Project Report

Mobile and Ubiquitous Computing - 2019/20

Course: MEIC

Campus: Alameda

Group: 20

Name: Mara Caldeira	Number: 83506	E-mail: mara.caldeira@tecnico.ulisboa.pt
Name: Lucas Lobo Fell	Number: 86464	E-mail: lucas.fell@tecnico.ulisboa.pt
Name: Rúben Silva	Number: 96999	E-mail: RubenAGSilva@tecnico.ulisboa.pt

(PAGE LIMIT: 5 pages – excluding the cover)

1. Features

Describe which features stated in the project specification were implemented. Fill out the following table. For each feature, indicate its implementation state. If partially implemented, describe what was achieved.

Component	Feature	Fully / Partially / Not implemented?
Mandatory Features	List Food Services by Campus	Fully
	Show Food Service Information and Menu	Fully
	Show Dish Information and Photos	Fully
	User Profiles	Fully
	Dish Information Submission	Fully
	Dish Photo Submission	Fully
	Automatic Campus Selection	Fully
	Food Service Queue Estimation	Fully
	Photo Caching	Fully
	Cache Preloading	Party: pre-loads everything
Securing Communications	Encrypt Data in Transit	Not implemented
	Check Trust in Server	Not implemented
Meta Moderation	Data Flagging	Not Implemented
	Data Sorting and Filtering	Not Implemented
	User Trustworthiness	Not Implemented
User Ratings	Dish Rating Submission	Not Implemented
	Average Ratings for Dishes / Food Services	Not Implemented
	Rating Breakdown (Histogram)	Not Implemented
User Accounts	Account Creation	Fully
	Login / Logout	Fully
	Account Data Synchronization	Fully
	Guest Access	Fully
Social Sharing	Sharing Food Services	Fully
	Sharing Dishes	Fully
Friend Tracking	Location Sharing Option and Selfie	Not Implemented
	Show Photos of People at Food Services	Not Implemented
	Sort Photos by Friend Likelihood	Not Implemented
User Prompts	Prompt Upon Queue Entry	Fully
	Prompt Upon Queue Exit	Fully
Localization	Translate Static Content	Not Implemented
	Translate User Submitted Content	Not Implemented
Dietary Constraints	Dish Categories and User Diet Selection	Fully
	Dish Filtering	Fully
	Food Service Filtering	Fully

Optimizations

Describe additional optimizations that you have implemented, e.g., to save power and improve usability

2. Mobile Interface Design

The application starts in the main activity which displays every food service available at that time for the given status according to the dietary constraints and campus. From the context menu, the user has the option to turn off the dietary constraints within the whole application until it is turned back on.

If the user clicks on the profile icon a new activity will open containing the user's settings (status, campus, dietary constraints, automatic location), and the ability to change them. A login or sign up button is also available. By pressing that button, it will take the user to the login screen which allows the user to create an account or login with an existing one. All activities include a back button which lets the user return to the previous activity.

Back in the main activity, when the user clicks on the available food services, it displays the queue time and walking time to the destination. It also includes a static a map with a marker where the food service is located. Below the map, the food service's menu including all the items is displayed, again according to the dietary constraints. By clicking on the map, a new activity which displays the map in full screen is opened. It calculates the path from the user's current location to the food service's location. By clicking in the food service's marker, it will display how long it will take to arrive as well as the distance.

By clicking on the button to add a new menu item, user is taken to a new activity which he can fill the details regarding that item (price, food type, description). After saving it will take the user immediately to newly created menu item activity.

Back in the menu list, if the user instead clicks on a menu item, the menu item's activity will be opened. It displays its information (price, food type, description), and a gallery of user pictures. Besides this, it has a button which lets the user add a new image photo for that item, which takes the user to their phone's gallery when clicked. If the user clicks on a photo it will be displayed in full screen.

3. Server Architecture

3.1 Data Structures Maintained by Server and Client

The client and the server take different approaches at organizing the data. As the client is the focus on this project, this is where the data model and logic are fully implemented, while few shortcuts were taken for the server. Persistence was not implemented in either of them, but in a real-world application this would not be the case. Some more details regarding persistence are explained in section 6.

The client holds a lot of application logic (e.g. checking whether a specific FoodService is open for the given user) which removes a lot of responsibilities from the server. This means that even in the classes that are common between the client and the server, on the server side the class is very simplified and only exists as a layer between the data and the API resource.

