FlushHub

Technical Specifications Document

Beck Sonstein, Aran Pattarachanyakul, and Zachary Gou

An Introduction

The goal of this project is to help those on Boston University (BU) campus find bathrooms. We took inspiration from the TerrierTransit application that helps students, faculties, and visitors navigate the public transportation system around Boston, but primarily on BU campus.

This project was made in collaboration between Beck Sonstein, Aran Pattarachanyakul, and Zachary Gou. Sonstein worked on all of the front-end components of the application. This includes all the custom icons, xml files, and fragment navigation, among other things. Pattarachanyakul worked on the map functionalities of the application. This includes adding animation to the map, making map components clickable and passing in proper information, launching Google Maps intent for routing, filtering bathrooms based on user's specifications, and translating the application into Thai. Gou worked on scraping the data from BU's website, making the database and making sure it can properly communicate with our application, fixing asynchronous threads issues, and making all the viewmodels. However, some contributions often overlap as we help each other debug and bounce ideas. Tasks are assigned verbally during in-person meetings as well as in group chat on Instagram. However, those task divisions have been translated onto a Jira board for better visualization. All instructors' BU emails, based on the rubric, have been added to the board. Please refer to this link:

https://bazflushhub.atlassian.net/jira/software/projects/KAN/list?atlOrigin=eyJpIjoiZDgyZDU5ZDU4NGE0NDEzNjkwNDYyNDQ4NWM2MjcxY2UiLCJwIjoiaiJ9.

The application allows users to to locate bathrooms on maps visuals as well as on recyclerview visuals. The user can route themselves to a selected bathroom through Google Maps as well as view information on that bathroom through our UI. The information includes other users' reviews, location of the bathroom, and the stars of the bathroom. Users are also able to leave reviews on a specific bathroom. In the case where users find themselves bored while using the bathroom, they can also choose to entertain themselves with any of our three entertainment options: Boggle game, cute animal pictures generator, and news articles.

FlushHub uses a total of six API's and one database: TomTom API, News API, The Dictionary API, Cataas API, Dog CEO API, Random Fox API, and our own MongoDB database. TomTom API is used to display the map on the application, add markers to the map, and calculating distance between the user and bathrooms with pathways in consideration. News API is used for our entertainment page to get news articles information. Dictionary API is used to check if a word exists in the English language so that users won't have to download the entire dictionary along with the application to play Boggle under the entertainments tab. Cataas, Dog CEO, and Random Fox APIs return a cute animal photo when called for us to display to the user as another form of entertainment. The MongoDB is used to host the information of each bathroom. Please refer to **figure 1** below.

Application Architecture

In terms of Architecture choice, our main data pipeline will follow the standard MVC model. In Overall UI architecture, all fragments will center or pivot around “nav_home” and each feature or entry point will “pivot off” of “nav_home” for a certain amount of fragments to which will then have routes that return back “nav_home” creating a close loop for user experience. For the backend, the team chose to go with Realm and MongoDB sync because the team believed that such ease (architecturally, not implementation-wise) will allow more focus development of the application rather than backend architectures.

