FITFLIP: A COMMUNITY-DRIVEN, ZERO-FEE ALTERNATIVE TO ONLINE CLOTHING MARKETPLACES WITH OPTIMIZED USER EXPERIENCE

Kyle He [1], Garret Washburn [2]

[1] Mount Si High School, 8651 Meadowbrook Way SE, Snoqualmie, WA 98065
[2] California State Polytechnic University, Pomona, CA, 91768

ABSTRACT

The method proposed in this paper, the FitFlip mobile application, was devised as a solution to the problem that originates from online clothing websites like Grailed, whilst providing a useful marketplace to buy and sell, also heavily taxing the users both monetarily and in the user experience [2]. This issue became prominent as it would take up to 30% of the income users would get from selling clothes, despite only acting as a middleman. The FitFlip application intends to solve this problem as it is a community based, zero-fee platform where the transaction terms are stipulated strictly by the users and is not imposed upon by the FitFlip service in any way. This fixes the major issue prominent with services such as Grailed or Depop. Additionally, the FitFlip application seeks to resolve a few other issues prevalent within these popular services, such as inefficient search and filtering tools and the inability to create custom transactions where users can trade items instead of only selling. Within this paper, multiple experiments are performed that ensure FitFlip mobile application has a reliable and consistent user experience [1]. These experiments consisted of an experiment to ensure the response time was efficient, as well as another experiment to determine the consistency of response time given the quantity of data to be loaded. Ultimately, the FitFlip application is more than sufficient solution to the proposed problem, as it provides users with an efficient platform to connect and trade clothing items, which boosts a sense of community and encourages the reuse of clothes.

KEYWORDS

FitFlipp, Trading, Community-Driven, Online Clothing Marketplaces

1. INTRODUCTION

As long as online fashion marketplaces have existed on the internet, the practice of the marketplace itself imposing a fee for moderating a transaction has been commonplace. This behaviour, while not always abused, is often so and most often leads to the marketplace increasing the fees increasing to a disagreeable amount. Additionally, while perhaps not purposefully doing so, this practice also disables the users from creating their own unique transaction details. For instance, users are not able to perform trades if they are interested in trading one clothing item for another. Ultimately, this results in an inflated marketplace where items end up becoming more expensive than their listing price anddoes not qualify as a free

David C. Wyld et al. (Eds): CSIT, AMLA, IPDCA, NLPA, AIS, IPPR, SPTM – 2025
pp. 51-60, 2025. CS & IT - CSCP 2025 DOI: 10.5121/csit.2025.151404

marketplace where users can trade what they want with whoever they want. Effectively, these practices create a poor trading environment for both the marketplaces buyers and sellers.

While researching the topic, and given our history with the problem, we were able to identify three current methodologies available for public use. These methodologies are the online Grailed store, Depop, and OfferUp. These methodologies, while have proven to be popular with their big user bases, all share similar issues that we have aimed to resolve with the FitFlip application. Grailedand Depop, both online clothing marketplaces, show consistent behaviour in having large transaction fees and high shipping costs [3]. Additionally, OfferUp, while sharing this problem, does not have the advanced searching system required for an efficient clothing marketplace. Finally, all three methodologies for our problem do not include a 'custom transaction' feature, where users can set the terms for their transactions or create custom trades to swap clothing items. FitFlip aims to solve all of these issues by allowing users to create custom trades however they want, having advanced clothes searching and filtering features, and being entirely free for the user base. FitFlip puts the power back into the Fashionista's hand.

The FitFlip mobile application is an online clothes trading marketplace where users can post items they are interested in selling or trading, and other users can browse these items and contact the seller via a chat in order to negotiate a trade.

Within this paper, we performed two experiments that examined important factors such as the consistency in our response time and how the quantity of data directly affects the response time from the back end. Our first experiment, dealing with the average response time of a basic request, proved a big success. The experiment was conducted by timing how long it took for a request to be sent, responded to, and displayed in the front-end for the user to see. The average response time was found to be ~1.85 seconds and was quite a convenient speed for the average user. However, our second experiment did display an issue that called for some resolution. This experiment was also conducted by timing how long it took for a user to see a response, however, before each trial 5 new posts were added to what the user's app would load in. This, ultimately, demonstrated a positive linear relationship between the number of posts and how long it takes to load. At that time, there was no limit to how many items the app could load, and we were incentivized to implement a limit. We have found a solution to solve this ongoing issue.

