CPSC 490 Fall 2018
John Amadeo Daniswara
Advisor: Kyle Jensen

InTouch

Abstract

InTouch is an Android app that allows relatives and loved ones of inmates to easily send physical mail to inmates for free. The project is comprised of three components: an Android app, a backend server written in Go, and a scraper also written in Go. The backend server, database and scraper are hosted on Heroku, a Platform-as-a-Service offering.

The Android app is implemented using Google's Material Design guidelines, and provides offline-first support that allows users to write and view letters without a network connection.

The backend server communicates with the Android app with a REST API, and is responsible for storing data in a Postgres database and integrating with Lob, a 3rd party API that allows developers to programmatically send physical mail.

The scraper is also written in Go, and scrapes through a list of all inmates in Connecticut correctional facilities.

This project has challenged me to quickly learn new skills and concepts. Prior to this project, I had no Android development or Go experience. In addition to having to learn a new framework and language from scratch, I was also challenged to understand and implement different architecture and design patterns. This included learning Android patterns such as the Repository, Observer and Singleton patterns, REST API design, relational schema design, authentication protocols such as OAuth 2.0, working with 3rd party APIs, synchronizing data between a local cache and a server, web scraping, and many other areas of software engineering that will be relevant for my future work in industry.

1 Motivation

InTouch is an Android app that allows relatives and loved ones of inmates to easily send physical mail to inmates for free, and the app was built to address our target market's needs by providing a free and easily accessible alternative to the logistical hassle of sending physical mail and expensive, extortionary e-messaging solutions of questionable quality by for-profit companies[1].

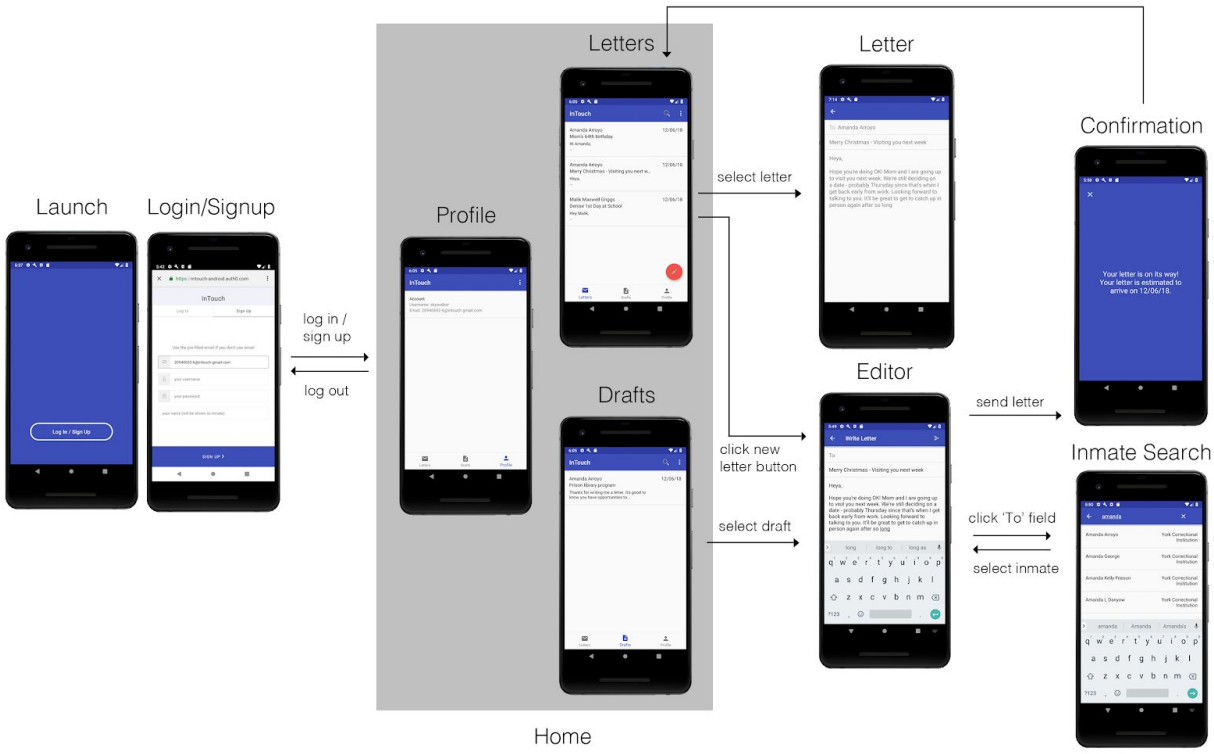
More broadly, the InTouch app can be seen as an effort to provide great, easy-to-use software to underserved communities free of charge. While everyday consumers get to use modern software built by tech companies, e.g Google Maps, Gmail, Facebook, Yelp, Amazon, with the latest features, UI/UX, and performance optimizations, often for free or at a reasonably low cost, there are still many segments of society that lack access to such software.

