Public News Droid System

Joseph Long

Department of Computer Science

University of New Brunswick

CS4982 Technical Report

Supervisor: Prabhat K. Mahanti

2022/04/11

Abstract

Social networking has become a hallmark achievement of online computation and networking, serving not only the advancement of networking technologies but also the advancement of human communication. Where existing and popular social network solutions allow for international and borderless communications, the enhancement of local communications is often overshadowed and ignored. Although this had led to the development of advanced networking solutions that allow communication from nearly anywhere on the planet, neglect for local communication systems is apparent, forcing potential neighbors to share the same communication channels as international pen pals. In this report, we propose, implement, and deploy the Droid system, a location based social network that restricts the communications of users to their localities, enabling the acquisition of more personal and relevant information for users. In doing so, a simple RESTful API server is developed, alongside a companion Android client, built using the Jetpack Compose UI toolkit. In this system, the Android client is used by the end-user to make posts, restricting said posts to a certain range originating from their location or to a predefined locality. Other clients are then able to query the server for posts, only being able to retrieve a post if it is within their set range. Being that users might expect such location restrictions to imply a certain level of privacy, we also explore possible bypasses to location restrictions, and how one might implement further checks to prevent such exploits.

Table of Contents

List of Figures

1 Introduction

In 1997, SixDegrees.com was launched by Andrew Weinreich, which is a website that allows users to invite people who they have an existing real-life connection with to the website. By doing this, users would effectively build a graph of other users who they have varying 'degrees' of connections with. SixDegrees.com is widely considered to be the world's first implementation of a social network[1], a concept which would soon be further developed on by many other implementations such as MySpace.com and Facebook by Meta, which boasts 2.9 billion active users as of third quarter 2021[2]. The outstanding success of social networks like Facebook from an economic point of view is implicit, often making use of learned user habits and collected data to enable increasingly personalized advertisements. From a technological point of view, the motivation created by social networks for the advancement of digital technology is also implicit, with social networking services being a hallmark achievement of online computation and networking, serving not only the advancement of networking technologies in general, but also the advancement of human communication. Social networks have enabled and normalized previously impossible forms of communication, such as global instant messaging and crowd sourced information, but while global communications have been greatly enhanced by online technology, there are some forms of human interactions that have been left nearly untouched, such as local and community communications. While these modes of communication can be facilitated through regular social networks, such networks are not purpose built for neighbors or communities, boasting features which favor globally separated pen pals, but which leave much to be desired for local communications.

It is our belief that a social network built specifically for local communications would have a significant amount of room for innovation, as such an idea has been rarely explored in comparison to the many global social networks that already exist. In this report, we will take a closer look at the concept of location based social networks. Furthermore, we will propose, implement, and deploy the

Droid system, a location based social network that restricts the communications of users to their localities, enabling the acquisition of more personal and relevant information for users. In doing so, a simple RESTful API server is developed, alongside a companion Android client, built using the Jetpack Compose UI toolkit.

1.1 Motivation

The use of social networks has helped foster the growth of large communities worldwide, independent of the location of users and the distances between them. This has allowed for large scale collaboration, such as the open-source development communities that can be found in large quantities on websites like GitHub[3], the accumulation of crowd-sourced and open knowledge, such as what can be found on websites like the actively moderated Wikipedia[4], and has even enabled large scale and crowd-sourced philanthropy, such as on the website GoFundMe which often hosts fundraisers for those affected by natural disasters[5]. While the fact that social networks remove barriers like country borders between users is a very positive outcome of their development, the use of social networks to foster physically local communities is not nearly as common as mentioned earlier, despite this application's huge potential. While an individual using social media is now more able than ever to stay knowledgeable on global affairs, politics, and niche internet communities, this flood of information not directly pertinent to a user can act as noise, often drowning out the less plentiful but more relevant information from their local communities.

The aim of the Droid system is to offer a solution to this, by allowing community members to publish information, stories, or other media directly to their local areas, and thus enabling users to receive information likely more relevant to them and their neighbors which might have otherwise never been seen. By leveraging the GPS capabilities of modern smartphones, the system can restrict the information published to those who the information is more likely relevant to, with the Droid companion app acting as a crow-sourced and decentralized analog to the more traditional morning

newspaper. It is our belief that in a few strict scenarios, the Droid system might even be a more powerful tool for local news acquisition, due to its decentralized nature. On the Droid platform, all users except for those with admin roles are equal in that no one user has a more of an ability to be heard than another. This fact alone could help foster independent news in areas where local news is highly controlled and centralized. Other uses of the system might include organizing events, or as location beacons for events to advertise themselves to users who are nearby.