The three common classes are *User*, *FoodService* and *MenuItem*. An additional data structure exists to store images.

3.1.1 User

For the User class, the client keeps a single instance with the current user's username and password (if logged in), queue information (which will be explained further ahead) and their user settings such as status, food restrictions, and campus. These user settings have values which are known a priori, which means the best approach is to use Enums to represent their possible values (an explanation for this will be made in section 4).

The server on the other hand keeps a hash map of all users with their username as a key. This makes it simple for fetching the user when needed. The User class holds the username, the password, the status, and a list of constraints of each saved User. For simplicity, the status and constraints are converted in the client from their respective Enum value to a fixed String representation and vice-versa to be stored in the server.

3.1.2 FoodService

The FoodService class keeps most of its information defined directly in the client, as this information is static for the context of this project. Some reasoning on why this approach was chosen (and why it might not be optimal in a real-world scenario) is explained in section 6. Everything regarding location and schedules never passes by the server.

The client stores two hash maps for FoodServices. The first one stores all the existing FoodServices and uses their IDs as a key. FoodService IDs are sequential and no automatic coordination is done to ensure the client and server FoodServices match. For simplicity, this was done manually (i.e. creating the same number of FoodServices in the server as are present in the client). The

second one simply maps the beacon names present in the specification to the FoodService IDs and is used for queue processing on the client. Each FoodService also holds information regarding its own queue time, which is fetched from the server at run time.

The server holds a hash map of all the FoodServices with their IDs as keys. In the server, FoodServices hold a HashMap with all its corresponding MenuItems using their IDs as a key. The ID is also sequential, and no overlap is ensured by using Atomic Integers. A second hash map stores information about the queue joins and number of people within the queue at the time of joining.

3.1.3 MenuItem

Both the client and the server's FoodServices hold a hash map of all its MenuItems with their corresponding IDs as a key. In the MenuItem, the same data is stored in both sides and these are completely crowd sourced. The images are not stored directly in the MenuItem, and a set of image IDs is stored instead. When needed, the image IDs are used to fetch the images.

3.1.4 Images

The client stores the images in an LRU Cache with the corresponding image ID as a key. When missing, the client fetches the server instead. In the server, for simplicity, the images are saved in a simple hash map with the same structure of the cache.

3.2 Description of Client-Server Protocols

The server protocols are simple and allow for all the basic communication and data exchanges to happen. The list of the existing protocols is: User Register, Save Profile, Fetch Profile, Save Menu Item, Fetch Menu Items, Save Image, Fetch Images, Join Queue, Leave Queue, Estimate Queue.

Only the profile requests require authentication and all other requests are essentially anonymous, as the only inputs they take are what is strictly necessary. The list is not extensive and does not allow for varying granularity, for allows for everything required by the specification. A few helper functions exist in the client and the server which helps convert the data from one format to the other. The API specification of how each request works is defined in the appendix. The datatypes are defined in the contract which is present in both the Server and the Client. Most requests return status messages which indicate whether the request was successful, or an equivalent way of inferring that.

3.2 Description of Client-Server Protocols

The requests which save and fetch images from the server, stream the messages instead of sending them in a single request. The images are converted into multiple byte arrays which are split into chunks and sent over the network with their corresponding image Id and position in the array.

4. Implementation

The server and the client are decoupled in the sense that either can work without the other. A new server can be created for the client, and the multiple new clients in different platforms can be created as well (desktop app, web app, etc.). The server is implemented in Java, uses the gRPC framework for communication and makes use of Apache Maven as the software management and comprehension tool. On the server side, a single endpoint exists and is implemented in the GrpcServiceImpl class. On the client side a flexible approach that could handle all kinds of gRPC requests was chosen. First, we have the GrpcTask class which defines the general skeleton for communicating with the server through a background thread and executes a GrpcRunnable. This GrpcRunnable is a generic abstract class which allows for either blocking or non-blocking requests and is later implemented individually for each different type of request from the API. The advantage of this approach is that is allows us to define a rigid structure for back-end requests while still while still maintaining all the flexibility of allowing the caller to implement the call-back function. Having one class per request brings multiple advantages. It avoids having a big class where all requests are implemented, which makes it makes easier to add, remove, or maintain requests, by having different responsibilities separated in different files. It allows each different request to implement its own constructor, allowing variation in the input classes. More importantly, it allows the class to include a generic result, where each individual class can define what type each specific request needs to expose.