Inmates and their families are one such community. InTouch Project is a non-profit that went through the TSAI City (formerly the Yale Entrepreneurial Institute) accelerator last year. Their mission is to use technology to help inmates and their loved ones keep in touch, relying on long-standing research that increased contact between inmates and friends/relatives reduces recidivism[2], and the Android app I have developed was built for them.

[1] https://www.wired.com/story/jpay-securus-prison-email-charging-millions/
[2] https://journals.sagepub.com/doi/pdf/10.1177/0887403411429724

2 UX Flowchart

At its core, the InTouch app is comprised of a text editor that lets users write letters to inmates, and a list view that lets users see all the letters and drafts they have written. I will go into a technical discussion of the app's implementation in section 5, but in order for the reader to have a clear sense of the final product, first I present a UX flowchart that illustrates a) all the screens that comprise the app, and b) how the user navigates between those screens.



All of the screenshots were taken on an actual working version of the app. Additionally, an enlarged version of the flowchart[3] and a video demo[4] are also available online.

3 Backend Server

At the core of every application is the data model that it uses (detailed in section 3.1) and the way in which the user can manipulate and analyze the data (detailed in section 4). Additionally, I will also detail the languages & technologies used in building the server.

---

3 https://raw.githubusercontent.com/JohnAmadeo/InTouch-Android/master/UX%20Flowchart.png

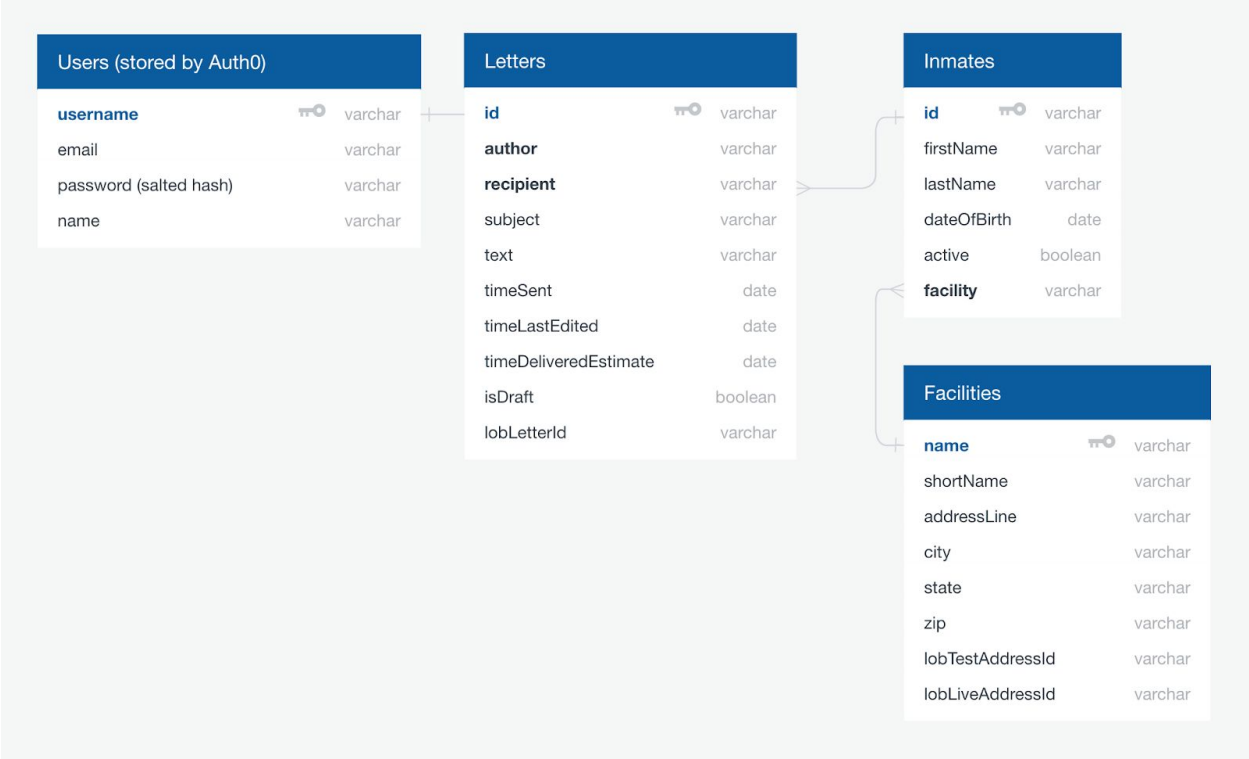
4 https://github.com/JohnAmadeo/InTouch-Android/blob/master/InTouch%201.1.mp4