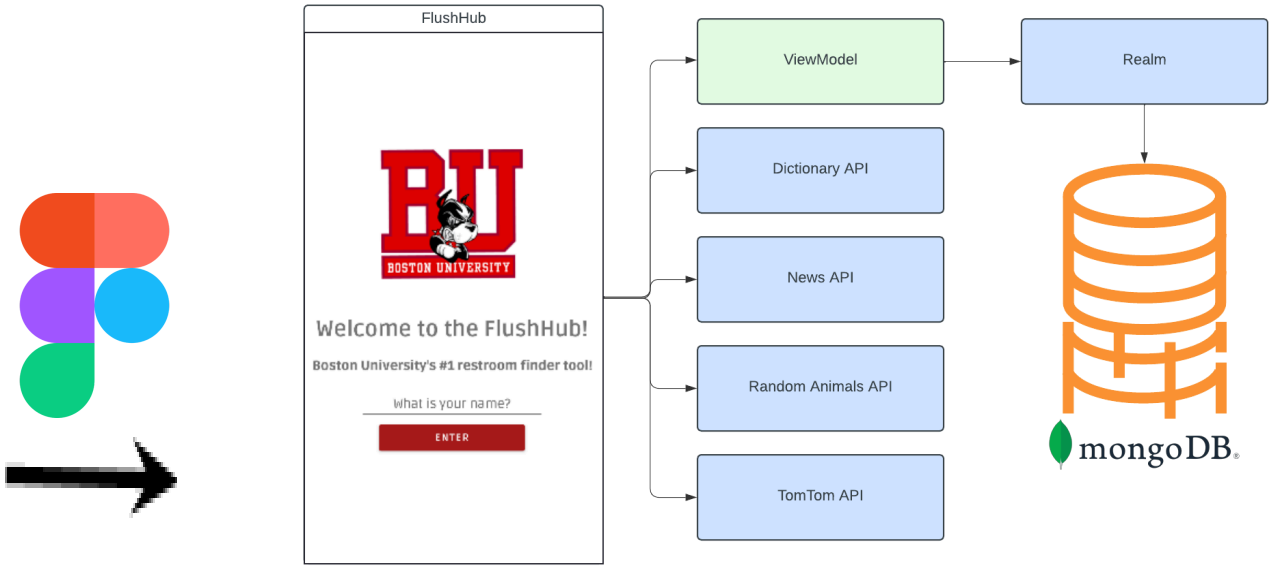
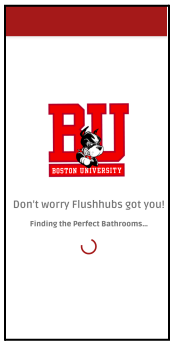


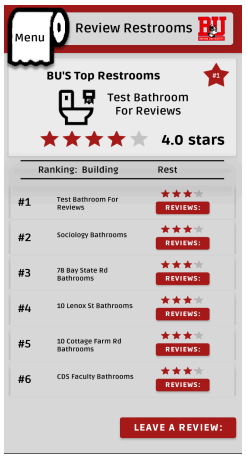
Figure 1: BU FlushHub Architecture Overview

Fragment/ Layout Architecture

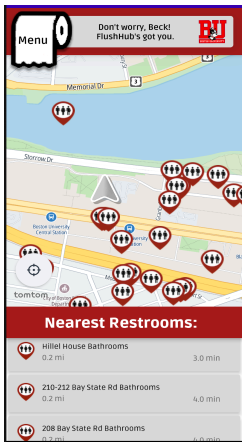
These are all the fragments that were created for our application. The fragments in the higher rows navigate to the fragments in the lower rows

LoadingFragment

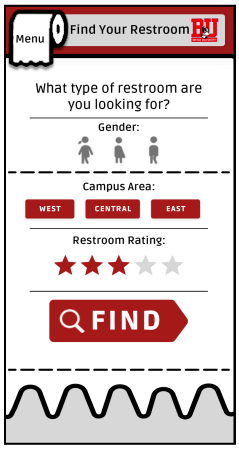




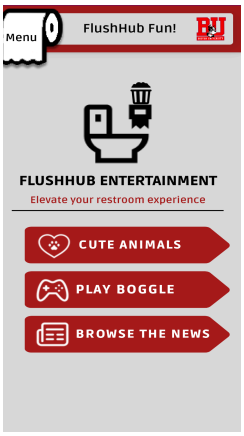
RatingsFragment:
The fragment for all Review related info



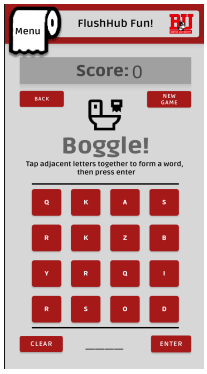
HomeFragment:
This is the home fragment That contains the main map



