T'Go

Final Report

May 17 22

Professor Yong Guan
Andrew Marshall - Team Leader
Michael Phippen - GitHub Manager
Alex Daly - Key Concept Holder
Jacob Spoelstra - Database Manager
David Lauria - Communication Manager
Darrall Flowers - Website Manager11
may1722@googlegroups.com
Team Website
may1722.sd.ece.iastate.edu

Table of Contents

1. Revised Design

1.1 System specifications

The only hardware that was required to make T'Go was that we needed a server. We were able to set one up going through Iowa State's Electronics Technology Group (ETG) using some of the space on Iowa State's servers. It holds our MySQL database, and uses Apache and PHP to send queries between our database and our application. The pros of setting up a server this way was because it had no cost to use and we were sure it was setup correctly because we were working with people who have more experience at setting up servers. The only con we have to this setup is that the only way we can access the server outside of the campus network is to VPN into it.

On the software side, we programmed the application in Android Studio. There are a couple of API's that we used to help manage the operations. The first one is Stripe, an integrated payment API that allows integration with Android and iOS. Using this, we are able to record people's payments, and for the purposes of the beta tests, we were able to mock credit card information. We used Google Maps' API to help form routes and help the courier navigate. While we were not able to simply open up Maps within our application we are able to jump to it from a job screen to a map of a route from with the courier's location to their desired end point. For encrypting important data, we used some special classes from Android's libraries. MessageDigest was used to salt and hash users' passwords for storage while Cipher was used to encrypt and decrypt the tokens created by Stripe to hold users' payment information.

1.1.1 Non-functional

One of the non-functional requirements we have for T'Go is how fast it takes for an instance of the application to communicate with the server. From the current beta, we are seeing that it takes in a few milliseconds to run calls to the server which means users are not constantly waiting while they are running the app. On the phone, we wanted to minimize the amount of energy and data required to run the application while we do not have any definite results on the amount of memory used since the application has to be run while connected to the campus network, we are seeing that it is using minimal power while running.

1.1.2 Functional

T'Go has four different classes of users each one will have functions specific for their task and the ability to do tasks of those at lower classes. Our lowest class is the User. Users have the main functions of being able to post jobs, submit reviews of couriers, and apply for a higher class of user. The next class is Couriers. They have the ability to accept jobs that the users have posted along with all of the other functions that a User can do. The next higher level is the Moderators. They will have the ability to review jobs to make sure Users and Couriers both respected their sides of the transactions and the ability to verify users are who they say they are. They are able to search past jobs though a special function that searches based on getting the usernames of the User and the Courier. It returns a list of dates that when clicked on opens up the job details page for the moderators to see. The final class of users are the Admin. These is specifically reserved for

the team members since we are the ones developing the application and have the access to the programming and database outside of the application. Within the application, Admin get the same functionality as the moderators.

The next functional requirement we have is being able to process of a job, which is detailed as follows. First, a User will post a job to the job board. They will specify what they need, where they need it delivered, the time they need it, and they suggested price of what the whole job will cost to the customer based off of the items' prices and travel costs. They will select a payment method currently, we only have it setup as credit or debit card. Next, a Courier can look at the job board and select one. They will have the ability to see the details of a job before choosing to accept it. After the Courier has selected the job, the database will be updated to say that the job is taken by the courier. A new button will be displayed allowing the User and Courier to communicate with each other in case changes in the job need to be discussed. For example, if a specific product the User wants is sold out but a substitute is available, the Courier can ask the User if they would be ok with the substituted item. Once the Courier arrives with the user's requested items, the job will be verified as complete by both the User and the Courier. The User will be given a 6 digit code that the Courier will have to input. Payment will then be handled, and the server will then mark the job as completed.

1.2 Design/Method

Our method for making the application was to first setup the functionality of the application that did not require the server. This was mostly setting up how the User Interface would look for the different activities and how the users would be able to move between activities. Towards the end of this, the server and database was set up. We were then able to start building the functionality of the application that dealt with interacting with the server. We split up the various tasks between team members to minimize the amount of dependables members would have to wait on others for. Most of the team was responsible for making their own php programs on the server when a function required it. Once we were able to completely run a job through the application, along with setting up the messenger system, class upgrade requests, and review functionality, we were able to start the beta testing for the application.