The server's implementation is available online at https://github.com/johnamadeo/intouchgo

3.1 Database Schema

For this project, I chose to use Postgres, a relational database. Conceptually, the schema contains four tables: *letters*, *inmates*, *facilities*, and *users*. In practice, the *users* table does not exist on the project's hosted Postgres database as I chose to use Auth0, a 3rd party identity management/authentication API, in order to ensure authentication flows were correctly implemented and user data was securely stored. I considered implementing this myself, but decided the time would be better spent focusing on core value-add parts of the app that were unique to InTouch's mission. Consequently, the *users* table is actually hosted on Auth0's server.

The schema diagram below illustrates both the data stored by each table, and the relationships between tables.



While the schema diagram is mostly self-explanatory, there are several additional points to note.

First, I also added length and non-null constraints on various fields to ensure that it would be impossible for the server to accidentally insert malformed data into the database.

Second, the *facilities* table contain fields specific to Lob, the 3rd party API we are using to schedule the delivery of users' letters via USPS physical mail to prisons. When creating a new letter with the Lob API, a request can specify the recipient's address as a Lob address ID, or as

a plain string. If it is passed in as a plain string, Lob will attempt to sanitize the string and verify if the address is a valid US address USPS can deliver to. However, the sanitization is not free and billed to the developer, so I decided to use some of the free API calls I had to create Lob address IDs for each prison in Connecticut and store the ID in the *facilities* table.

3.2 API Design

InTouch's backend server communicates with the Android app via a REST API. A core part of REST APIs are *resources* i.e data models, and *routes/endpoints* i.e URLs at which a client can fetch or manipulate resources.
With that in mind, below is a list of resources that InTouch implements and their associated routes.

Resources

- Letter
    - Models a letter whose recipient is an inmate in CT
- Inmate
    - Models an inmate that is currently or has at some point been in a CT correctional facility
- Facility
    - Models the name, geographical location, and other metadata of a CT correctional facility

Routes

- POST /letter
    - Description:
        - Create a new letter. Stores the letter in the database and sends it to the Lob API to be delivered by USPS.
    - Payload**:**
        - letter: object *required*
            - The Letter object exists as a Java class on Android, and a Go struct on the server
- GET /letters
    - Description:
        - Fetch all letters belonging to a user
    - Params:
        - username: string *required*
            - Username of the user whose letters the client wants to fetch
- GET/ inmates
    - Description:
        - Fetch all inmates whose name matches the search query passed by the client
    - Params
        - query: string required
            - An inmate is said to match the search query if the search query is a substring of the inmate's name

Additionally, each API request to the backend server must also be accompanied by a valid (i.e has the right issuer and is not expired) access token, included in the request as a header using the *bearer authentication* scheme.

There is also a file server exposed at the '/' route of the server that serves static assets (e.g the InTouch logo).

3.3 Languages & Technologies

Language
The server is written entirely in Go. I had not encountered Go prior to this project, and was adopted following the recommendation of my thesis advisor Kyle Jensen. In practice, Go worked very well for this project. The lightweight syntax of the language allowed for quick development (as opposed to a language with a more cumbersome notation like Java), while still providing safety and reliability through static type-checking, robust error handling, and an object-oriented paradigm (although the language eschews classes in favor of structs and interfaces). It also has interesting concurrency primitives but my project did not end up needing them.

Another advantage to Go is that it has a fairly extensive standard library. The Go standard library provided HTTP client and server implementations, as well as a generic interface around SQL databases. This meant that I could keep the number of external packages (often referred to in other language as libraries or modules) I used fairly low, as opposed to other languages I had used in the past such as Javascript or Python where extensive use of third party libraries and frameworks is more common.

3rd Party APIs
My backend server integrated with 2 APIs - Auth0 and Lob. As has been previously discussed, Auth0 is an authentication/identity management API and Lob is a physical mail API. Fortunately, both APIs had good, developer-friendly documentation, but building the actual integrations was still challenging, and gave me the opportunity to learn different concepts (e.g MIME types, PEM certificates, JSON Web Tokens, etc.) and tools such as curl and Postman.

