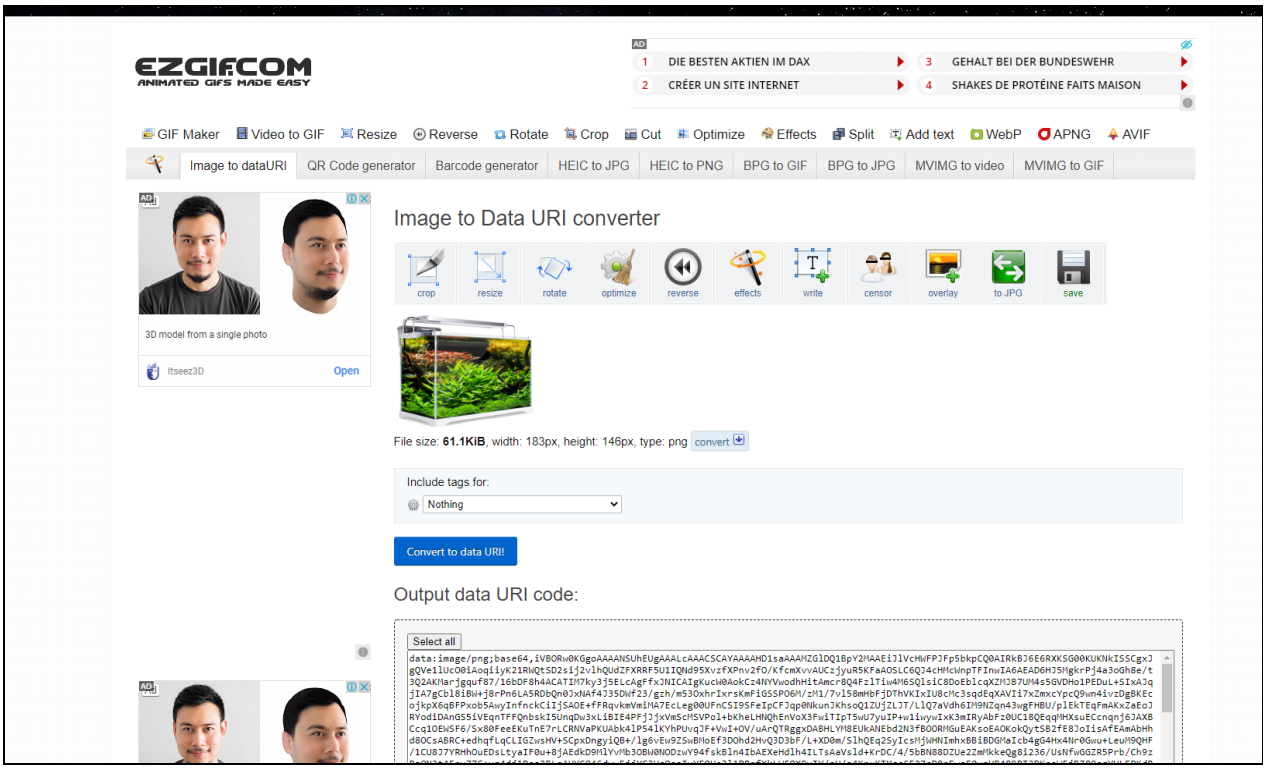
Press 'SAVE' to store it into the database and you will be redirected to the 'Products' list page.

Here is the recommended dataURI generator and resize tool

https://ezgif.com/image-to-datauri

The recommended size for an image is less than 100KiB because any value larger than that will gradually slow down the chatbot processing speed which induces the chatbot cannot display the recommendations properly.