2. CHALLENGES

In order to build the project, a few challenges have been identified as follows.

2.1. Flutter's Learning Curve and Instability

The first major component in which we encountered some challenges was the mobile application front end that employs the use of the Flutter framework alongside Dart. Particularly, what we found challenged was the ever-evolving nature of the Flutter framework, and constantly having to upgrade and/or update the Flutter SDK itself or packages that the project relies on [4]. Additionally, learning how to use the Flutter framework was a challenge itself, as it has its work set rules and objects to creating custom pages and widgets [5].

2.2. Managing Complexity in Flask Server Routing

Another major component that we encountered a few challenges with was the Flask powered back-end server. During the creation of the back-end server, we found ourselves managing a lot of different server routes, each with their own purpose and functionalities. It became challenging

to keep them organized and remember how they all worked and where they were located in the server. Additionally, while developing the mobile app, we also found it challenging to remember what server route what did when it came time to call them from the mobile app. However, we were able to sort through the server routes and create an optimized server capable of supporting our app.

2.3. Learning and Integrating Firebase with Flask

The last major component where challenges were faced has to be the Firebase database [6]. Due to Firebase's cloud nature, utilizing it was not as straightforward as a regular database. Firebase was new to us, and discovering how to use it required some learning. Additionally, we also had to learn how to utilize the Firebase admin SDK in Python, as we were required to do so in order to pull data from the database inside the Flask back-end server and return that to the mobile application front-end users. However, ultimately, employing the use of Firebase was a great option, as the speeds of the server and database are great.

3. SOLUTION

The FitFlip project has three major components. These major components include the mobile application front-end, the back-end server hosted on Render.com, and lastly the Firebase database for storage of data. During the development of the mobile application, we utilized the Flutter framework for source management as it allowed us to easily manage one source that is compatible with both iOS and Android platforms [7]. Additionally, we did employ the use of the devices geolocation, therefore, we did have to create custom logic to handle each major distribution's geolocation requirements. For the creation and management of the back-end server, the Flask framework was employed [8]. The back-end server contains a variety of routes, and essentially handles all major tasks of the FitFlip application such as logging in or registering, creating a post, and getting the user's feed. Lastly, Firebase was used as the collection point of all data relevant to the service. This includes all user data for posts, messages, location data, created posts, direct messages, and all transactions. All of these components are necessary for trading clothes locally, and are deleted upon request of the user or any party involved. The FitFlip service is designed to keep the server as a middle man between the user and the database. This is done purposefully so that the back-end server may directly view and limit all data that is going in and out of the database and to the user.

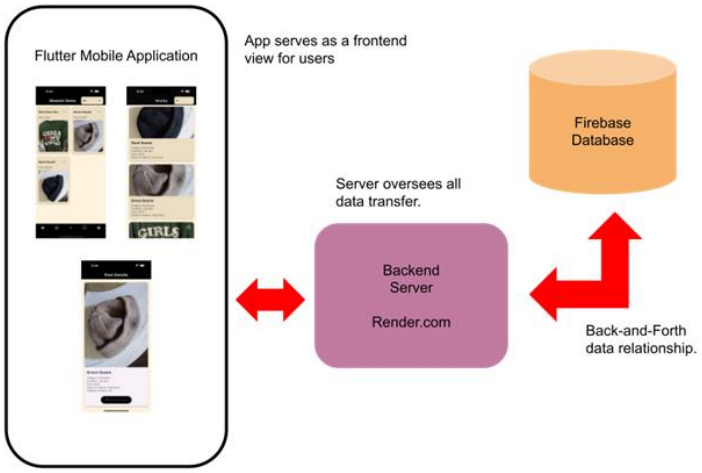


Figure 1. Overview of the solution

The FitFlip app is designed to be a simple and easy to navigate trading app, that allows you to connect and exchange goods that are more tailored to the fashion industry, as people can trade their sneakers for hoodies or watches for necklaces. The app is equipped with code that allows it to pull data about local listings and potential trade offers. The app was created utilizing the Flutter framework.