Deployment
The server and the Postgres database are both deployed on Heroku, a Platform-as-a-Service offering. I chose Heroku because it has an easy-to-use CLI app for managing my deployed apps (e.g logging, setting environment variables, etc.), and a generous free tier.

4 Android Application

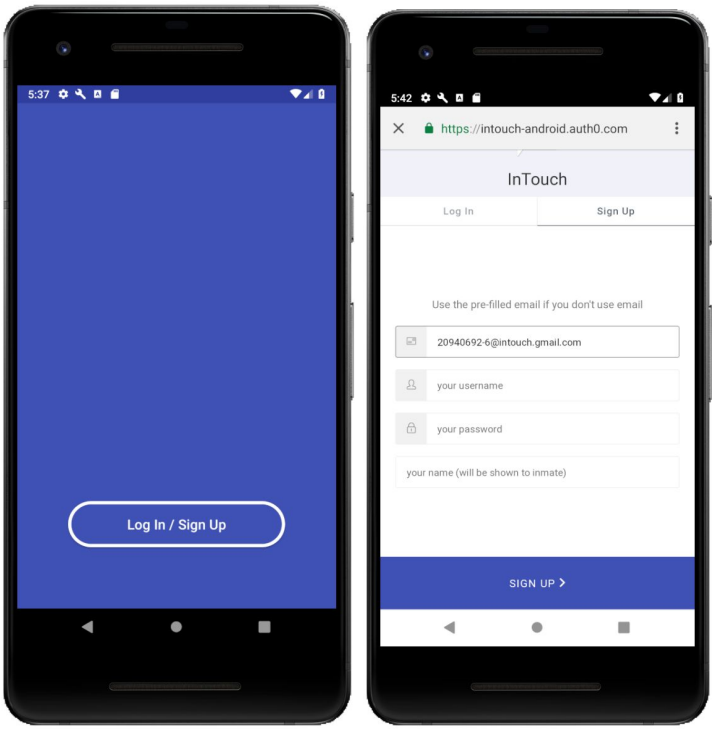
The source code for the Android app is available online at https://github.com/JohnAmadeo/intouch-android.

4.1 UI/UX

In section 2, I presented a flowchart of the Android app's UX. In this section, I will go into more detail about specific UI/UX design choices made in the creation of the app.

Regarding the overall design philosophy, all UI elements (with the exception of the "Log In/Sign Up" button on the launch screen) in the app follow Google's Material Design[5], a cross-platform design system that Google both uses for all of its products, and recommends third party developers to use as well. I chose to use Material Design in order to make the app feel native to the Android platform. Moreover, Android provides several built-in components that use Material Design such as a floating action button and a bottom navigation bar, which also allowed me to speed up the development cycle.

Launch Screen & Login/Signup Screen



The first two screens are very simple. There is a button that triggers the login/signup process, which reroutes the user to the browser and opens Auth0's login/signup portal. Having the entire login/signup flow within the app would have been preferable for the user experience in comparison to opening a browser, but IETF, the Internet standards body, now recommends as a best practice that OAuth 2.0 (the authentication protocol InTouch uses) authentication requests should be made from a browser for security purposes[6].

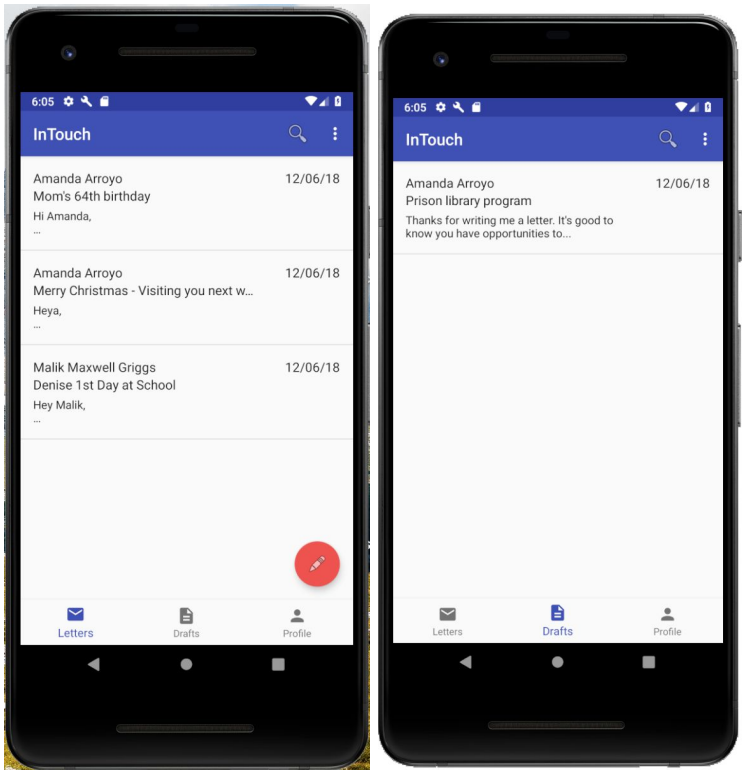
[5] https://material.io/
[6] https://tools.ietf.org/html/rfc8252