FindFragment:
Fragment for querying

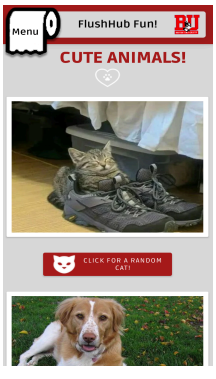


EntertainmentHomeFragment:
Homepage for all entertainment fragments

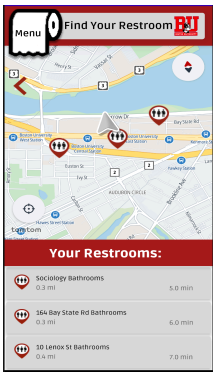


FlushHubBoggleFragment:
Boggle game fragment

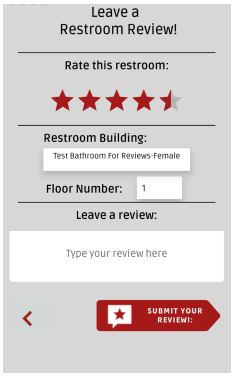
NewsFragment:
Browse news



RandomAnimalsFragment:
View cute animals



QueryResFragment:
See query results

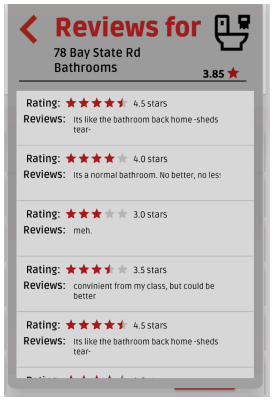


SubmitReviewFragment:
Leave a review fragment

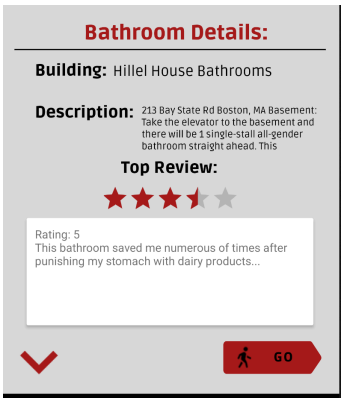
For some pieces of the app that we initially wanted to make fragments, we decided to make small layouts to include in our fragment layouts for organization and practicality of accessing the layout attributes in a fragment

We decided to make a custom navigation drawer to fit our vision for the app. We wanted to have a nav. drawer that rolled down from the top of the page, to resemble a sheet of toilet paper! This is a custom layout since Android Studio only supports navigation drawers that open horizontally

These are the other pages that we made into individual layouts referenced in the fragments:



ReviewListLayout



ExpandedBathroomDetailLayout

ShowGotoRouteBannerLayout

As seen in **Figure 1**, the team chose the following API's to interact with the application. Dictionary API, News API, and Random Animals API are all used for the entertainment section. However, TomTom API (for maps) and Realm are persisted and used across all parts of the application. Architecturally, the user enters a series of initial loading screens and possibly a landing page if the user is a new user that has not given a name nor has set permissions for tracking user current location. Then "nav_home" is the center where the user can pivot to different parts of the application. The application has three feature lifecycles: Query, Reviews, and Entertainment.

From the Query feature, the user navigates to a "nav_gallery" which hosts the query options such as gender, campus location, minimum number of stars, etc. Once the user has filled out their desired query, they are moved unto the "query_loading_page" where the page will submit a request through "BathroomViewModel" functions which in turns contacts Realm to pull the data off of MongoDB given the user constraints. Depending on the user queries, these processes may take a substantial amount of time due to the limitations of the TomTom API we use to calculate distance and ETA. TomTom limits developers' QPS (Queries Per Second), thus we can not instantly get the query as opposed to the rapid speed of fetching data from MongoDB. Once the query is satisfied, the "query_loading_page" ceases and then moves onto the "queryResultsFragment" which is a lightweight version of "nav_gallery" showing the results. From there, the user can select a bathroom to route to or choose to navigate elsewhere.