To manage the state, we extend the Application class, which is alive during the whole lifecycle of the Application and which all activities have access to during their execution. In this class we initialize all food services, receive the broadcasts from Termite regarding all the beacons which were found in range as well as store the User object that is currently running the application. This class is also the one responsible for creating the prompts that the user receives whenever a queue is joined or left. Within the whole application, all activities share a global state, and do not make local copies of the data. RecyclerViews have their own lists but the contents are always the same object references. These lists need to be manually updated whenever a change occurs. As for communication between activities, the idea is simple. When an object needs to be passed, an intent is created which states the global ID of the required object. Then, the activity accesses the global Data instance to fetch this object directly.

Whenever Termite broadcasts to the application that a FoodService beacon is close by, the application sends a request to the server to join the corresponding queue. Later, when the FoodService beacon is no longer present in the list of devices, the leave queue request is sent with the corresponding queue time.

As stated before, we used Enums to define variables whose values were known a priori. This allows for a higher flexibility when implementing the logic and displays of the application, for three main reasons. The values become irrelevant as simple comparisons can be made (i.e. user status is equal to schedule allowed status) instead of doing one on one checks. Displays are easier to make as iterators can be used instead of manually writing all possible values. The resource-ids can be stored together with the Enum which allows for easier String changes and translations.

Other libraries used were the Google Maps API and the Google Directions, as to allow a user to map its way to the desired food service and calculate the distance to reach the wanted food service. The requests for the APIs are again done in a background thread to avoid interfering with the UI. To speed up the process, an invisible dummy map instance is placed within the main activity to start initializing maps before they are needed. This allowed for the Food Service activity to load much quicker.

Lastly, we also used the ThreeTen-Backport library that allowed us to have access to the Java 8 time, which is normally directly available on all versions of Android, in order to infer and use the correct time and date during the course of our application. Another possible approach would be to use official Android Desugaring which allows for some usage of Java 8 classes, but for the context of the application the first option is simpler.

5. Running the Prototype

To have the prototype completely running, one should run the server first. For this, Maven needs to be installed. Afterwards, *mvn install* should be run in the /Server folder. If necessary, this step should be repeated in the contract and in the /Server/server. To run the server the command is *mvn compile exec:java*. This should be enough for the server. To run the Android application the one necessary tool is Android Studio and an appropriate emulator (e.g. Android 29).

6. Simplifications

All IDs in the application are generated in the server and issued sequentially. This makes it easier for testing purposes, but is not a good idea for deployed applications, as users can infer information from these numbers. Ideally, a random ID is generated and checked against used IDs.

As stated before, persistence was not implemented in the client nor the server. In this case, whenever the client is turned off (i.e. the application is closed and eventually removed from memory), all data is lost and must be fetched from the server again in the next use. If the user does not have an account, user settings are lost as well. In a real-world case, the device should store locally at least the user settings. Additionally, food services and menus could be stored in an SQLite database with the assistance of the Room library, while images could be stored in a cache. This would improve overall user experience, as it would allow for faster load times and unnecessary data and battery usage. In a real-world scenario, it would be mandatory to have persistence on the server.

Since this project is focused on the client, FoodService information is stored directly in the client and never passes through the server. A lot of domain logic is handled directly on the client as well. This means that the server does not need to implement anything regarding locations, opening times, food types, etc. Because of this, the server is very simplified and only exists to store data from the users and trusts the information is correct. In a real case, food services would be stored on the server as well, together with all the information mentioned before. This would make the server more complex in terms of data structures and would imply a longer development time. The API would need to be more robust to handle these differences.

Security was not considered in this project and communication is done via plain text. The system itself does not require high security but the current model does not guarantee a minimum-security level. Passwords are stored both in the client, for repeated uses in the requests, and in the server, to verify the user's authenticity. Normally, a few changes would be required for this application. First, communication should be done via SSL/TLS, in a way that packet information cannot be intercepted in transit. The password, together with the salt and pepper, should be hashed and stored in a database. The salt should be unique for each password and stored in the database, while the pepper should be a constant stored in the server code. Lastly, the client should not store its own password. Instead, on login, the client sends its authentication which the server validates using the salt, pepper, and hash. In return, the server sends a temporary access token which the client can store and use for future requests. This approach means that in case the client is compromised the password is not compromised as well.