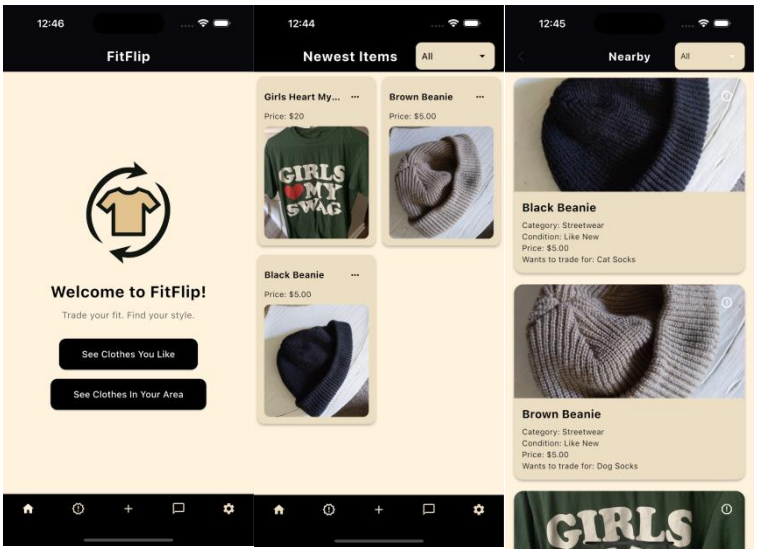


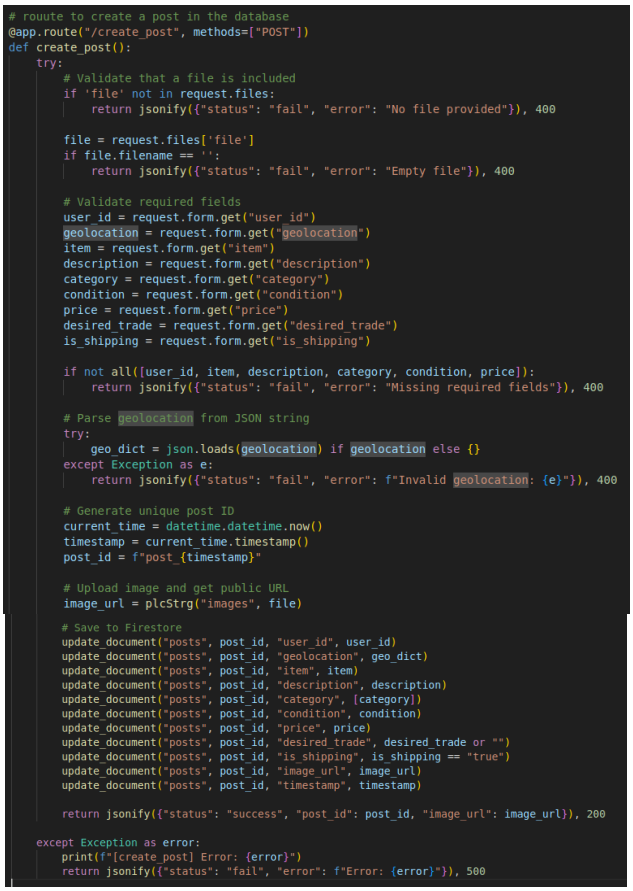
Figure 2. Screenshot of the APP

```
Future<void> _getClothingFeed() async {
  try {
    // Get user geolocation
    final position = await Geolocator.getCurrentPosition(
      desiredAccuracy: LocationAccuracy.high,
    );

    var response = await customHttpClient.post(
      Uri.parse("$backEndUrl/get_posts_by_geolocation"),
      headers: {"Content-Type": "application/json"},
      body: json.encode({
        "user_id": userId,
        "geolocation": {
          "lat": position.latitude,
          "lon": position.longitude
        },
        "max_distance": maxDistance ?? 25
      }),
    );

    if (response.statusCode == 200) {
      final data = jsonDecode(response.body);
      setState(() {
        posts = data["posts"];
        filteredPosts = posts;
        _isLoading = false;
      });
    } else {
      print("Unable to get posts: ${response.reasonPhrase}");
      print(response.body);
      setState(() => _isLoading = false);
    }
  } catch (e) {
    print("Error fetching posts: $e");
    setState(() => _isLoading = false);
  }
}
```

