Criterion C: Development

Techniques used

- Object oriented programming (OOP)
- Templating language methods (Jinja2)
- Database management and querying
- External API and library access
- Handling and storing input files
- Constructing input forms with specific criteria and validators
- Routes and redirecting of user
- Efficient combination of languages such as Python, Javascript and Sass
- Algorithmic thinking

The below sections include code that illuminate many of these techniques, and how they were implemented in the project.

Sorting and filtering processes

The flowchart for the sorting and filtering process of the items on the home page is shown in figure 5, meeting success criteria 5 and 6.

When the user first visits the site, the default sorted view of the items is “new to old”. However, applying the sorting occasionally failed when there were zero or one items. To account for this, the following code can be used:

```
if form.validate_on_submit():  # True when search form is submitted
   ##### Search and filter process here #####
else: # Runs every time page is refreshed or user first visits page
   # Handles exceptions if there are no items or only one item in store (in which case sorting will not happen)
   try:
       items = sort_high_low(Item.query.all(), len(Item.query.all()), "published_date")
   except:
       items = Item.query.all()  # Don't perform sorting

# Renders template with the appropriate pre-processed variables
return render_template("main.html", google=google, users=users, items=items, form=form, category=category,
                       current_date=datetime.now())
```

Here, the try/except error handling runs the sorting by New-Old if the algorithm does not raise an exception. If it fails to run (e.g. there are 0 or 1 posts), the program simply queries all the items in the database.

The code below uses a user-inputted search term and finds matching database entries:

```
search = form.search_term.data  # Gets search term

# Queries NAMES of items that include search term returns list
s_items_name = Item.query.filter(Item.name.contains(search)).all()

# Queries DESCRIPTIONS of items that includes search term, returns list
s_items_description = Item.query.filter(Item.description.contains(search)).all()

# Creates new list, including all items from s_items_name and all items from s_items_descriptions that are not already there
items = s_items_name + [i for i in s_items_description if i not in s_items_name]
```

The algorithm searches for the input term in both the “title” and “description” attributes of the items. Because duplicates may arise, the last line concatenates all the searched item “names” list with the searched item “description” list, while excluding elements that are already stored (thus eliminating duplicates).

This filtering/sorting functionality was implemented on the front end as shown in figure 7:

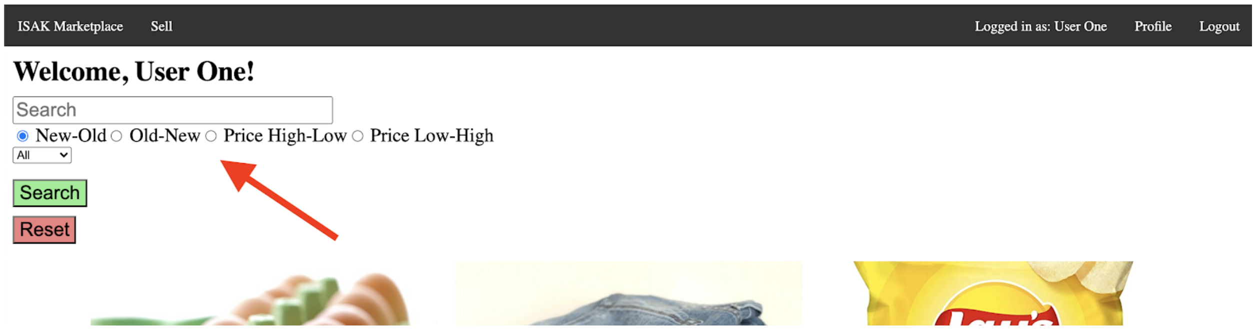


Figure 7: Screenshot of the website, with the red arrow highlighting the sorting and filtering capabilities, both with radio buttons (for sorting) and dropdown category menu below that (for filtering).

The implementation of the flowchart shown in figure 4 for the recursive insertion sort is shown below, meeting success criteria 6. This is relevant because it is an efficient sorting algorithm as compared to linear methods.

```
def sort_low_high(items, n, metric):
    # Recursive insertion sort
    if n == 1:  # Base case
        return

    # Recursive call
    sort_low_high(items, n - 1, metric)

    last = items[n - 1]
    j = n - 2

    if metric == "price":  # If metric = price, the price attribute of items[j] is compared
        while j >= 0 and items[j].price > last.price:
            items[j + 1] = items[j]
        j -= 1
    elif metric == "published_date":  # If metric = published date, the published date attribute of items[j] is compared
        while j >= 0 and items[j].publish_date > last.publish_date:
            items[j + 1] = items[j]
            j -= 1
    else:
        raise Exception("Wrong metric input")  # For debugging: if wrong metric is passed into the function

    items[j + 1] = last

    return items  # Finally, return item list
```