When creating an account, the password is only asked for once. This might lead to the user making a typo and not realizing. No e-mail is requested either. Once created, account auth information cannot be updated. Username and password are final. Normally,

it should be possible to update at least the password. It is also not possible to delete the account. In real apps these problems should not exist. Additionally, when the saves profile request is made and for some reason the Auth information is wrong, the application logs the user out instead of issuing an error. This is not a problem in this case since the save profile request only happens after login, and after logging in it is impossible for the Auth information to be wrong (as it cannot be changed). Nevertheless, in a real case, this situation should be handled.

For the queue times, a few simplifications were made too. When a user sends a queue join request, the server assumes that sometime in the future it will receive a queue leave request at the appropriate time. This means that the server does not do periodic checks to remove "stuck" users that never sent the second request. While this does not affect queue times, as each join queue request is given a unique queue id, over time some phantom queue ids will be present in the database. For testing purposes, it is the client that calculates its own queue time and sends this to the server on the queue leave request. Realistically, it might be a better option for the server to do the calculation instead based on request arrival times. Queue times are stored in seconds and should be displayed in minutes, but to make visualization easier during testing, the display is in seconds as well. Lastly, all queues are updated from the server every 10 seconds. Normally, the client should not ask for updates in such a short interval, nor should all FoodService queues be queried, but only those present in the current display. This could be handled by using RecyclerView onBindViewHolder function.

Most server requests are very strict in what they allow, and the API is not very extensive. For example, when requesting menus, it is only possible to specify the target FoodService. Ideally, it would be possible to specify one specific menu item (and not all from that food service), or all menus from all food services. When estimated queue sizes, it is also only possible to request it from one FoodService, instead of being able to specify which FoodServices the client is interested in.

During the startup of the Application, it requests everything the server has regarding FoodServices menus and images. Ideally, FoodServices that are not relevant (i.e. different campus, are not compatible with dietary constraints) should not be fetched. When the user loads the application, only information regarding visible FoodServices (and those the RecyclerView has decided to populate) should be fetched. Same thing applies for queues and distance calculations to the map APIs.

7. Bugs

As stated, all images and menus present in the server are fetched as the app is loaded, instead of doing gradual requests as needed. Additionally, in some cases it may happen that a result is fetched from an API after the activity has finished loading and the result is only displayed after the activity is created or resumed again, for example when the app is opened through a shared URL.

The project package was accidentally created with another group number, 16. Our group number is 20 as specified before.

8. Conclusions

All in all, this project is quite wholesome in the fact that allowed us to work in multiple basic needs that every application available online should have.