On the signup flow, a user is asked to fill in their email address (Auth0 only allows account creation if an email is given as it is their primary account recovery mechanism). My research on how likely low-income users are to have an email address produced inconclusive results, so to account for users who don't use email I pre-filled the email field with a dummy InTouch-related email address and provided text instruction for users to leave the email field pre-filled if they don't use email.

For convenience, if a user is already logged in, the app will skip over these two screens entirely and navigate straight to the list of letters.

In the future, I would like to add an SVG illustration and some text to explain the purpose of the app. Given that the user has already read the app's description in the App Store beforo downloading it, this improvement is not strictly necessary, but it is a common UI pattern.

Letters Screen& Drafts Screen

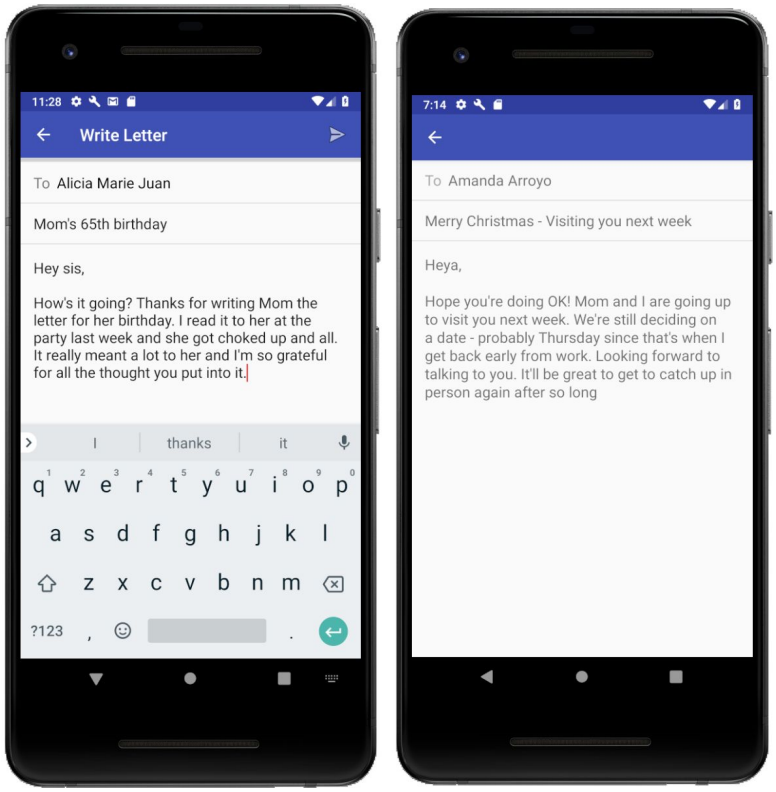


The letters screen shows a list of all letters the user has sent, inspired by Gmail's UI. The search button on the top of the screen expands into a search bar that allows the user to search for a letter by typing in a queryr. Search results dynamically respond to the user as he/she is typing, instead of waiting for a user to finish and press 'Enter'.

The letters screen also serves as a hub for navigating to other screens. Tapping on any of the list items navigates to the letter viewer screen, where a user can seean individual letter. The bottom navigation bar can take the user to the drafts or profile screen, and tapping on the red button navigates the user to the editor screen, where the user can compose a new letter.

The drafts screen shows a list of all drafts the user has create and has a very similar implementation to the letters screen. Drafts are created when a user starts writing a letter but hasn't sent it yet. If a draft is succesfully sent, it will no longer appear on the drafts screen.