From the Ratings feature, the user simply can navigate to the “fragment_ratings” to view all ratings and also conveniently see the top bathroom out of all bathrooms from the database. Naturally, if the user wants to write a review, they can navigate to “fragment_submit_reviews” to write a review. Upon successful completion (when the review passes all valid checks), the user is automatically navigated back to the reviews. However, due to Realm and API constraints, the reviews page does not refresh itself automatically when new data is populated. Thus, the user must manually refresh the page by swiping down the RecyclerView to which they can now view their review they had just created.

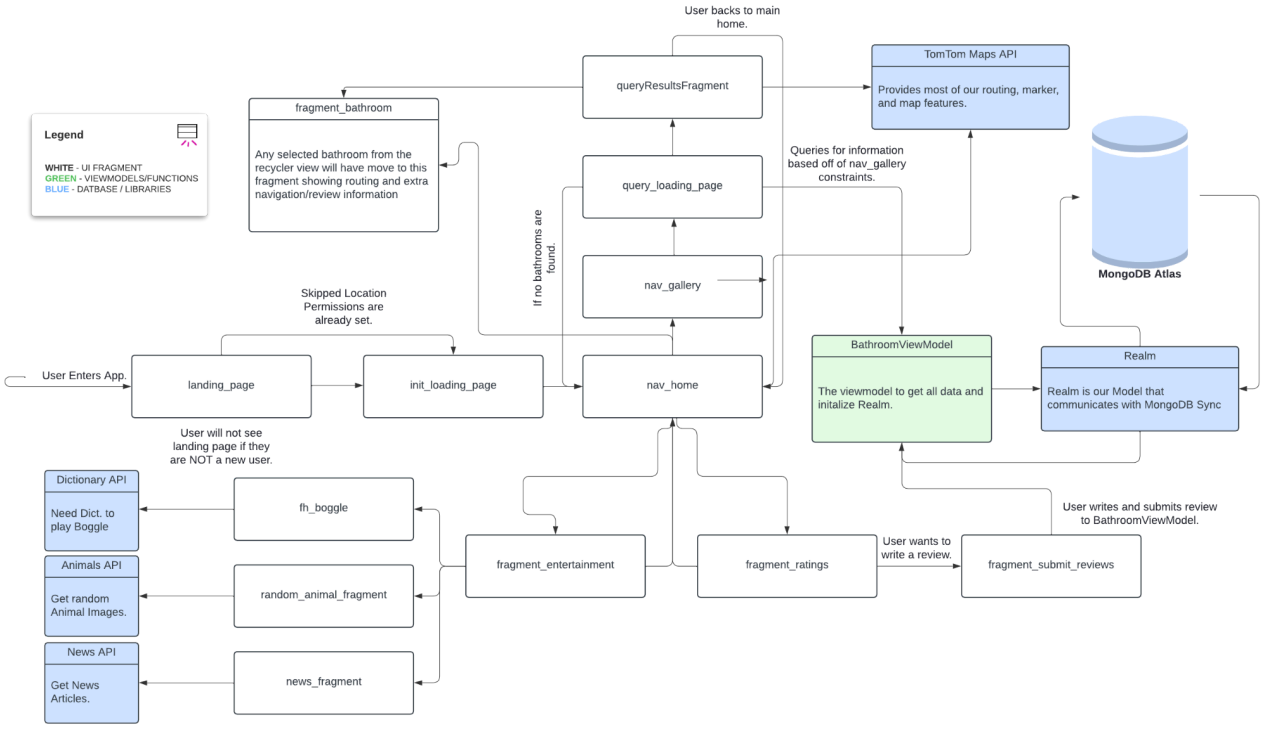


Figure 2: [BU FlushHub Mobile Architecture]