1.2 Goals

The concept of the Droid system is inspired by the following report topic prompt, offered by Professor Prabhat Mahanti.

Droids, less commonly known as robots and automatons, were mechanical beings that possessed artificial intelligence. This is one of the excellent computer science projects for beginners. The public news droid is an informative software application that informs users about the trending news, occurrences, and interesting events happening in and around their locality. Thus, the idea behind creating this information system is to keep the users informed about the happenings in their vicinity. The system uses Android Studio as the frontend and SQL Server as the backend.

The system involves two modules, one for the admin and one for the user. The admin monitors the accuracy and relevancy of news and information. For instance, if the admin encounters fake news or app misuse, they can take necessary action to stop the spread of such irrelevant information. On the contrary, users can view news and informative articles only of their respective localities/towns/cities, and they can add news related to any other city. Mentioning computer science projects can help your resume look much more interesting than others.

To use the app, users need to register into the system to use this app and add all the necessary details. Once the registration process is successful, the user can see the latest news, refresh the app, browse for more information, add new information and upload it (within 450 words), and so on. Users can also add images and title for the news they add.

Given this prompt, a more concise list of goals for the functionality of the Droid system can be created, and is as follows:

1) Develop an app for Android devices using Android Studio
2) Develop a server/SQL backend to enable online connectivity for said app.
3) The app should allow users to (a) login, (b) register, (c) make posts regarding local news/community happenings, and (d) retrieve such posts in a timeline like fashion.

4) The app should be geo-locked. This means that:
    a. Users should only be able to post to their local vicinity, and that
    b. Users should only be able to retrieve posts in their local vicinity
5) It is assumed that rights to modify and delete posts should strictly lay with the original poster (OP), and therefore an authentication solution is required.
6) The above requirement also implies an ability to delete and modify posts.
7) Posts should allow images to be embedded or attached in one way or another.
8) Posts should also have titles, and content (text, within 450 words)
9) An admin module of some form should allow superusers to moderate posts within an admin module to prevent app misuse.
    a. An admin module may be in the form of a website access point.

2 Background

The development of an application and a server, along with the design of the desired behaviors and functionality of a complex social network, is no small task. Fortunately, there exists frameworks, tools, and methodologies that allow a developer to avoid building an app from scratch and a designer to avoid designing while being uninformed of the best practices, both of which are tasks that would be considered “reinventing the wheel”. By making use of said tools, one can speed up development and conceptualization time, and dedicate more resources to the creative aspects of development, enabling the development of more novel programs. In the development of the Droid system, such resources are used extensively. The Droid client is built using Android app APIs and frameworks, the Droid server is built using the increasingly popular REST API structure, and both the client and server are designed with the concepts of location based social networks in mind. In addition, all areas of the Droid system take heavy influence from existing social networks and location-based apps, such as Nextdoor, Yik Yak, and

Tinder. In order for this report to be as comprehensive as possible, in the following sections we will discuss prerequisite knowledge that is believed to be useful to those uninformed of any of the previously mentioned concepts. We will begin by discussing the Android operating system and its app development process, briefly exploring the Jetpack Compose UI framework.

2.1 Android Apps

Android is a Linux based multi-purpose operating system, capable of running on a wide plethora of device types, including car media systems, smart wearables, smart TVs, and predominately smart phones[6], making up 70.94% of the global smartphone market as of February 2022[7]. While Android is a proprietary operating system like Windows, its core components are shared with AOSP, the Android Open-Source Project[8]. This makes the operating system much more transparent in terms of source code, despite not being entirely open source, allowing for a significantly more developer-friendly platform to develop for. With that being said, the components of the operating system have become increasingly more closed-source since Google's acquisition of the operating system in July of 2005, with components such as AOSP search, Android's original open-source search engine client, being replaced with closed-source Google analogs, like the Google Search app[9].

Although Android has been famously known as a Java first platform, meaning that apps for the operating system have been predominately written for the Java runtime environment, so much so that many Android APIs are exclusively available through Java packages, this has recently changed. With claims that using the newer language would result in 20% less required code, Google announced first class support for the Kotlin language at its annual Google I/O conference in 2019, stating that newly developed operating system APIs would be developed for Kotlin first, phasing out the explicit use of Java[10]. Kotlin is an object-oriented language developed by JetBrains and the Kotlin Foundation, boasting many functional improvements over Java, such as increased null safety to combat the infamous null-pointer error, type inference, and according to the Kotlin Foundation, a more concise and expressive

syntax[11]. The transition from Java to Kotlin is made easier by the fact that Kotlin, although it can be compiled to native code, can be trans-compiled to Java bytecode, meaning that the language runs on the JVM (Java virtual machine) and is thus interoperable with Java code. This means that any Java package or class can be used and referenced directly by Kotlin code.