Figure 3. Screenshot of code 1

Depicted in the screenshot above is the _getClothingFeed method responsible for getting the clothing feed from the backend given the user's geolocation and the max distance in which they would like to see clothing items from. We start off by gathering the client's location data and creating a HTTP Post request to the back-end server containing the client's user_id, geolocation, and max_distance [9]. This allows the back-end server to process what clothing items could be near that distance, upon which a list of posts will be returned to the user in the server's response. Once the response is gathered, the status code is checked to ensure the request was good and the app now has the post data to display. From there, the posts are parsed from the response and the posts are then loaded up and displayed for the user. If the request did not go through correctly, then errors are logged and the loading stops so the client will see a blank page.

After the mobile application, another crucial component in the FitFlip project is the back-end server. The back-end server acts as a connector for the user to communicate with other users as well as interact with and change data in the back-end server. The back-end server is what creates the outline for how the service works, and was written entirely in Python with Flask.



```python
# rouute to create a post in the database
@app.route("/create_post", methods=["POST"])
def create_post():
    try:
        # Validate that a file is included
        if 'file' not in request.files:
            return jsonify({"status": "fail", "error": "No file provided"}), 400

        file = request.files['file']
        if file.filename == '':
            return jsonify({"status": "fail", "error": "Empty file"}), 400

        # Validate required fields
        user_id = request.form.get("user_id")
        geolocation = request.form.get("geolocation")
        item = request.form.get("item")
        description = request.form.get("description")
        category = request.form.get("category")
        condition = request.form.get("condition")
        price = request.form.get("price")
        desired_trade = request.form.get("desired_trade")
        is_shipping = request.form.get("is_shipping")

        if not all([user_id, item, description, category, condition, price]):
            return jsonify({"status": "fail", "error": "Missing required fields"}), 400

        # Parse geolocation from JSON string
        try:
            geo_dict = json.loads(geolocation) if geolocation else {}
        except Exception as e:
            return jsonify({"status": "fail", "error": f"Invalid geolocation: {e}"}), 400

        # Generate unique post ID
        current_time = datetime.datetime.now()
        timestamp = current_time.timestamp()
        post_id = f"post_{timestamp}"

        # Upload image and get public URL
        image_url = plcStrg("images", file)

        # Save to Firestore
        update_document("posts", post_id, "user_id", user_id)
        update_document("posts", post_id, "geolocation", geo_dict)
        update_document("posts", post_id, "item", item)
        update_document("posts", post_id, "description", description)
        update_document("posts", post_id, "category", [category])
        update_document("posts", post_id, "condition", condition)
        update_document("posts", post_id, "price", price)
        update_document("posts", post_id, "desired_trade", desired_trade or "")
        update_document("posts", post_id, "is_shipping", is_shipping == "true")
        update_document("posts", post_id, "image_url", image_url)
        update_document("posts", post_id, "timestamp", timestamp)

        return jsonify({"status": "success", "post_id": post_id, "image_url": image_url}), 200

    except Exception as error:
        print(f"[create_post] Error: {error}")
        return jsonify({"status": "fail", "error": f"Error: {error}"}), 500
```

Figure 4. Screenshot of code 2

Depicted in the screenshots above is the /create_post route of the Flask server. This server route receives the crucial information supplied by the user to create a new post for other users to see. The route starts by ensuring the user sent over a valid file for the image, and then validates all critical information that the user should have supplied to create a successful post. This information includes the user's id number, their geolocation, and qualities of the clothing such as the description, category, and condition. If any of these values are not included, then the server will return an error response. Once all of the values have successfully been received, the

geolocation is parsed from JSON to a dictionary and the current timestamp gathered. With these two last items, all of the data is then saved into the Firebase database utilizing the update_document helper function. After the post is successfully saved, the server returns with a successful response.