A GIT repository was used to allow for group programming. Whenever a team member wanted to work on a new feature for the application, they needed to make a new branch off of our master branch. Once they finished implementing the feature in the new branch, they would notify our GitHub Manager. At the next group meeting we had, the two would then sit together and in the branch where the new feature was made, they would merge it with the master branch so that they could see if the new feature worked without breaking some other part of the application. Once it was verified that it did not affect anything else, the new branch was then merged back into the master branch for everyone else to pull into their own local repositories of the application.

1.3 Design Analysis

We are in the beta testing stage of the application. Our current findings will be discussed in later sections. We are in the process of fixing bugs as they are found and taking into account user feedback on how the application could be improved. At this moment, we are able to run

through a job completely with payment along with some additional features like allowing User and Courier to chat. We spent most of the project worrying about getting the functionality of the application ready over worrying if the aesthetics looked nice. We figured functionality was the more important of the two, so now we are taking time to improve the aesthetics.

2. Implementation Details

2.1 Basic Database Calls

There are many times that the application will need to query the database, either to return some result or to insert new data to track jobs and user information. The main process for handling this was to set up asynchronous tasks for the different activities to run. At the start of one of these tasks, a PHP program on the server is specified and the parameters needed to run the query are assembled by a Uri Builder. The PHP program is then run on the server and the results are sent back to the application to handle as it needs. For example, when a user is logging into the application, an Asynchronous task called AsyncLogin is run. This specifies the needed URL is http://may1722db.ece.iastate.edu/login.inc.php, and we assemble the parameters of the given username and encrypted password. The server then runs login.inc.php and if successful, returns the user's information, such as their id number, username, and email. The returned variables are added to the intent to open the Profile Activity, which handles the display of this information to the user.

2.2 Password Encryption

To protect user accounts, a system was setup to encrypt passwords. The object PasswordEncrypter was implemented using Android's MessageDigest to form hashed passwords from the passwords it was given. In the SignUp Activity, it is called upon after we confirm that the user's desired password. The MessageDigest is initialized and then a random number is chosen to salt the password with. Once the Digest is updated with the salt, it then hashes the password with the SHA-256 hashing algorithm. SHA-256 was chosen because there has yet to be a way to break its algorithm other than running a Brute Force Attack through it with every possible password and comparing results, making it a very secure option. The salt value adds even more complexity and time needed to crack to this and strengthens the security of our passwords. After it is formed, the encrypted password and its salt value are stored within our database in our users_table.

PasswordEncrypter is used slightly differently when it is utilized in the Login Activity. First, the username is compared to the values stored in our database to see if it exists. If it does, we return that user's salt value for encryption. The MessageDigest is then initialized with the given salt value, and the given password at login is then encrypted. The output is then compared to the encrypted password stored within our database for that particular user. If they match, the user is allowed into the application. Otherwise, they have to retry logging in again.

2.3 Creating a Job

The idea-driven on creating a job was to link a user to a courier to complete a task. To do this we had to first take the user input on what is wanted to be delivered. More specifically the items they needed picked up. So we begin this process by having the user login to his/her account then make a selection of the items they want to pick up and the quantity of the item.

To start creating a job, a user must first submit their desired drop-off location and the date and time they want the drop-off to occur. Upon submitting, the asynchronous task AsyncAddLocation is run to the first check if the location has been added to the database already and if it has not, add that location to the database. The location's id number will then be returned to the application. With the successful running of AsyncAddLocation, the next asynchronous task, AsyncAddJob, is run to add the job to the database. The values stored here are the user's id, the date and time the job was created, the data and time the job needs to be completed by, and the location id the drop-off place was given. When the task is finished, the Add Item Activity is started so the user can choose what items they want to get.

2.4 Adding Items to a Job

When a user is adding an item to an order, the application was made it so they could choose from a preset or create a custom item. If they want to choose from a preset, the asynchronous task AsyncGetProducts is run to retrieve the presets from our database and then displayed. After an item is choose, the application populates a listview using a custom item for the list items. This item then allows the user to edit it by changing the quantity of the item that they want, or delete the item from the list completely. Changing the items in this list also changes the price being shown. When the user accepts, puts in payment information, and submits, the items are inserted in the database with an association with the created job by running the AsyncAddItem asynchronous task for each item.