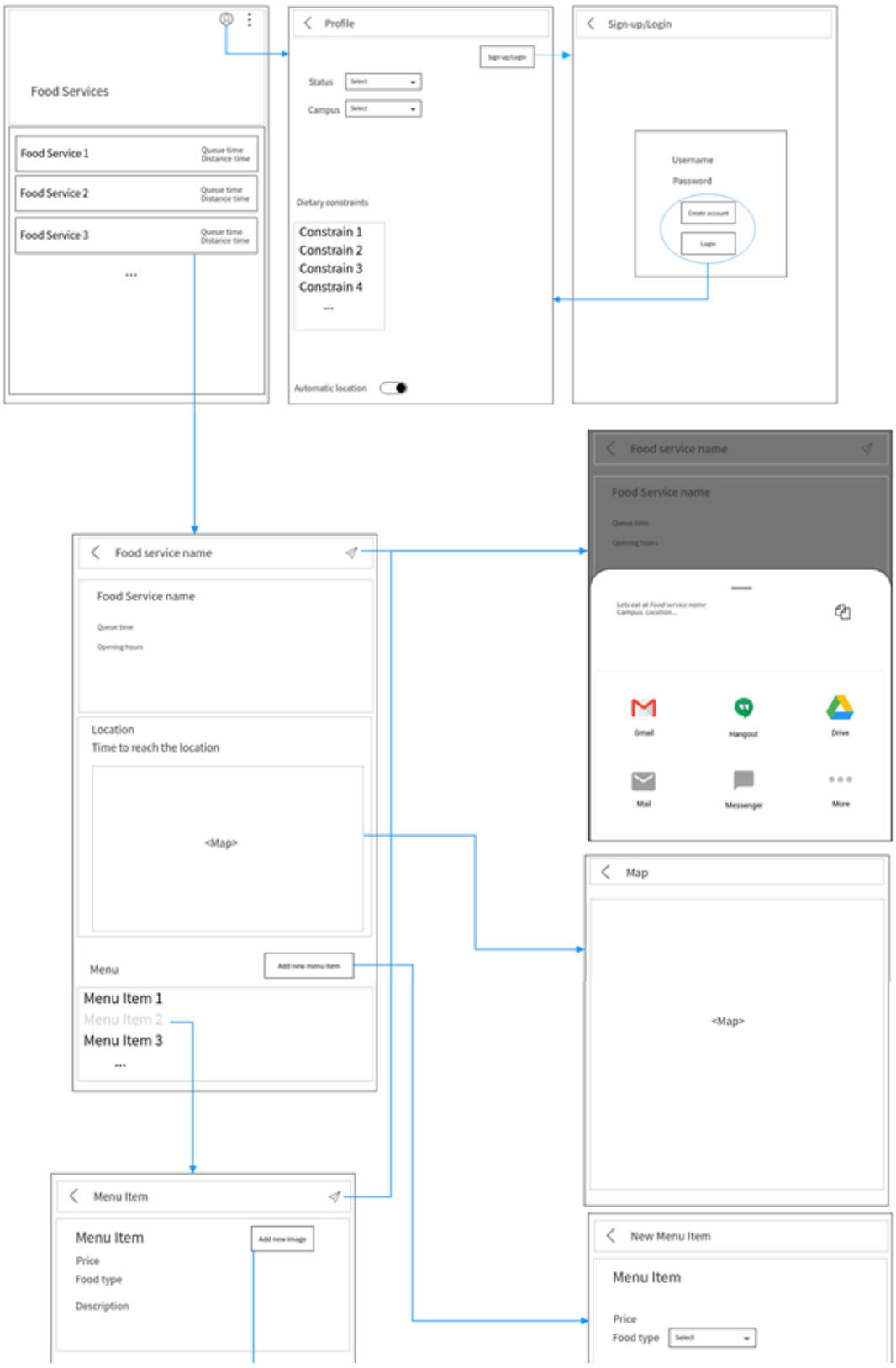
It gave us a clear view of some architecture components of android and how to properly use them in a more efficient way, having in mind the fact of how to conjugate said components with external libraries of google or other external entities, that may be useful whenever a project of this type is in question.

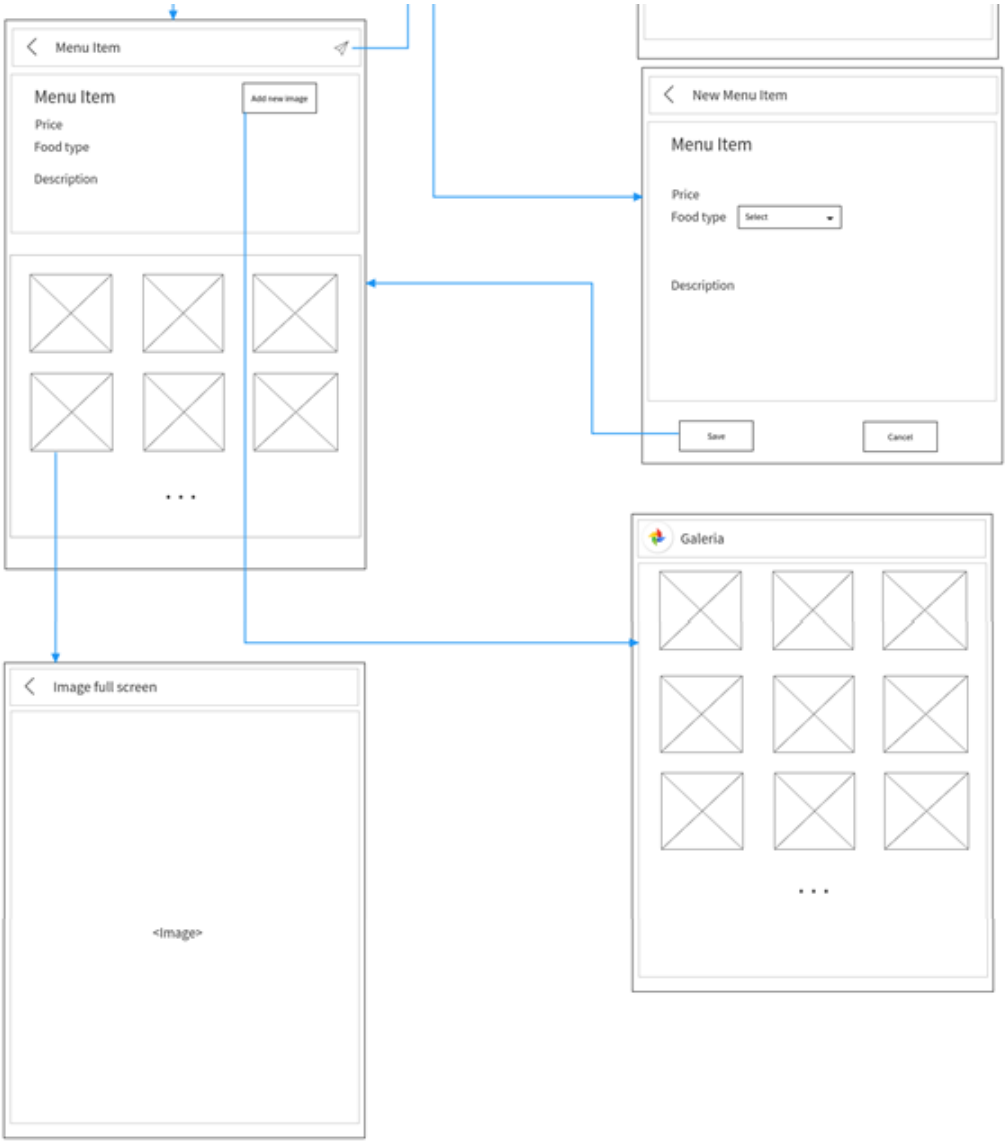
Nevertheless, the "world" of android is far too vast for us to completely grasp every main aspect of it, therefore, we are aware that there is much to be learn on this field.