The Firebase database is another crucial component in the FitFlip project, as it is responsible for all of the data storage and management for service. All of the user's posts, location data, interactivity analytics, preferences, and direct messages are stored in the Firebase database.

```python
def get_newest_posts_from_firestore(limit=20):
    try:
        posts_ref = db.collection("posts")
        query = posts_ref.order_by("timestamp", direction=firestore.Query.DESCENDING).limit(limit)
        docs = query.stream()

        newest_posts = []
        for doc in docs:
            post_data = doc.to_dict()
            post_data["post_id"] = doc.id
            newest_posts.append(post_data)

        return newest_posts
    except Exception as e:
        print(f"[firebase.py:get_newest_posts_from_firestore] Error: {e}")
        return []

def create_or_get_direct_message(from_user, to_user):
    try:
        messages_ref = db.collection("direct_messages")
        docs = messages_ref.stream()

        # Check for existing conversation
        for doc in docs:
            data = doc.to_dict()
            if from_user in data and to_user in data:
                return {"status": "exists", "dm_id": doc.id}

        # Create a new DM if none exists
        new_dm_ref = messages_ref.document()
        new_dm_ref.set({
            from_user: {},
            to_user: {}
        })

        return {"status": "created", "dm_id": new_dm_ref.id}

    except Exception as error:
        print(f"[firebase.py:create_or_get_direct_message] Error: {error}")
        return {"status": "fail", "error": str(error)}
```

Figure 5. Screenshot of code 3

In the image above, the get_newest_posts_from_firestore and create_or_get_direct_message functions can be seen. These functions each are responsible for getting all of the newest posts from the Firestore database and creating or getting a direct message instance between users, respectively [10]. The former, starts by referencing the posts document and querying it to find the 20 most recent posts created. Once the post docs have been gathered, the posts are organized into a list and then returned. The next function starts by creating a reference to the direct_messages collection in Firestore, and streaming all of the contents back to the docs variable. The function then iterates through all conversations to see if a direct messaging instance already exists between the two user ids passed as parameters. If the messaging instance exists, then the doc id will be returned and if not, then a new document will be created to keep track of the conversation.

4. EXPERIMENT

4.1. Experiment 1

A major property of the FitFlip mobile app is that it loads quickly and consistently when a user has the urge to browse. It is important that clothing items up for barter load in an appropriate amount of time, as the user could lose focus or become uninterested in using the application.

To ensure the speedy loading time of the FitFlip app, an experiment will be performed to find the average response time when a user visits the Latest page. This experiment will involve two testers, one responsible for loading the clothing items in the Latest page and the second for recording loading times with a stopwatch. The first experimenter will notify the second when they have started loading their items, upon which, the second experimenter will begin stopwatch. Once the items have been loaded, the first experimenter will inform the second to stop the timer. The time will then be recorded in a spreadsheet. This experiment will consist of 8 trial runs, to get a good idea of the average loading time.

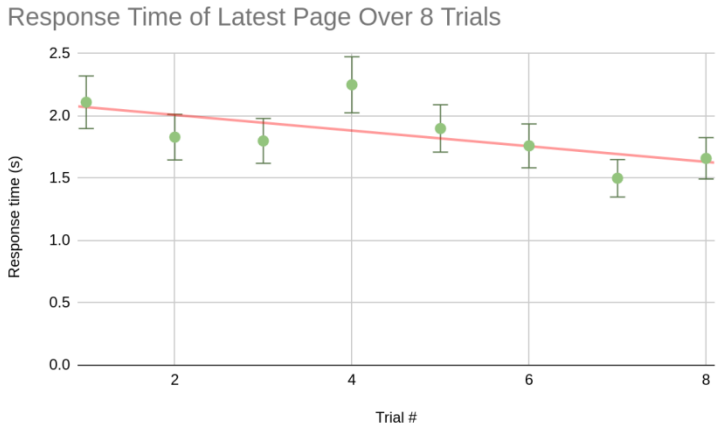


Figure 6. Figure of experiment 1

After performing the experiment, it is clear that the response time for the Latest page is quick and consistent. The average response time from the back-end server was around 1.85 seconds, and loaded consistently with few outliers. Specifically, there is only one noticeable outlier in the chart above, however, the difference between the outlier and the average response time is that of a measly .4 seconds. While the consistency did not catch us off guard during this experiment, the quick responses did. Originally in our testing, the loading times did feel longer than what was recorded in the experiment, however, this could be due to other various factors such as momentary internet speeds or first time running the app. Additionally, previous tests were conducted under app development, so it could be said that with the production server that the response time is understandably quicker.