2.5 Chat

For the Chat Activity, there are two tasks that are required to make it function properly. The first task is the SendChatRequest task. It is a simple task that is activated every time the user presses the send button and sends the chat id, user id, and message text to be inserted into the database. The other task is the MessageUpdater task. This task is continuously checking if a new message has been sent between the users in the given chat. Every time it finds a new message, it signals the application to update the screen with the new messages appended to the bottom. If nothing was found, it would jump into the next iteration of this task. Since this is always running while the Chat Activity is displayed, there was an issue where the application would freeze when the chat was opened. To fix this, the thread where MessageUpdater was being run was told to sleep for a half second at the start of every iteration. This allowed other tasks like SendChatRequest to be run when they are started. The sleep was set to a half second in length because it gave the application enough time to register other actions the user may take at this time while only slightly delaying the time between when a message is sent and when it is shown on the message board. The delay should not be noticeable to the users.

2.6 Two-step Verification

To provide a way for both the job's creator and courier to verify that the job has been completed to the satisfaction of both parties, marking a job as complete requires actions from both. To verify the job is complete (and authorize payment), the user requests a 6-digit code and

the courier must enter the code on their device. To cut down on queries to the database, instead of randomly generating a code each time the user requests it and storing it, the code is generated using the information specific to the job. This allows identical codes to be generated locally on the requester and courier's devices, though only the requester can see it. This verification system was added to help mitigate potential problems with have one user verification, such as a courier delivering goods without the job being marked complete by the requester (and payment not being sent) or a courier marking a job as complete and receiving payment without delivering the goods to the requester's satisfaction.

2.7 Payment

For payment, the application uses a service called Stripe. Stripe gives an easy to use API that allows us to receive payments from all users. When setting this up, an account needed to be created to get a set of test and production keys to use in the application. The test key was used so that payments could be received without being charged a fee or exchanging real money during the beta phase. In order to send a payment through this test key, the user will have to use the card number 4242 4242 4242 4242. If a valid card number is used, it will fail. This card information is then encrypted using Android's Cipher and stored on the database to later be used again using AsyncUpdateJob. When it comes time to pay the courier, AsyncSendPayment is utilized to handle Stripe's payment methods on the server.

2.8 Google Maps API

In order to help couriers find stores and drop-off locations, Google's Google Maps API was chosen to use to navigate them. It was chosen because it was reliable and free to use. To implement the API, an API key had to be acquired from Google. The key is stored in the application's resources for later use. It allows the application to call the API's functions to retrieve data for locations, directions, and navigation for the couriers. In action, when the courier wishes to navigate to a particular location, they will tap on the "Go To Map" button in the Job Details Activity. This opens a screen that enables searching for a location with auto-suggestions. When the courier makes a selection, it will open Google Maps with directions to that place from their current location. The courier can also give location permissions to that application which will allow them to access "last locations" the application has gathered. This "last location" is used to determine the courier's location.

2.9 Upgrading Account

When a user or courier wants to upgrade their account to the user level above them, they can submit an application which will notify a Moderator or Admin to review them. When reviewing the application, the moderator will be reviewing the user's history and the reviews they've received to determine if the user generally provides a high-quality service. If they are approved, the user's level is upgraded and different options are opened to them in the Profile Activity.

3. Testing Approach and Results

The primary way we tested was using the test-driven development model. Since we all have worked in the professional world throughout our collegiate experience, we found the test driven development model efficient for this project. We started off by creating unit tests based upon use-cases and the main functionality of the application, such as login and creating jobs. Then we wrote unit tests before we even started coding to ensure we hit our marks for each sprint throughout the development cycle while minimizing bugs and errors.

The secondary way we tested our project was creating a beta for our friends to use. In creating this beta it gave us the ability to take in user feedback. Such as different aspects of the user interface that needed to be altered or in the worse cases glitches that would occur. We ended up with the following findings. First, there were some phones whose aspect ratio caused issues with seeing all of the features on the profile screen. While implementing the application, we tried to use as many phones as we could for testing, but with the beta, we were able to see how they acted on the phones we did not have access to. Secondly, we found that there were some people who had issues logging into accounts they had made. After spending some time looking at their information, we were able to see that their hashed password contained an apostrophe, which caused the query to break. This occurrence is rare, but we were able to implement a simple fix to work around it. Finally, people recommended we update the aesthetics. We were planning on doing this anyways but getting their input on how was useful.