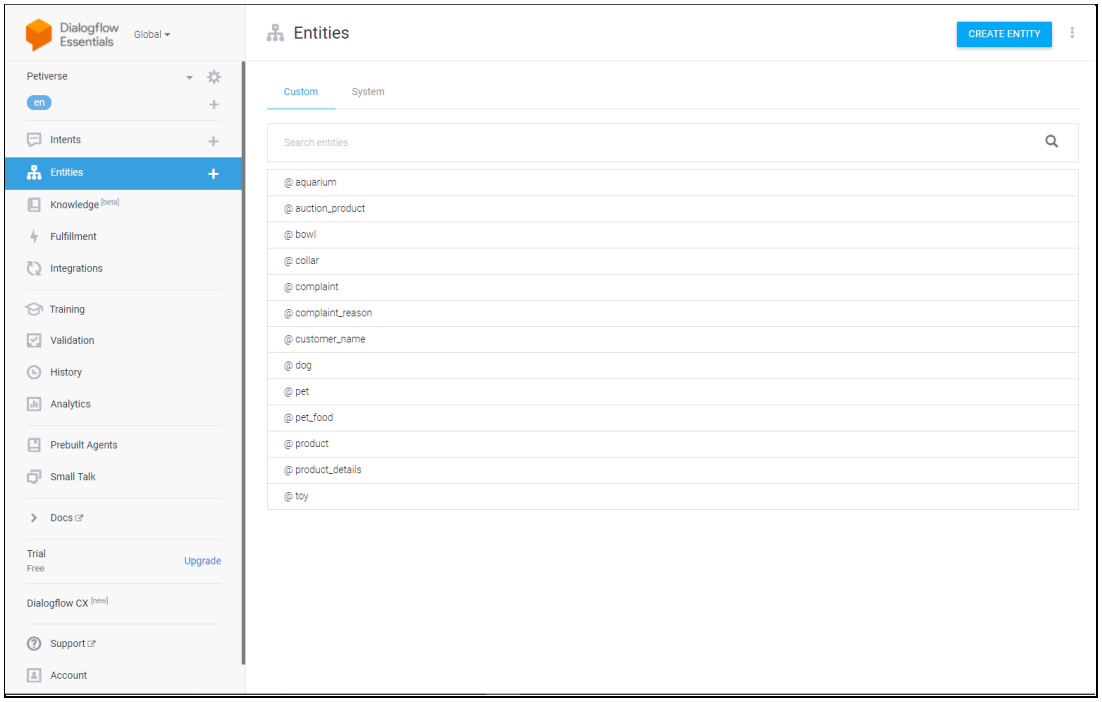
Thus, the suggestion will be to resize the image size percentage to 50% or lower and then save it and use the resized image to generate the dataURI.