4.2. Experiment 2

Another feature of the app that is important that it works well is the “Clothes Nearby” page. This page loads for the user clothes that are in a given area whose distance the user is able to set. This page does not have a limit for how many clothing items can be listed on the page, and will attempt to load every clothing item in the user’s area. This could potentially cause loading issues, which would negatively impact the user’s experience.

This experiment will also consist of two team members. The first experimenter will, as previously, be responsible for the loading of the “Clothes Nearby” page and the second will control the stop watch. This experiment will differ, however, as in between each trial, both experimenters will come together to add only 5 new clothing items. As the trial number increases, the number of clothing items will also increase. This will clearly define if there are any loading issues (extended loading times or timeouts) as the time of loading will be affected if there are.

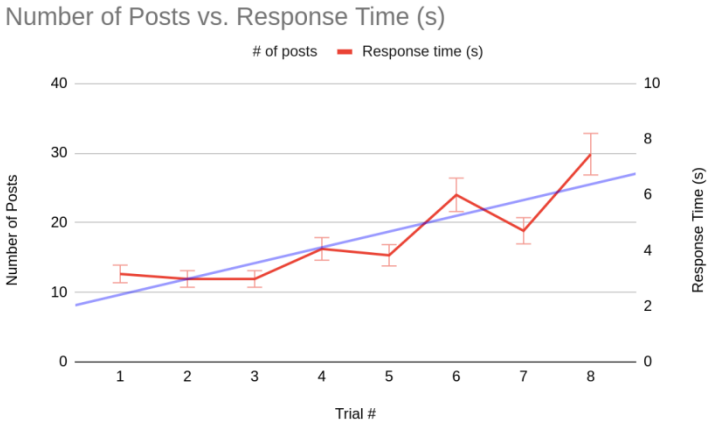


Figure 7. Artistic Autists Adventure Game Difficulty

After performing the experiment and recording the results in the chart above, it is clear to see that there is a positively linear relationship between the number of posts being loaded and the response time. It is quite obvious as to why this is happening, the reason being that simply there is just more data being transferred. During the experiment, we found very surprising the response time average that we encountered. We found this surprising as we were expecting a bit of a slower experience, considering we were using the free version of Firebase. As seen in the graph, however, there are a few dips and inconsistencies in the data, such as how in Trial #6 the response time is much greater than the response time from Trial #7 despite having more posts to load. This can be due to a variety of factors, and is impossible to nail down to just one. A few ideas could be that the backend was slower to respond in this instance, there was momentary lag, or simply could have been an experimentation error. Regardless, despite these dips, the trendline shows a consistent positive linear relationship and displays the behaviour of the current system.

5. RELATED WORK

Grailed is a second-hand online shopping option for internet shoppers looking to find specific clothes [11]. Additionally, Grailed has an option for users to sell their own clothes, enabling them to make a profit off of their old fashion. However, Grailed has a steep 20% fee for sellers who are looking to sell their old fashion, and also require that buyers pay for shipping if necessary for them to acquire the items. This puts Grailed in a position of pure profit, while the buyer and seller lose money. FitFlip, however, has no finders fee and allows users to negotiate their own prices and mediums of shipping, putting the power into the hands of the buyers and sellers. This allows for a more free marketplace, and a better user experience and transaction environment.

Depop, very similarly to Grailed, is an online service where users can buy and sell their fashion [12]. However, also very similar to Grailed, takes a fee during transactions. The biggest issue that we have found, which we have also found in Grailed but did not explicitly mention, is the inability to create custom trade rules so users can trade their own specific items as opposed to flatout paying. FitFlip provides an entirely free marketplace for any kind of transaction that the users are willing to participate in. As opposed to a strict trading system, FitFlip focuses purely on connecting users and enables them to trade on their terms.

OfferUp is a very popular mobile application that enables users to buy and sell from other users to make local or distant transactions [13]. OfferUp allows users to buy and sell not only clothes,

but also any items that the user is interested in selling or buying. This does not sound like an issue up front, however, it becomes apparent users are likely to struggle to find specific clothing items that they are interested in. Additionally, the moderation and filtering available on OfferUp for clothing items is not sufficient for finding exactly what the user is looking for. FitFlip solves both of these issues with an advanced filtering system and is tailored to show the user exactly what clothing items they are looking for.