Road Blocks & Solutions

Below we list some major roadblocks in development. Of course, this is not an exhaustive list. The following roadblocks were identified:

1.) Hitting TomTom API and QPS Limits yielded unwanted outputs and crashed the application often. Bathroom distance would return as N/A when API call's limit is hit. In fact, if instructors find all the distances to be N/A during their own testing, it could be because the API key has hit its limit.
2.) Loading times took too long due to TomTom API limits when calculating distance and time from the current user location to the desired bathroom.
3.) The initialization logic and the order of async calls caused crashes and invalid/empty data. This included Permissions Logic.
4.) TomTom Map crashing because of VAO and OpenGL errors due to making the map global through a companion object.
5.) Navigation errors that included not referencing the right Navcontroller or ID's leading to crashes.
6.) Refresh Logic and when to update the AdapterRecylerView when user location changes.
7.) Whether to allow users to add their own bathrooms that they discovered.

Next, we discuss the solutions. We solved (1) by catching API errors and handling them through our try catch pipeline. Any API errors occurring due to QPS or API not available is handled gracefully and all calculations are substituted with "N/A" values. For (2), we decide to get all our data parallel by spawning threads. Although such a

process consumes a huge amount of CPU, it significantly decreases the runtime as the fetch is obtained parallelly. (3) is solved by careful composing of LiveData types through a series of permission if-conditions to which we place the boolean flags in certain places that guarantee order of execution without missing any dependencies. (4) took long to debug and figure out. Originally, we wanted the map to be available throughout the entire application, thus we placed such map in a companion object. However, navigation away does not guarantee that certain states are saved in companion objects (especially graphically heavy objects), thus we run into VAO errors. We solved this by avoiding putting any graphical heavy objects into a companion object. (5) was solved through careful debugging and reorganization of fragments. (6) was not entirely solved as we denoted that getting all locations is computationally expensive. Moreover, we are uncertain the time interval the user may move significantly from their original location. It would not be ideal for loading bathrooms/checking user current location if it is computationally expensive. Henceforth, we left it to manual where the user can swipe to refresh on their own regard. (7) was hotly debated as we would not like bathrooms to be placed where it may be seen as offensive or no use (a bathroom that does not exist wastes the user's time). Henceforth, by not allowing bathrooms to be crowdsourced we maintain a level of quality in information given. However, such a trade off occurs in bathroom discovery to which we recognize that we do not cover all bathrooms in BU, let alone BU's other campuses.

1) Multiple API keys associated with different accounts are rotated each time a key has hit its limit. Alternatively, one can wait for the API key to reset every 24 hours.

Future Work

For future developments, the team would like to raise fundings to get over the maps' API paywall. Currently, the free version of TomTom Map is used with strict daily call limits as well as per-second call limits. Google Maps, though a paid API, has more detailed documentation as well and broader community support due to its more widespread usage. The team would like to move over to Google Maps API from the free version of TomTom if funding wasn't an issue. The concerns of false information provided by individuals with malicious intent is too great for the feature to be implemented in the application's current state, however, crowdsourcing bathroom information is also a feature the team would like to consider going forwards. If the feature is realized, it would enable the service scope of the application to expand outside of BU's campus. Furthermore, with more time, the team would also like to implement an open-and-close-times filter for each bathroom. This would require the team to research the operation times of each BU build during different times of the year. However, it would allow users to filter for bathrooms that are open during a specified time.

FlushHub is also available in Thai!

Citations

- https://developer.tomtom.com/maps/android/introduction/introduction#location
- https://developer.tomtom.com/routing-api/documentation/routing/calculate-reachable-range
- Question asked to LLM (Chat-GPT4)
    - How do I parse string in kotlin separated by , ?
    - How would I reference my view model from an adapter in kotlin?
    - Can we use 2 different bindings in the same class?
    - How would I properly round a float to the first decimal in Kotlin?
    - How would I index a recycler view based on a float attribute?