The recursive algorithm does the following:

1. Base case: n = 1. Returns the array
2. Recursively sort the n-1 first items
3. Insert the last element in its correct position before returning the sorted array

To improve the reusability of the function, the lines below were implemented to dynamically alter which attributes were compared:

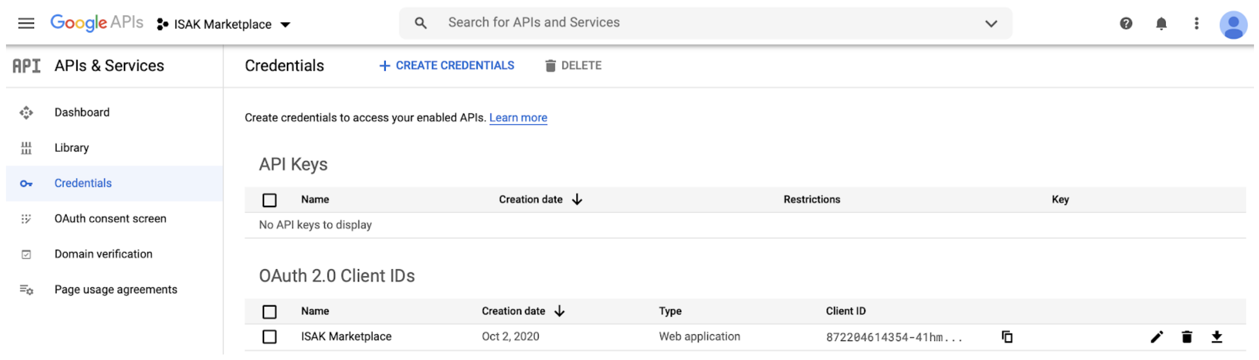
```
if metric == "price":
        # Compares  prices
elif metric == "published_date":
        # Compares dates published
else:
        # Returns error (for debugging purposes)
```

Moreover, this function was reused for the opposite operation (e.g. ascending and descending order) by simply reversing the output:

```
def sort_high_low(items, n, metric):  # Returns the reversed the sort_low_high output list
    return reversed(sort_low_high(items, n, metric))
```

Storing users and Google authentication

A top priority of the client is that users are able to log in to the webpage using a Google account. This can be done through the Flask Dance library [3] which can automatically communicate with the Google OAuth 2.0 Client, set up through the Google Developer Console (see figure 8), meeting success criteria 1.



Credentials

Create credentials to access your enabled APIs. Learn more

API Keys

Name	Creation date	Restrictions	Key
No API keys to display			

OAuth 2.0 Client IDs

Name	Creation date	Type	Client ID
ISAK Marketplace	Oct 2, 2020	Web application	872204614354-41hm...

Figure 8: Google Developer console, where the Google OAuth 2.0 can be set up

Storing user data is important when creating and retrieving posts and transactions. Users are only stored once in the database under the “User” table the first time they log in. After a successful authentication process with the Google OAuth client, the user is redirected to the “store_user” route, defined as such:

```
@app.route("/login/google/store_user")  # User redirected here after Google Auth.
def store_user():
  resp = google.get("/oauth2/v1/userinfo")  # Gets user info from Google server
  resp = resp.json()

  user = User.query.filter_by(g_id=resp["id"]).first()  # Check if user already exists in database

  if user:  # User already in database
     login_user(user)  # Built in method that registers that user as logged in
     return redirect(url_for("index"))  # Redirects to homepage
  else:  # User is new
     # Get Google user info
     g_id = resp["id"]
     g_email = resp["email"]
     g_name = resp["name"]
     g_picture = resp["picture"]

     user = User(g_id=g_id, g_email=g_email, g_name=g_name, g_picture=g_picture)  # Create new user object
     # Add and commit to DB
     db.session.add(user)
     db.session.commit()

     login_user(user)

     return redirect(url_for("index"))
```

There are two especially essential components for this process. The first is checking if the user already exists, done with the following statement:

```
user = User.query.filter_by(g_id=resp["id"]).first()  # Check if user already exists in database

if user:  # User already in database
   # *** Redirect user to homepage ***
else:  # User is new
   # *** Store user in database ***
```

The first line queries the database for a specific user (with the criteria “g_id”, which is the Google account ID). If the user is not in the database, the query returns an empty list, thus the “else” statement would run instead of the “if”.