Editor Screen & Letter Viewer Screen

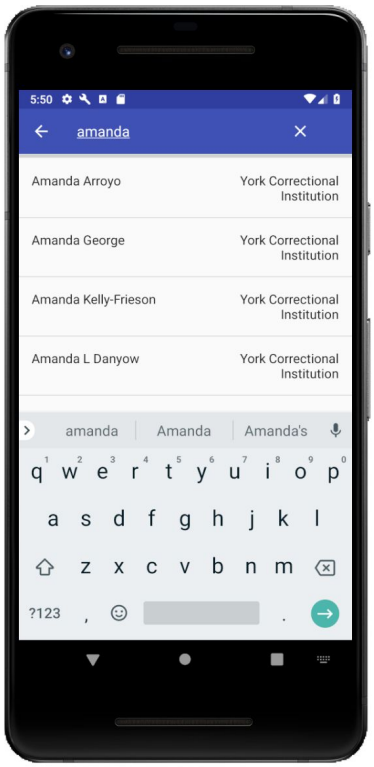


The editor screen lets the user compose a new letter. If the user is navigating to the editor screen by selecting a draft on the drafts screen, the editor will pre-load the UI with the draft's contents. Otherwise, the subject, and text body will show hints of what each of the fields are for.

Tapping the send button on the action bar brings up a modal that asks the user to confirm if they want to the send the letter to the inmate, in order to prevent accidentally sending an in-progress draft and having to rewrite the letter from scratch. Additionally, the send button will only show the modal if the letter can be sent, i.e it has a recipient and the text body is not empty.

If the user confirms their intent to send a letter, the modal will disappear and the progress bar will become animated to indicate the app is trying to complete the process of sending the letter. If for some reason (e.g no network connectivity) the app cannot create a new letter on the backend, then a snackbar[7] (i.e a temporary floating text box) pops up to tell the user that it failed to send the letter and that the letter has been saved as a draft so the user can try to send the letter another time. If the letter is successfully sent, the app navigates to the confirmation screen.
If the user taps the back button, the letter is automatically saved as a draft, accessible from the draft screen. This way, the user does not need to write an entire letter in one go and can resume a draft whenever they have time. To indicate this behavior, a message with the text "Saved draft" is briefly shown on the next screen. Lastly, tapping the 'To' field navigates to the inmate search screen.
The letter viewer screen shows the user a letter that the user has already sent, and is a simplified version of the editor screen.

Inmate Search Screen

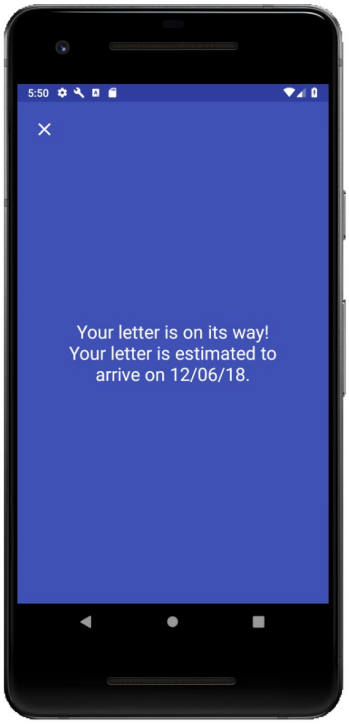


The inmate search screen allows the user to search for inmates. The UX is similar to the search feature on the letters screen, with the exception that the inmate search is debounced[8]. As the inmate search sends an API request to the backend server to find matching inmates, we don't

[7] https://material.io/design/components/snackbars.html
[8] https://css-tricks.com/the-difference-between-throttling-and-debouncing/

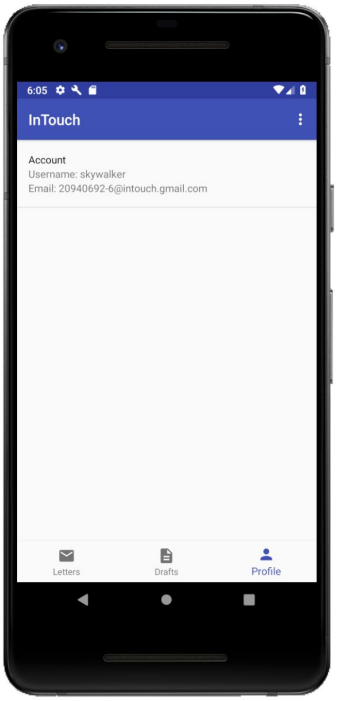
want to fire a new API request for every new keystroke. Rather we wait for some time to pass after the last keystroke, and only then fire an API request, with the assumption that the user is still midway through typing if successive keystrokes occur in rapid succession. The letters screen doesn't do this since they are searching a local cache, and the number of letters a user has is much smaller that all inmates in CT. Tapping an inmate navigates back to the editor.