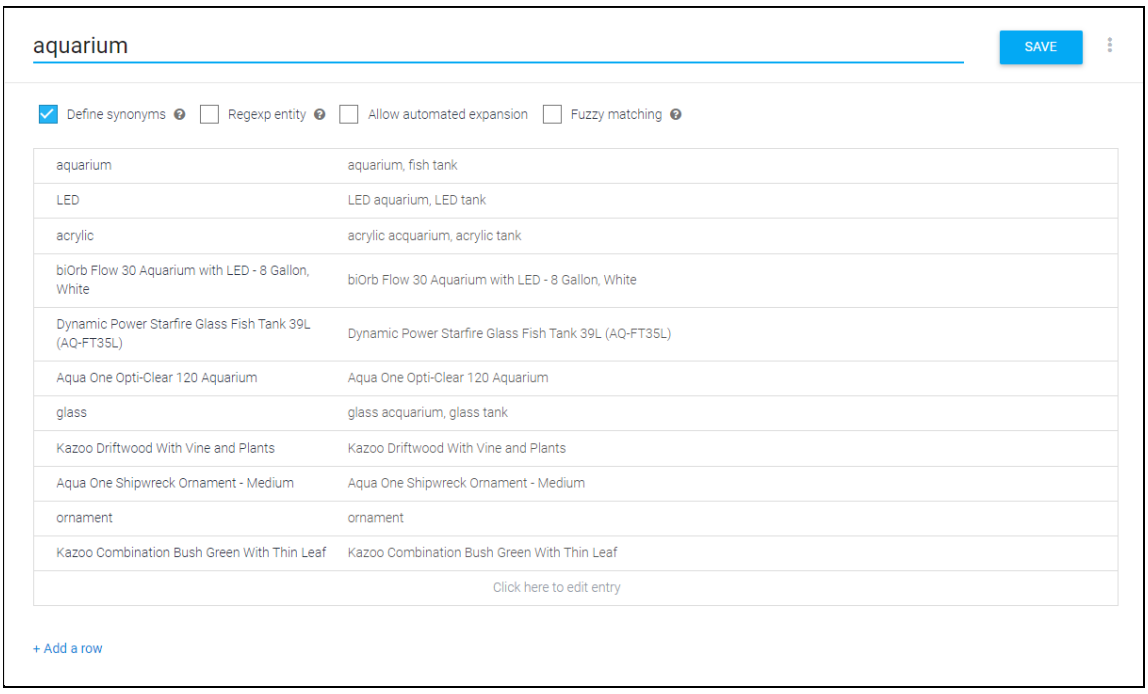


Tips: select ‘Nothing’ for ‘Include tags for:’ section, then, use ‘select all’ to copy the dataURI text and paste it into the product ImageUri section. All the images are stored in ecommerce/static/images/. A sample is shown on the top.

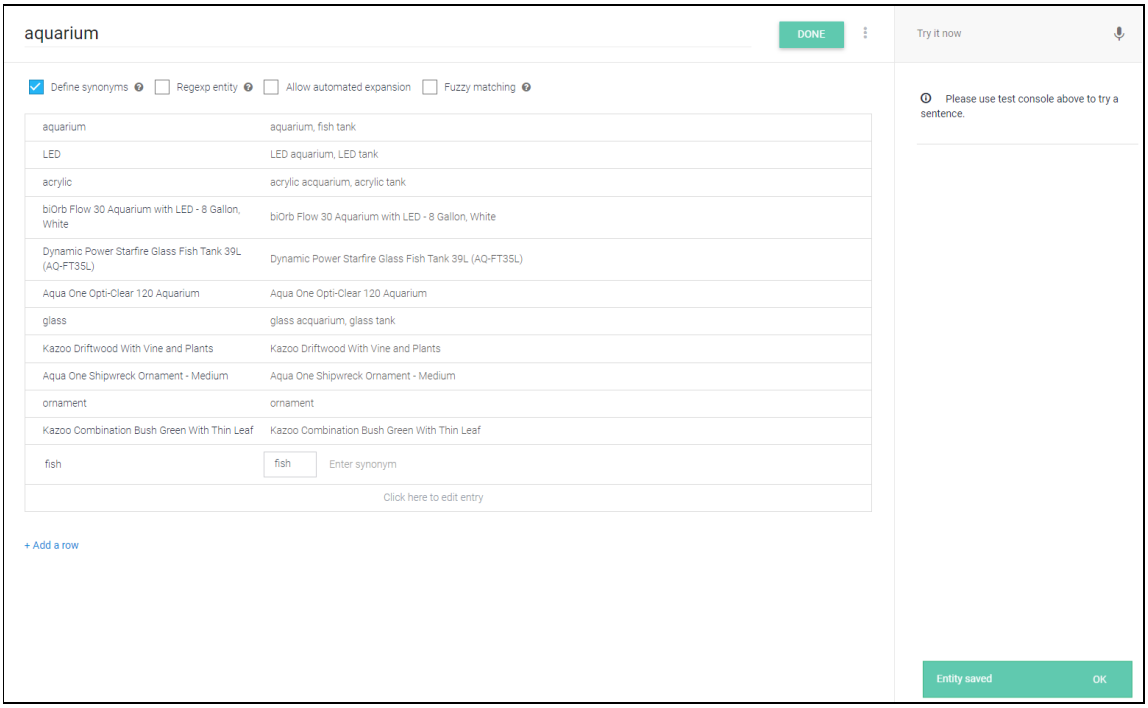
Add a phrase into an entity set (DialogFlow):

1. Select the entity set you want to modify. (Here we selected the aquarium entity set)





2. Enter the new value you want to add in the last row and enter a few synonyms for that and press ‘Save’ (Example shown has entered fish in the last row). After saving it, the ‘Entity Saved’ notification will pop up at the bottom-right corner. This means the new phrase has been added successfully.



Implementation Challenges

Website Layout

All members of our group were inexperienced with frontend web design, and thus a large amount of time (especially within the first 2 sprints) was spent making small adjustments to page styling, in an attempt to make site pages reflect our original wireframe designs more closely. While we believe the final site has a fairly clean and modern visual design, it relies heavily on absolute (rather than relative) units for the positioning and sizing of elements. Thus, page layouts sometimes react poorly to unusual screen resolutions and aspect ratios. If we had more time, (or more existing experience with web development), it would have been worthwhile to rewrite the site's styling with responsive design in mind.