6. CONCLUSIONS

A very prominent issue that became apparent as we started testing the mobile application was the lack of content moderation within the app. Users are able to submit any post idea that they have, clothing item or not, and currently there is no live way for us to monitor what is visible in the app. A solution is to implement some content moderation system that determines whether the post is a valid clothing item or not, whether that be AI or human powered [14]. Additionally, it would be great if there was a reputation system within the app that allowed users to rate and view the rating of other users to see if they are reputable. This would enable users to be confident about going into a high-profile trade without the fear of being scammed. Lastly, another issue that became apparent during testing was the response time of loading above 20 post items. To ensure a more consistent and quick user experience, the back-end server should limit the number of post items to be loaded to be a maximum of 20.

Although this project has certain flaws and issues that are currently left unattended, I am happy to see my concept into action as it solves the issues I constantly have to deal with when selling clothes online [15]. One of the biggest issues I would say is the safety and security of the transactions as without the middleman users must trust and rely on both sides to hold their promises. I hope to find a solution to that sometime soon!

REFERENCES

[1] Islam, Rashedul, Rofiqul Islam, and Tohidul Mazumder. "Mobile application and its global impact." International Journal of Engineering & Technology 10.6 (2010): 72-78.

[2] Hassenzahl, Marc, and Noam Tractinsky. "User experience-a research agenda." Behaviour& information technology 25.2 (2006): 91-97.

[3] Veselova, Andzela. "Marketplace vs online shop." Economic and Social Development: Book of Proceedings (2020): 30-36.

[4] Tashildar, Aakanksha, et al. "Application development using flutter." International Research Journal of Modernization in Engineering Technology and Science 2.8 (2020): 1262-1266.

[5] Faust, Sebastian. Using Google´ s Flutter framework for the development of a large-scale reference application. Diss. Hochschulbibliothek der Technischen Hochschule Köln, 2020.

[6] Moroney, Laurence. "The firebase realtime database." The Definitive Guide to Firebase: Build Android Apps on Google's Mobile Platform. Berkeley, CA: Apress, 2017. 51-71.

[7] Lazareska, Lazarela, and Kire Jakimoski. "Analysis of the advantages and disadvantages of Android and iOS systems and converting applications from Android to iOS platform and vice versa." American Journal of Software Engineering and Applications 6.5 (2017): 116-120.

[8] Mufid, Mohammad Robihul, et al. "Design an mvc model using python for flask framework development." 2019 International Electronics Symposium (IES). IEEE, 2019.

[9] Sakurada, Reiko, Takaaki Moriya, and Junichi Akahani. "Extracting user posting behavior using HTTP flow." 8th Asia-Pacific Symposium on Information and Telecommunication Technologies. IEEE, 2010.

[10] Kesavan, Ram, et al. "Firestore: The nosql serverless database for the application developer." 2023 IEEE 39th International Conference on Data Engineering (ICDE). IEEE, 2023.

[11] Padmavathy, Chandrasekaran, Murali Swapana, and Justin Paul. "Online second-hand shopping motivation–Conceptualization, scale development, and validation." Journal of Retailing and Consumer Services 51 (2019): 19-32.

[12] Ferraro, Carla, Sean Sands, and Jan Brace-Govan. "The role of fashionability in second-hand shopping motivations." Journal of retailing and consumer services 32 (2016): 262-268.
[13] Abbes, Intissar, Yousra Hallem, and Nadia Taga. "Second-hand shopping and brand loyalty: The role of online collaborative redistribution platforms." Journal of Retailing and consumer Services 52 (2020): 101885.
[14] Jiang, Na, et al. "Beyond AI-powered context-aware services: the role of human–AI collaboration." Industrial Management & Data Systems 123.11 (2023): 2771-2802.
[15] Chen, Kuan-Ting, and Jiebo Luo. "When fashion meets big data: Discriminative mining of best selling clothing features." Proceedings of the 26th international conference on world wide web companion. 2017.