Confirmation Screen



The confirmation screen is shown after a user successfully sends a letter. In Gmail, sending an email simply navigates back to the home screen. However, I chose to add a confirmation screen on the belief that the UI/UX of an app should do more handholding when the user is performing an unfamiliar or important action. For example, when a user buys a product on Amazon, the user is rerouted to a confirmation page that confirms the product is indeed going to be shipped and that the user's payment went through. In the context of InTouch, users send letters to inmates at a much lower frequency than people send email or SMS, so I decided it was important to reassure the usertheir letter was indeed on their way to the inmate. The estimated delivery date shown is obtained in the response from the backend server.

Profile Screen



The profile screen shows the user's username and email address.

4.2 App Architecture

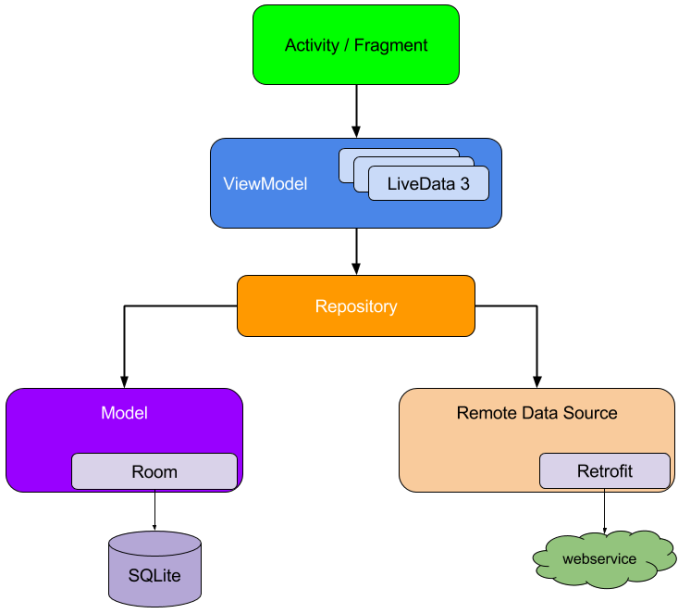
Understanding the best practices for Android app archiecture was one of the most challenging aspects of the project.

Coming from the client-side Javascript web application world, where there is more consensus on best practice, Android has a wealth of different architectures used across industry, such as Model View Controller, Model View Presenter, Model View View-Model, Model View Intent etc.

In addition, the Android framework has changed a lot across the years, so best practice recommendations from Google itself can sometimes be inconsistent.

Lastly, with the recent introduction of Kotlin as an officially supported language for Android development, I had to choose between Java and Kotlin. I chose Java on the hope that there would be more articles/reference material on best practices in Java. This proved only to be somewhat true. For example, one of Google's key sample apps had been rewritten into Kotlin without any corresponding Java version.

In the end, I stuck very closely to the high-level app architecture recommended by Google's most recent Android architecture guide: https://developer.android.com/jetpack/docs/guide.



Fragment & XML
The UI layer in Android consists of Fragment classes (refer to the diagram above) and their corresponding XML files. The XML files are similar to HTML files, in that they merely specify the layout of the UI, while the Fragment job is to handle logic specific to updating the UI in response to user interactions, as well as dispatch those user interactions to the ViewModel. The best practice is to define a single parent Fragment for each app screen

For example, InmateSearchFragment contains logic that updates the list of inmates on the screen depending on the search query typed in by the user. but it doesn't perform the logic involved in finding the list of inmates that match the user's query. Rather, it just dispatches the user's latest search query to the InmateSearchViewModel and receives a new list of inmates.

ViewModel
The ViewModel is the "data container" for a Fragment. Every Fragment (and thus every app screen) has a corresponding VIewModel. The ViewModel contains business logic that is specific to the particular app screen.

Repository
A Repository in Android jargon, is a class that is responsible for establishing the API the app will use for interacting with a particular data model. Consequently, in my app there is a UserRepository, a LettersRepository, and an InmateRepository.

Repositories abstract away the details required for working with a data model so that a ViewModel can just initialize a Repository and use its methods, as well allowing the reuse of common logic across multiple ViewModels. For example, SentLettersViewModel simply calls the

function LettersRepository.getLetters, letting the LettersRepository handle the details of how to synchronize fetching from both the local cache and the backend server.