We heavily utilised the Bootstrap library, which provided easier access to modern styling and many readymade UI elements. However, Bootstrap's own system for positioning and sizing elements added to the learning curve of styling pages as envisioned.

User Authentication

Since all of the developers do not have much experience with user authentication, we thought we will spend much time on implementing this feature. Luckily, *Django* comes with a built in user authentication system. *Django* has saved us immeasurable time in this case due to the reason that It handles user accounts, groups, permissions and session-based user authentication with its built-in 'User' class.

Recommendation System

With the current implementation of the recommendation system, whenever a user requests the main store page, their 'profile vector' (which encodes their product preferences using their history) is built from scratch. While this system works fairly well for our small test database, it would scale poorly with a 'real world' system, as constructing the vector has complexity O(n) in the number of listings they have viewed across their whole history. We opted for this system as the alternative - storing a profile vector for each user and only updating it when user activity is logged - raised problems. It greatly increased the complexity of ensuring that user profile vectors and other elements of the site database stayed synchronised, for example when a product had its tags changed or was deleted.

Additionally, with our implementation of the recommendation system, the user profile vector is compared to each product listing with the cosine similarity metric, another slow O(n) operation. While not improving its time complexity, this operation could have been significantly sped up by computing cosine similarity using a C based mathematical library

such as *Numpy*. This change was not implemented due to time constraints, and our current solution - implementing the user profile vector and cosine similarity operation using Python dictionaries (which are internally hash tables), proved fast enough.

While our recommendation system provides fairly accurate recommendations to logged in users who have a history of viewed and purchased products, for new or guest users it falls back on using only review ratings for product ranking. This unfortunately leads to recommendations for new users being somewhat unhelpful, as for example all products with no reviews (regardless of number of purchases or views) are ranked equally. Fully solving this problem may have necessitated using a significantly different social-based recommendation system than the tag-based (content-based) system we chose.

Auction

One problem of the auction features was the date time picker when a user is attempting to create a new auction listing. At first we were trying to use the date time picker in *Bootstrap*, somehow it worked independently but the date time picker does not show up when we have included it in our code.

Luckily, the model field reference of *django* has included a DateTimeField. DateTimeField is just date and time, represented in Python by a datetime.datetime instance. This field takes the same extra arguments as DateField so it helps to build up a date time picker by ourselves. An HTML input element is represented as a widget in *django*. The widget handles the rendering of the HTML, and the extraction of data from a GET/POST dictionary that corresponds to the widget. By adding one of the existing widgets, DateTimeInput, into the form that creates a new auction listing, *django* does all the work for us and display the date time picker that we expected to see.

Another problem would be the auction timer. Some developers were testing out this feature by creating a new auction listing and extending or reducing the end date of the auction to see the changes of the timer. However, we have realised the timer is not accurate that it is always around 30 mins ahead of the actual end date time. After running several test cases, the reason for this weird behaviour has been found, which is because of the time zone difference.

The concept of the auction timer is simply subtracting the current date time from the auction end date time. When the difference is larger than zero, then it will keep updating the remaining time. Whereas the timer will display “Auction Ended” when the difference is less than or equal to zero. We have realised the current date time created by using the *python* library, does not contain the timezone information. In contrast, the end date time, created by using the *django* library, does contain the timezone information by default. Therefore, we have solved this issue by removing the timezone (tzinfo) from the end date time object.

A developer has found an issue about the layout in the auction product page. He mentioned that he could not see all 5 recent bidders under the bid history section of the auction product page whereas another developer claimed that he could see all of them. After some discussion, this problem happened with the resolution difference between the developers. A developer has a lower resolution screen and another developer has a higher resolution screen which presents them a different layout of the page. Since this bid history section position was shifted right by the number of pixels, changing this shift that it scales percentage has fixed the position layout issue. In order to view all 5 recent bidders, we have added a scroll bar so even a user with a small screen and low resolution can observe every bidders.