The second essential component is storing the user in the database. This is done through the lines:

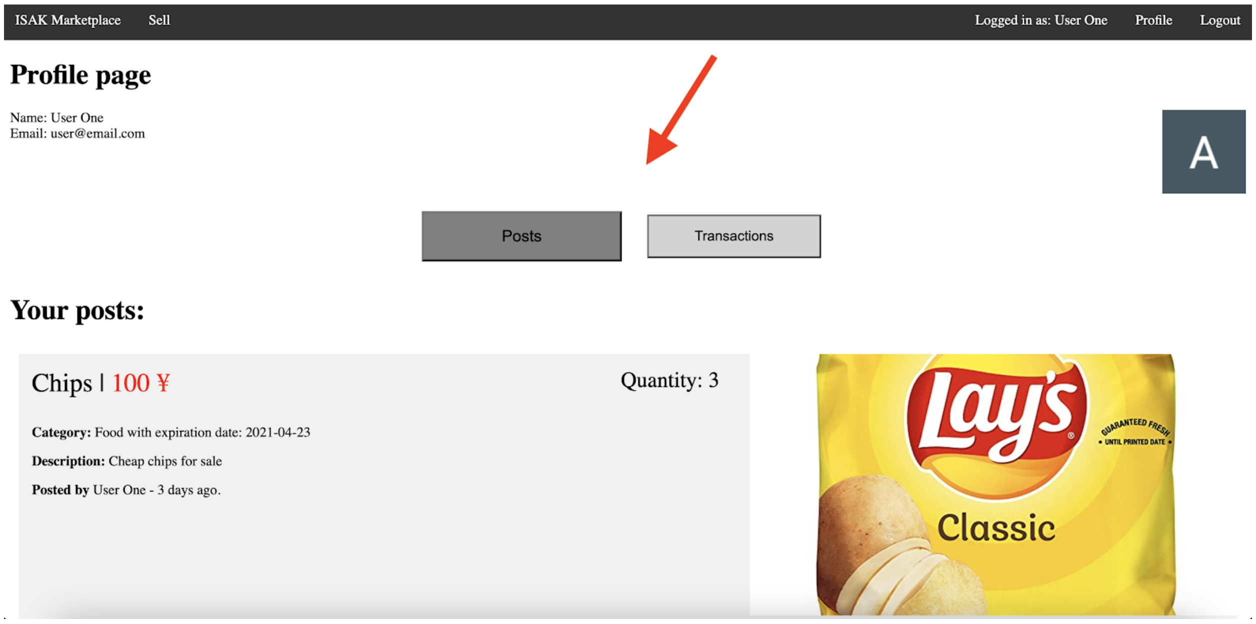
```
g_id = resp["id"]
g_email = resp["email"]
g_name = resp["name"]
g_picture = resp["picture"]

user = User(g_id=g_id, g_email=g_email, g_name=g_name, g_picture=g_picture)  # Create new user object
# Add and commit to DB
db.session.add(user)
db.session.commit()
```

Here, a “user” object is created from the class “User” with the attributes defined on the first 4 lines. This object is added and committed to the database.

Displaying profile and transactions

The profile page displays both user posts and transactions, meeting success criteria 4. To improve the usability of this page, these sections were created to be toggleable using buttons (indicated with a red arrow in figure 9).



ISAK Marketplace Sell Logged in as: User One Profile Logout

Profile page

Name: User One
Email: user@email.com

Posts Transactions

Your posts:

Chips | 100 ¥ Quantity: 3

Category: Food with expiration date: 2021-04-23

Description: Cheap chips for sale

Posted by User One - 3 days ago.

Figure 9: Screenshot of webpage, where the red arrow in the center highlights the buttons that toggle the “post” and “transaction” sections.

To accomplish this, client side Javascript can be used, while also reducing the server traffic, because the server does not receive requests every time the user changes views. The server harnesses the combination of Jinja (templating language) by passing all the user's item posts and transactions into the template, and Javascript to toggle the sections. This backend process is shown below:

```
@app.route("/profile")
@login_required
def profile():
  all_users = User.query.all()
  all_items = Item.query.all()
  current_user_items = Item.query.filter_by(user_id=current_user.id).all()

  transactions = Transaction.query.all()
  transactions = [transaction for transaction in transactions if
                transaction.buyer_id == current_user.id or transaction.seller_id == current_user.id]

  return render_template("profile.html", google=google, all_users=all_users,
current_user_items=current_user_items, all_items=all_items, transactions=transactions,
current_date=datetime.now())
```

Two especially relevant line of code are the following:

```
transactions = Transaction.query.all()
transactions = [transaction for transaction in transactions if
                transaction.buyer_id == current_user.id or transaction.seller_id == current_user.id]
```

The server queries all the transactions of the database, however on the second line these queries are filtered to only include the transactions the current user was a part of - either by being the seller or buyer. This is done using "list comprehensions" [5], effectively saving space and increasing readability of the code.