Webservices and Local Cache
One of the big goals of the app was to provide offline-first support. Since a user writing a physical letter can do so without a network connection, InTouch has to be perform well offline as well!This is done by having a local cache in the form of a SQLite database. Letters and drafts are cached allowing users to view and work on letters offline.

The app also has 2 separate webservices, one that connects to the Auth0 server for authentication, and one for talking to InTouch's backend server.

The last aspect of the architecture worth touching upon is how asynchronous operations are handled. While different languages have different patterns for handling async actions (e.g promises/callbacks in Javascript, goroutines in Go, async/await in C#), the Java community has largely settled on the Observer pattern, where an observer can observe a stream of data across time (e.g in order to implement search debounce, search queries are modeled as a stream of queries that come in as the user types in the search bar). Initially, I used Google's official Observer library, LiveData, but soon discovered it was underpowered compared to RxJava, the Android community's preferred library. As a result, there is a mix of LiveData and RxJava in the codebase. In the future, it would be good practice to refactor uses of LiveData into using RxJava.

5 Scraper

The scraper is written in Go and is available online at https://github.com/JohnAmadeo/intouchgo/tree/master/scraper.

To get an up-to-date list of all inmates in CT correctional facilities, I scraped http://www.ctinmateinfo.state.ct.us/, a publicly accessible database of all inmates in CT.

The CT website did not allow a blank search and forced users to search at least by last name. I ended up filling the search query with a single letter, going from a-z, where each query would return a list of all inmates whose last name started with that letter.

Moreover, submitting the search query rerouted to a new URL, http://www.ctinmateinfo.state.ct.us/resultsupv.asp. However, as you can see the new URL does not have path parameters related to the search query. Many modern web apps will implement the URL of a search page using some variant of https://websitename.com?query=something, which would have allowed us to simply make 26 GET requests to http://www.ctinmateinfo.state.ct.us/q?=a, http://www.ctinmateinfo.state.ct.us/q?=b, …., http://www.ctinmateinfo.state.ct.us/q?=z to obtain all inmates. Instead, I was forced to

programmatically fill in the form, click the submit button, and reroute to the new search results page for each letter. This also meant having to use Chromedriver. I ended up also using the Go library chromedp, which provides a set of convenient APIs to use Chromedriver.

Finally, once I obtained the HTML pages and scraped all inmate data, I updated the database with the new inmate data. Specifically, new inmates are added as new rows to the inmates table, and inmates that are in the table but no longer found on the website are assumed to have been released and marked as inactive.

FInally, deploying the scraper to Heroku required me to change the default app configuration. Specifically, I ended up having to figure out a way to deploy Chromedriver to Heroku as well.

6 Future Work

There are three categories of future work I would like to explore going forward:

Analytics
Knowing how and if users are using an application is crucial to a developer's ability to improve an app. Given this, I plan on integrating tools such as Google Analytics or Mixpanel.

Scalability
Right now, the codebase has very minimal test coverage. I would like to write unit tests and integration tests for both the Android app and the backend server to ensure future changes do not break the functionality of the app. Additionally, adding continuous integration to the project would allow me to automatically check new builds against my existing test suite.

More detailed error logging will also be useful to monitor app performance once it goes live on the App Store.

Product
Adding a mechanism for users to give feedback or ask for help when they get stuck or encounter bugs should be a top priority since it will allow me to get feedback on how to improve the app. Integrating a chat SDK like https://www.zopim.com/product/live-chat-mobile-apps/ is a potential way to do this, although a faster hack could be achieved by including a phone number in the app that navigates the user to the Messages app on the phone.

Another potential feature would be to integrate with the Lob API via webhooks in order to update the user when a letter has been delivered by USPS to the prison, just like how Amazon notifies users when a package has arrived. This will give users more reassurance and confidence into the trustworthiness of the app.

Lastly, adding animated illustration to the app when appropriate, such as on the launch screen and confirmation screen, can make the app more delightful and memorable to users.

7 Acknowledgments

I would like to thank my thesis advisor Kyle Jensen, who has been a fantastic source of encouragement and advice throughout the project. Among other things, I'm thankful for his advice on how to be better at estimating the technical difficulty of different tasks, how to handle remote versus local data, the benefits of developing both front-end and backend components of an app in tandem, and lastly for introducing me to the wonderful world of Go.

I would also like to thank various friends who have encouraged me and spurred me on as I worked on the app this semester.