Another issue was a developer claimed that he could not place a bid to the auction product while it works perfectly fine to the rest of the developers. The reason has been found that the developer which had a problem placing bids was using *Firefox* as his browser. This problem has been fixed by accessing *Petiverse* with other browsers like *Safari* and *Chrome*. Although we could not find an actual reason for this trouble, we hypothesise that *Firefox* will block some certain requests by default and a user may not place a bid because our server could not receive the blocked requests.

Email Feature

Although some of us have experience handling emails with *Python*, *Django* provides a better mail sending interface. *Django* sends email quicker, helps test email sending during development, and provides support for platforms that cannot use SMTP.

The problem is we did not implement sending email as a background process due to the shortage of time. Therefore, it usually takes a few seconds to load when an email is being sent on certain pages like the email registration and password reset features. The loading times get a lot worse when it comes to checking out a large number of items because the system will have to send out many emails to the buyer and sellers of all products.

The biggest problem of the email sending feature appears when it comes to the auction feature. We designed that our system will automatically send an email to both the winner and the seller of the auction when an auction has ended. If we set up an event listener in the html, then the email will only be sent to the winner and seller when someone has accessed the corresponding auction product page after the auction has ended. Which is not a good idea because the email will never be sent if no users have accessed the product page after the auction has ended.

We have thought of a better idea of implementation, which would be adding an event listener to the Product class in *models.py* so the system will send the email when it signals the end of the auction. Although we have spent a significant amount of time to approach

this hypothesis, it was too complicated that we always come out with errors that we have never encountered.

Unfortunately, in order to complete other features on time, we could only choose an inferior, but much simpler way to implement this email sending feature. We have decided to use the threading module in *python*, which allows the system to run multiple threads (tasks, function calls) at the same time.

We have implemented a function that checks the end date and time of every auction product. If any auction products have reached its end date then this function will send an email to the highest bidder and seller of the product that is stored in the Product class of *models.py*.

By using the threading module, we initially have set a threading timer which loops this function every 5 seconds. Although this implementation works fine, if we have a large number of auction products in the database, the system may end up spending more than 5 seconds to loop through. Before the end of the current thread of the function, another thread would probably start over which may lead to unknown error. The solution that we have made is just increasing the threading timer from 5 seconds to 15 seconds, which is still not an ideal solution unfortunately.

Lastly, one of the serious problems of using thread in the operating system is that all the local and global variables are both shared between threads. This creates a security issue as the global variables give access to any process in the system. Another problem would be the execution of the function via threads is time-consuming. It is because threads depend on the system and the process to run.

Chatbot

The implementation challenges we faced on the in-built chatbot for the website were the connection difficulties with DialogFlow and the usage of the Actionable Messages from Kommunicate. We were having a series of issues by following the Google Developers Team guide video to test out the DialogFlow mechanism. The issues such as the provided html template not working as the video shown and the DialogFlow was unable to be connected to our system. After several trials, we ended up using DialogFlow webhook feature to establish the connection and utilise Kommunicate to assist us rendering the chatbot interface. The webhook functionality allowed us to have stable, reliable and fast connection and DialogFlow is able to detect the user intention and parse the parameters to our backend instantly.

Another difficulty we encountered was the Actionable Messages layout problem. We spent some time troubleshooting the rendering issue and we realised that the sample fulfilment template has some minor issues on its layout. Therefore, we did some modifications to the fulfilment messages and eventually everything worked as expected.

In addition, we also found a shortcoming of Kommunicate which is Kommunicate unable to handle multiple large dataURIs for list of images. For instance, to display a list of items on chatbot we are required to include dataURI for each image but the original size of the image is greater than 100Kib the generated dataUri will be gigantic and Kommunicate will be overwhelmed. We found this issue was related to Kommunicate and neither DialogFlow nor our issue because we inspected the outputs from our backend and DialogFlow were generated immediately. The solution for this was we had to resize each image to less than 100Kib and used the resized images to generate the dataURIs. This strongly increased the processing speed of Kommunicate to display a list of products.