The final way we tested our application was by placing the application on different phones and Android updates. In doing this, we were able to see where our code lines had depreciated or were too advanced for some android devices on the market. Throughout all of this robust testing, we were able to discover the shortfalls within our app and keep pace with our weekly sprints while remaining user focused throughout our app.

Appendix I: Operation Manual

A guide to setup, demo, and test the T'Go Android mobile application.

1. Download Project

Begin by downloading the Android project at
https://github.com/drallflowers/T-Go

2. Install Android Application

Once you've downloaded the project, you will need to do one of the following:

a. Import into IDE (Android Studio used for development) and run app on emulator or physical device.
https://developer.android.com/studio/intro/migrate.html
contains the directions needed to import into Android Studio IDE.

b. Install app file to physical device.
https://developer.android.com/training/basics/firstapp/running-app.html
contains the directions needed to install the app file.

Note that, at the moment, the servers the app is communicating with are provided by Iowa State University, and as such cannot be connected to from off campus without using a VPN. Keep this in mind when running the app on a physical device or on an emulator if your machine is not connected to the Iowa State network.

3. Signup/Login

Once the app starts up, you will be presented with a login page asking for username and password. A new account can be created by tapping the "Signup" button and filling out the username and password fields, or the following accounts can be used for testing at three different privilege levels:

a. User
Username: test_user Password: tucker_blue

b. Courier
Username: test_courier Password: sarge_red

c. Moderator
Username: test_mod Password: wash_freelancer

4. Create Job as User

Once you are logged in, you will be taken to the profile page, which serves as a hub for the different activities the T'Go app supports. A user of any privilege level can create a new job by tapping on the "New Job" button. Doing so will open a form with several fields that need to be filled out. Fill in Street Address, City, State, Zip Code, and Apartment Number (if applicable) and select a Date and Time for delivery deadline before tapping on "Continue" to advance. Next, you will need to add items to your order. Items can be added either by selecting from presets in the database after tapping "Presets" or items can be manually added to the order by tapping "Add Custom Item" and filling out the form that displays. Once all items you want have been added to your order, enter your payment info before tapping "Submit." The job is now posted to the Job Board and can be seen by users of Courier rank and above. Additionally, the user who posted the job can see its details if they return to the profile page and tap "My Jobs."

5. Locate and Accept Job as Courier

Only users of Courier rank and above can find and complete jobs not posted by themselves, posted on the Job Board, which can be accessed by tapping on the "Find Jobs" button on the user's profile page. A list of jobs will be displayed and the Courier can view each job's details by tapping on them. If they see a job that they want to accept, they can tap the "Accept Job" button and they will be added as the job's courier. The job will now be removed from the Job Board and can be accessed only by the requesting user and the accepting courier through their respective job lists.

6. During Job

While the job is ongoing (accepted, but not yet completed), the requester and courier can chat with each other by tapping on the "Chat" button on the job detail page. This allows them to communicate if, for example, the exact item the requester asked for is out of stock and the courier is willing to pick up a similar item. The courier can also access Google Maps from the job details page to help them navigate to the destination the requester has specified for delivery by tapping the "Go To Maps" button.

7. Complete Job

Once the job is complete (requester has received requested goods), the requester and courier must register the job as complete together. A six-digit code is provided to the requester after they tap the "Complete Job" button on the job details page and the courier

must input this number into the field provided to them after tapping the same button on their device. The job will then be marked as complete and the payment from the requester to the courier will go through.

8. Upgrade Account

Users of the T'Go application are divided into four privilege levels, in ascending order: User, Courier, Moderator, and Administrator. Users of User or Courier status can request to be upgraded to the next privilege level by tapping the "Request Upgrade" button on the profile page. Moderators and Administrators receive these requests as messages in a mailbox, which can be accessed through the "Mail" button on the profile page and can choose to either accept or deny requests based on the user's history and reputation.