2.1.1 Jetpack Compose

In traditional Android development, a developer must develop not only in their programming language of choice, a selection which is usually restricted to either Java or Kotlin, but also in XML, or the extensible markup language. While the programming language of choice is used to define business logic and update the user interface of the app, XML is used to define the views and fragments of the user interface itself, as elements[12]. This is a language pairing quite like that of web languages, where HTML is used to define the contents of a webpage and a scripting language such as JavaScript is used to define its logic, modifying the HTML elements when needed. The need to switch back and forth between a programming language and a markup language throughout development can be seen as cumbersome, though, which is why JavaScript frameworks like React[13] and Vue JS[14] have been developed for making web applications. These frameworks blur the line between the two languages by enabling declarative and stateful user interface definitions. Just as the need to blur the lines for web languages led to the development of these frameworks, the same can be said for Android development, where the need to define UI components without a separate language led to the development of Jetpack Compose.

Jetpack Compose is a framework developed by Google's Android team, which offers a toolkit for UI development that allows developers to define app user interfaces and components in Kotlin alongside their business logic, replacing the need for XML user interfaces whilst still allowing an appropriate separation between interface definitions and business logic. Compose uses a declarative design philosophy, meaning that as opposed to imperative design, where a programmer must describe how they would like something to be done, a programmer instead simply must describe what they want

done. This preserves the simple nature of markup languages, in which a developer only must describe the user interface, allowing the runtime to decide how to render it, whilst allowing the developer to declare the interface in an imperative language. Jetpack Compose is also stateful, meaning that state changes automatically populate down the composition tree, triggering any UI components dependent on the specific state variable to redraw without the programmer needing to explicitly tell it to. This is the Jetpack Compose state flow, where events flow up to component tree, and state flows down.

2.1.2 MVVM Architecture

A variation on Martin Folwer's 2004 presentation model design, MVVM or model-view-view-model architecture is a design pattern adapted by Microsoft employees to help separate the business and presentation logic of a program from its user interface[15,16]. It can be thought of as a layer of abstraction between the user interface (the view), and the business logic (the model), facilitated by the view-model, allowing for greater decoupling between the two, making them independent of each other and thus improving maintainability and testability. The job of the view-model in this pattern is to both facilitate the view's communication and calls to the model, and to provide data from the model to the view in a form that is easy and intuitive to present to the user. At the same time, higher testability for both the UI and the business logic of a program is achieved by allowing the developer to replace the business logic with stub classes when testing the user interface alone, and vice versa. This design architecture has been further popularized by Android development and is Google's recommended pattern for Android development. The view-model concept has its own class in the Android APIs, "ViewModel"[17].

2.2 APIs

In MVVM architecture, its common that the model portion of the pattern takes the form of an API adapter, or an application programming interface. An API is an interface, or layer of abstraction, built on top of a program that allows other systems, which may or may not share little in common in

terms of design, architecture, or functionality, to call on the original program. APIs allow a program to selectively expose certain aspects or functions of itself to other independent software to call on, whilst also allowing the program to hide or "abstract away" the details of how it accomplishes the requested task[17]. This can be seen as similar to how a developer might call on a built-in library to process an object and return a result. The developer is interested in the input and the output produced, and most likely isn't interested in how it the library accomplishes the task, much like a black box concept. While the most common use case of APIs are for client-server communication in web applications, where a client (such as an Android app) might call on a server using it's exposed API, passing parameters in the form of a URL, possibly a body, and possibly headers, and receiving data returned in the form of a JSON object, the earliest use case for APIs were predominantly for communicating with the host operating system or other applications running on the same system[17]. This use case of APIs is still used heavily in most areas of software development, often taking the form of software libraries such as Java packages or the OpenGL graphics library, or the Win32 library, which is the operating system library for used for calling built-in functionality of the Windows operating system, allowing developers to create windowed applications without needing to know the intricacies of the operating system, among other things. The key takeaway from this is that APIs allow developers to create interfaces that other applications can call on without needing to know how the requested tasks are done, and without needing to occupy the same address space or scope of the exposed program. In the following section, we will briefly explore the history of client-server APIs, touching on standards and patterns such as SOAP APIs, RESTful APIs, and newer forms such as GraphQL.