Pinax Messages

The biggest challenge with doing the messaging system was that there were little to no APIs out there that was optimised for web deployment. Most of the APIs we encountered were built mostly for mobile deployment. The problem also was we never had any experience building a user to user chat or messaging system before hence finding resources on the topic was especially hard. Many resources online only provided help on creating a public chat room. Seeing that this was not what we wanted to implement, we researched other methods of implementing this system and that was when we came across a collection of open-source projects called *Pinax*.

Pinax had a variety of open source projects that were implemented using the Django framework. Which was perfect considering our platform was also relying on the same framework. This made integrating *pinax-messages (their open-source messaging system)* very easy work. However, a problem that came up during installation was that the html templates would not render properly. This was due to the fact that pinax messages was quite reliant on other pinax projects. One such project is called the *pinax-templates.* This is where pinax stores all their templates for all their projects. For some reason this was not working and we pinned it down to pinax missing their .css files. Luckily, Pinax also has a website showcasing their templates online to show what they really needed to look like. A trick that we ended up using was to inspect the page of the respective html pages we needed and downloaded their css files. This also came with it's set of problems as the css files where minified into a MIME file and google chrome refuses to implement it. A simple workaround we came up with was to merge their css files with our main css files.

Another problem was the fact that the html templates for *pinax-messages* needed to be updated for them to be able to render properly using our css file. Because we had set up a virtual environment for our project, any packages installed would end up there. Meaning that a fresh install of pinax-messages would not work right off the bat. After scouring the pinax slack forums. A user replied to us by saying that we simply needed to have a pinax subfolder in our templates directory that holds all the new html files. And this solution fixed our problem perfectly.

Group Collaboration

One of the main problems that we have always encountered is merge conflict. Since we have divided the work into several features among developers, each of us has a different version of code.

One common situation is a developer has finished his work but when he was about to merge into main, he realised that another developer has already merged his code. Unfortunately, the developer's code is a few commits behind the master branch, git usually cannot merge automatically in this case, so the developer could only merge manually.

The common way to merge manually is to copy and paste the developer's new changes to the newest version of master branch. However, developers often have forgotten to copy and paste some part of his codes into master, which causes some features not working properly as he expected.

Even if the manual merge works perfectly fine, there are always database and migration files conflicts between different developers. The initial problem is a developer has created new fields in the class within the *models.py* which require the user to migrate the file before running the server. However, we have encountered the same problem many times that errors occur due to some missing data for the new added fields in the master branch.

For example, a developer has added a new *WIshlist* class that connects with *Customer* and *Product* class. Although there aren't any mistakes in the code, users have to create a wishlist for existing customers and products in the database or else errors may occur due to the non-existence of wishlist. Which is the reason why a developer has to upload his database and migrations files if he has made some changes in *models.py* that relates to the database.

References

Alake, R. (2020, September 15). *Understanding Cosine Similarity and its Applications*. Retrieved from:

https://towardsdatascience.com/understanding-cosine-similarity-and-its-application-fd42f585296a on March 5th 2021

Bootstrap (2021), *Bootstrap*. Retrieved from:

https://getbootstrap.com/docs/5.0/getting-started on April 21st 2021

Django Software Foundation (2021), *Django*. Retrieved from:

https://www.djangoproject.com/ on April 21 2021

Google (2021), *Google DialogFlow*. Retrieved from:

https://cloud.google.com/dialogflow/docs on April 22nd 2021

Google *Developers*. (2020, November 18). Retrieved from Google APIs Terms of Service: https://developers.google.com/terms/ on April 23, 2021

JS Foundation (2006, August 26). *jQuery*. Retrieved from: https://jquery.com/ on April 23, 2021

Kommunicate. (2021, January 9). Retrieved from KOMMUNICATE TERMS OF SERVICE: https://www.kommunicate.io/terms on April 23, 2021

Kommunicate. (2021). Retrieved from Actionable Messages: https://docs.kommunicate.io/docs/actionable-messages on April 23, 2021

Ngrok. (2021). Retrieved from Documentation: https://ngrok.com/docs

Shrivastava, P. (2018, May 11). *Kommunicate*. Retrieved from:

https://www.kommunicate.io/blog/integrate-bot-using-dialogflow-in-kommunicate/ on April 22, 2021

SQLite Consortium (2021), *SQLite*. Retrieved from:

https://www.sqlite.org/about.html/ on April 21 2021