On an ending note, we feel like it would be beneficial to the project to create a division of the mandatory features by sequential steps defined by the course faculty. This way, a student that never worked with android before this course, would feel more structured on the necessary main functions that he needs to implement and so, would be more clear what is the correct base structure of any android project, not skipping ahead any concerning implementation step.

9. Appendix

Wireframe Diagram





Server API

1. User Register

The client sends a message with Auth information, which the server validates. If the username is valid (i.e. not taken), the request is successful and “OK” is sent to the client. Otherwise a “USERNAME_TAKEN” is sent, which the client uses to update the user interface with a message asking for a different username to be chosen.

2. Save Profile

The client sends a message with Profile information Auth information and, which the server validates. If the auth information is valid, the profile is updated and “OK” is sent to the client. Otherwise a “USERNAME_DOES_NOT_EXIST” or “INCORRECT_PASSWORD” is sent instead.

3. Fetch Profile

The client sends a message with Auth information, which the server validates. If the auth information is valid, a profile is sent back together with “OK”. Otherwise a “USERNAME_DOES_NOT_EXIST” or “INCORRECT_PASSWORD” is sent instead.

4. Save Menu Item

The client sends a message with a new menu item and a FoodService ID, which the server validates. If the FoodService ID is valid, the MenuItem is saved and a new MenuItem ID is generated and sent back to the client together with “OK”. Otherwise, “FOOD_SERVICE_NOT_FOUND” is sent instead.

5. Fetch Menu Items

The client sends a message with a FoodService ID, which the server validates. If the FoodService ID is valid, all its menu items are sent back to the user. Otherwise, an empty request is sent (ideally and error code should be present as well).

6. Save Image

Images are implemented as streams, in which client sends a series of messages. One of the messages must include the image metadata, which is the FoodService ID and the MenuItem ID. The other messages are ImageChunks which include the data bytes and chunk position. The server combines the chunks into a full image and stores it. The server sends back the generated image ID.

7. Fetch Images

The client sends a list of image IDs it wants to receive. The server takes the images, splits them into chunks, and sends a stream of ImageChunks to the client. In this case the ImageChunk includes data bytes, chunk position, and image ID. The client combines all chunks (putting them in their corresponding image and position) and stores the images.

8. Join Queue

The client sends the FoodService ID it wishes to join. The server sends back the generated anonymous UserQueue ID.

9. Leave Queue

The client sends the FoodService ID it wishes to leave, the UserQueue ID, and the time in seconds it spent in the queue. The server validates the UserQueue ID and answers “OK” if valid, “USER_NOT_IN_QUEUE” or “FOOD_SERVICE_NOT_FOUND”.

10. Estimate Queue

The client sends the FoodService it wishes to estimate. The server sends back the time in seconds if possible, -1 otherwise.