Appendix II: Alternative Versions of Design

Our project really started to take shape and change during the 2nd semester. Initially, for the payment system, we wanted to use a technology called Stellar. Stellar is a newer blockchain technology that allows users to send money to each other directly. This would also allow us to make use of smart contracts. These would have allowed us to have more accountability for users in displaying what the two parties agreed on in the case of a dispute. We found multiple issues with this API. First, we had issues integrating Stellar's API into our application. Even though it is a free API, there were license issues that prevented us from compiling with the technology included. We searched for solutions to this but since it is a newer API, we were unable to find a solution to the problems we had. The result of this issue leads us to a different payment system called Stripe. Stripe allows for easy integration in Android and provided a stable solution for the payment system in our project. The downside to using Stripe is that we lost the ability to keep smart contracts and show what was agreed to in an order. We mitigated this by creating an activity in the application that would allow moderators and administrators the ability to search the database for completed jobs between the two users and pull up the information regarding any specific job. This helped to validate claims.

We also made changes in the way the customer and courier would interact when delivering goods. Initially, we did not have any sort of verification method other than physical proof of a good being delivered or the price being correct in case of a dispute. We quickly realized this was an issue and decided to implement a verification system that the courier would need to use with the customer to be able to say that an order was complete and disburse payment. At first, we thought that a QR code would be the best way to go. While developing this we initially made it so verification would occur with a 6 digit randomly generated code. This was used for testing and making sure everything behind the scenes would work correctly when an order was verified. We then came to a decision that this was actually an acceptable approach for the verification system since we had seen it used in other types of programs where something needed to be verified.

Our design also changed in the fact that we completely added a new communication/notification system for all users. While we have the chat rooms for users to talk to couriers during an order, we did not have a system to handle notifications that a user might have to act on or might need to see. We fixed this by implementing a mail system. Right now, this system allows for user upgrade requests to be handled directly by an administrator or moderator through the app. When the request is sent, the mail shows up in the mailbox for any moderator or admin to respond to. In the mail item, they can accept or deny the request. When a request is responded to, it will remove it for all other administrators and moderators since a request is not needed to be responded to more

than once. This system is easily expandable into other sections of the application for use-cases like order complete notifications and updates.

Security and safety were on our minds when it comes to dealing with the user and courier. Originally, we stated that we were going to provide updates to the user of where the courier was. We were going to provide real time updates like in UBER of where the courier was and status. We then realized that the user may not be using a car, thus if someone wanted to harm a courier, it would be easy to track where the courier is. While most people would not use this for a malicious purpose, it is something that someone should not have to worry about. Our chat room was useful for this problem since it allows the user and courier to connect instantly and provide updates on the status of the order, as well as confirm any price discrepancies to the original order.

We improved our application's security by encrypting passwords for all users. While this was planned, it wasn't planned to encrypt payment information for a user. Like other issues, this was something that became an obvious problem quickly. We thought we could use the same encryption process to store the payment information, but unlike the passwords, we needed to retrieve the payment information from our databases at various times. An issue with this was when we were encrypting it would create special Unicode characters that we were unable to retrieve. While the returned Unicode characters looked correct, it became apparent that the values of the bytes of the Unicode characters were changing from when we had them before the information was sent to the database, to when we had them after we retrieved them from the database. The byte values became an ongoing issue so we eventually decided to fix this issue by constructing a string of the byte values and storing the actual byte values in the database that could then be retrieved to reconstruct the correct encrypted payment information and decrypt it as needed.

In our original design, we had it planned so that a job would be created in one step. We changed this after some thought because our Create Job activity was getting to the point where it was too long. We combatted this so that a user would enter the address, date and time of the delivery on the first page. The user would then get a second page where they can enter a custom item to their order or add an item from a preset list. This made the experience much easier and much more satisfying for the user since you didn't have to scroll through a long page in order to see different information.

Appendix III: Other Considerations

We learned many things while working on this project. During the first semester, we were able to experience what goes into the initial planning stages for a group project. We got to handle researching technology to see what is usable and how to integrate it and learned how to determine a project's functional and nonfunctional requirements. We gained experience in setting and preparing meetings, along with practice on giving product presentations. For the second semester, we got more into how to work on a group programming project. We gained more experience in how to test our application during all stages of development. We learned how to work with others to solve issues that we are having with our tasks and how to schedule tasks to maximize productivity. In the end, we had a project we were proud of and was able to function with the base requirements we wanted from it.