In the front-end HTML template the following code is used:

```
<div class="display-selection">
  <button id="enable-posts-btn" onclick="enablePosts()">Posts</button>
  <button id="enable-transactions-btn" onclick="enableTransactions()">Transactions</button>
</div>

<div class="post-container">
   ##### POST RENDERING ######
</div>

<div class="transaction-container">
  ##### TRANSACTION RENDERING ######
</div>
```

An important part of this code is the buttons at the top and the two main blocks of html code; one with class "post-container" and the other with class "transaction-container". When the buttons are pressed, the "display" CSS property of one of these is set to "none", effectively not rendering the element. A javascript example used in this client-side processing can be seen below:

```
function enablePosts() {
  // Runs when "posts" button is pressed

  let selection = document.querySelector(".post-container"); // Gets the post container element
  let selection_display = selection.style.display

  // Checks if the block is not already visible
  if (selection_display == "none") {
    selection.style.display = "block"; // Renders posts

    // Changes the appearance of the button pressed, indicating which page the user is on:
    let button1 = document.getElementById("enable-posts-btn");
    button1.style.transform = "scale(1.15)"
    button1.style.backgroundColor = "grey"

    // Hides the transaction container
    document.querySelector(".transaction-container").style.display = "none";
    let button2 = document.getElementById("enable-transactions-btn");
    button2.style.transform = "none"
    button2.style.backgroundColor = "lightgrey"
  }
}
```

This function is run when the “posts” button is pressed, showing the posts section and hiding the transaction section. Additionally, the button appearance changes to become bigger, more easily indicating which page the user is viewing. A similar but opposite function is used to toggle the transaction posts.

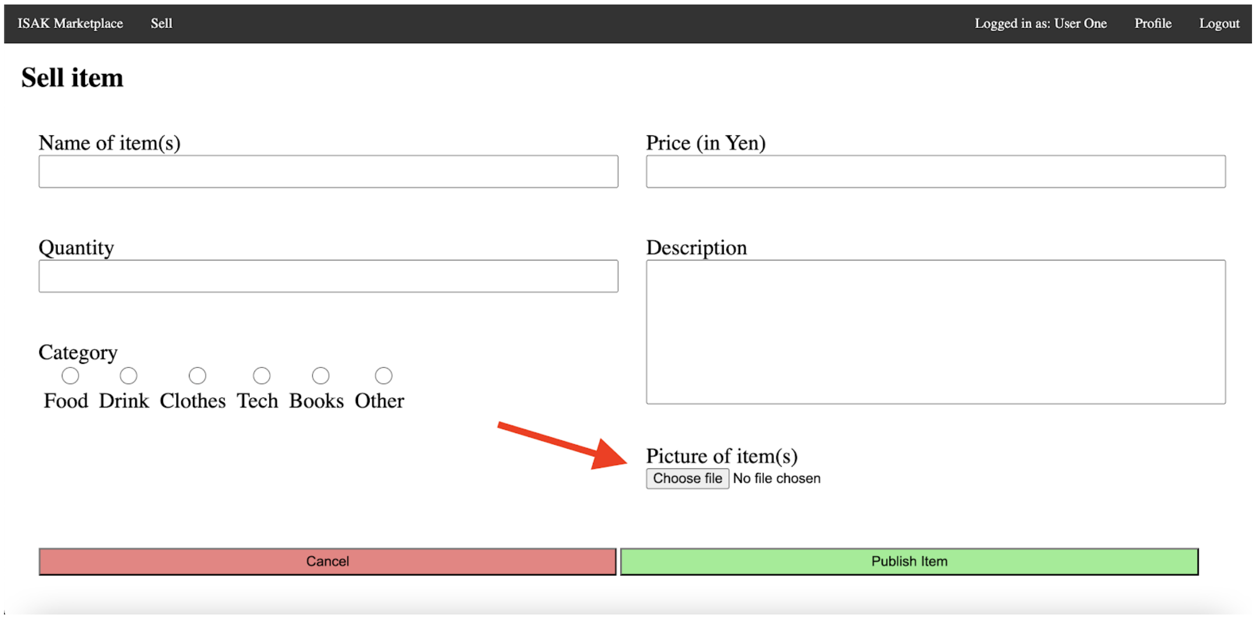
In rendering the posts/transactions, the Jinja templating language is frequently used to make this process more effective and dynamic. An example is shown below:

```
<div class="item-category">
  <strong>Category:</strong> {{ item.category }}
  {% if item.category in ["Food", "Drink"] %}
      with expiration date: {{ item.expiration_date.date() }}
  {% endif %}
</div>
```

Because the expiration date only applies to food and drinks, control flow statements can be used to determine if this attribute needs to be rendered. Using “{% if <condition> %}” can accomplish this.

Handling files

Pictures of items are an important part of the visual experience and usability for users, and therefore uploading files to the server is crucial. Meeting the success criteria 2f, the sell form includes a file upload system, as shown in figure 10.



Sell item

Name of item(s)	Price (in Yen)
Quantity	Description
Category ◯ Food ◯ Drink ◯ Clothes ◯ Tech ◯ Books ◯ Other	
	Picture of item(s) Choose file No file chosen

Figure 10: Screenshot of the sell form, where the user can input the information for posting an item. The red arrow highlights the file upload section.

To solve errors that occurred when no file was uploaded, a “try/except” expression was used, thus handling the errors, as shown below:

```
# Validating the sell form
if form.validate_on_submit():
  # Handling errors if the user does not submit a file
  try:
      # Attempts to retrieve file data
      img = form.pic_file.data
      filename = secure_filename(img.filename) # Function from werkzeug library to ensure user input
does not involve exploits
      filename = "u" + str(current_user.id) + "_" + filename # Create filename format
      img.save(os.path.join(os.getcwd(), "static", "item_pics", filename)) # Saves the file to the correct
directory
  except:
      # If no file submitted:
      filename = None # A "None" filename will result in the default item image being used
```

Especially important is the line below, which was added as a precaution for potential user exploits that could be used through the file upload system.

```
filename = secure_filename(img.filename)
```

The function “secure_filename()” is defined in the flowchart in figure 6, and the implementation of the algorithm can be seen as follows:

```
def secure_filename(filename):
  filename = list(filename) # Convert string to list
  illegal_characters = list("{}[]()%$#!/&=?+*^.") # Which characters to exclude
  last_period_ind = None # Storing the index

  # Looping through characters in filename
  for i in range(len(filename)):
    wrong_char = False # Starts each iteration assuming the char is valid
    for j in range(len(illegal_characters)): # Looping through the illegal char list
      if filename[i] == illegal_characters[j]: # Check if the filename is invalid
        wrong_char = True
        break # Stop checking illegal characters

    if wrong_char: # If the char is invalid...
      if filename[i] == ".": # Checks if the char is a "." to keep track of the last one (which is a part of the .png/jpg)
        last_period_ind = i
      filename[i] = "_" # Replace the old

  if last_period_ind: # if a period is found
    filename[last_period_ind] = "." # Replace the last found period index with a "."
    return "".join(filename) # Convert the list of characters back to string
  else:
    # Runs if no period found - then it must be invalid
    return "default_img.png"
```

Here, nested “for” loops are used to iteratively compare every character in the “filename” to characters in “illegal_characters”. After checking the validity of the character, it is either replaced or not. A crucial part of this program can be found in these lines:

```
if filename[i] == ".": # Checks if the char is a "." to keep track of the last one (which is a part of the “.png/jpg”)
    last_period_ind = i
```

```
if last_period_ind: # if a period is found
    filename[last_period_ind] = "." # Replace the last found period index with a "."
```

This serves the purpose of keeping track of the last “.” in the filename, because the last one is a part of the file extension (e.g. .png or .jpg). The latter code snippet replaces the character at the last “.” index.

Initially, when users did not upload a file, the formatting of the website became unreliable (effectively breaking the frontend layout). Instead of requiring the user to upload a picture, a default image (“default_img.jpg”) was added as a placeholder when no file was received, thus maintaining the aesthetics of the webpage.

A full list of Python libraries used can be found in the appendix section [M].

To run the program, Python 3.x and pip must be installed. Dependencies can be installed using “pip install -r requirements.txt” in the project directory, or by activating the virtual environment. Thereafter, the “app.py” file must be run by running “python app.py”.
Word count: 1037