One of the earliest forms of a client-server API came as an evolution to Microsoft's interprocess communication method known as COM, or component object model, and was called the distributed component object model, or DCOM. DCOM was meant to be the model for communication between networked computers and was hoped to eventually become the standard for process communication

over the Internet. As COM was a binary interface, so was DCOM, and although this was somewhat mitigated by use of serialization and deserialization between hosts, it proved to be difficult to use a binary interface, and so while HTTP requests became the standard instead, Microsoft would replace DCOM as an attempted Internet standard with its SOAP protocol[19]. SOAP stands for Simple Object Access Protocol, and is a protocol maintained by the XML Protocol Working Group. The most notable changes between DCOM and SOAP was the use of XML as a messaging format as opposed to binary messaging, and the ability for SOAP to work over HTTP, although it is not restricted to this[20].

In recent years, the use of SOAP APIs has fallen somewhat out of fashion in favor of RESTful applications. As opposed to SOAP, REST or "representational state transfer" is not a standard or protocol but instead is a design pattern that imposes a set of constraints on the design of web applications, and thus an application is RESTful if it adheres to its set of constraints. RESTful applications are built on top of HTTP or HTTPS and uses HTTP methods such as GET and POST to access and modify resources on servers, returning data in either XML, JSON, or HTML. There are 6 constraints that make a service RESTful[21], and they are as follows: 1) REST architecture has three unique components, clients, servers, and resources, and thus REST follows client-server architecture. 2) Requests are stateless, meaning that no state about a client or its session is stored on the server, and instead the client manages its own state. 3) Content should be cacheable, to minimize the amount of communication required between the clients and servers. 4) RESTful services can have additional layers to extend functionality, such as a security layer. 5) The server can extend the clients with code on demand, meaning that executable code can be sent to clients by servers given proper security. 6) RESTful APIs should feature a uniform, and thus predictable and understandable, interface.

2.3 Location Based Social Networks

In this report we're detailing the design and implementation of a location based social network, which is a social network that leverages the ever-increasing availability and accessibility of GPS

technology, which is integrated into most modern smartphones through the use of satellite and cellular positioning and is even made available to many non-GPS enabled devices by use of various WiFi. Bluetooth, and IP based techniques[22]. This allows apps to access both the fine-grain and coarse locations of users, which can be used to enable previously unknown functionality. In the Droid system, it is used to tag posts made by users with the location the user was at when they submitted the post, stored as coordinates, which in combination the user's coordinates at the time of querying posts from the server, can be used to restrict access to posts based on an arbitrary range, or a community ID. It is through this behavior that the Droid system is able to create a community focused social network, keeping irrelevant information from other communities hidden so that posts from within a community can be front and center to the user. This is far from the only use of GPS information in the context of a social network, and so it is worth exploring some of the many other applications of the concept of location based social networks. At the same time, it is also important to do this due to the fact that the design of the Droid system takes significant inspiration from many other location based social networks. Such projects include Nextdoor, Tinder, and Yik Yak, all of which have received quite a lot of attention due to their unique application of range/community restricted communication. In this section, we will briefly explore the history and behavior of these applications.

2.2.1 Tinder

Tinder is an online dating application available on IOS, Android, and the Tinder website. Founded by Sean Rad in 2012, Tinder leverages the availability of GPS technology to the platforms it supports to connect users of the dating app to those nearby[23]. Although Tinder is closed source, it can be assumed by the functionality and behaviors of the app that the service retrieves GPS coordinates or some other form of an approximate location measurement from its frontend client and compares the location against the recently logged locations of other users in combination with the platform's other considerations and filters in its matching algorithm. Assuming GPS coordinates are used in most cases,

the coordinates can be compared with other coordinates using the haversine method, detailed in section 3.2.1, or some other functionally equivalent algorithm to get a distance between the users, which can be compared to the range filters set by each user to determine if the user in question is within the range of another user before serving the potential match to the user's client. A similar approach is used by the Droid system when considering which posts should be served to a user given both the post's location and the user's location.

2.2.2 Nextdoor

Nextdoor is another location based social network which is used to connect nearby neighbors within a physical neighborhood. Founded by Nirav Tolia, Prakash Janakiraman, David Wiesen and Sarah Leary in 2008, the network's surface level motivations are concisely stated on the landing page of the company's user facing website[24] as follows:

It's where communities come together to greet newcomers, exchange recommendations, and read the latest local news. Where neighbours support local businesses and get updates from public services. Where neighbours borrow tools and sell couches. It's how to get the most out of everything nearby.

Nextdoor differs slightly from previously mentioned location based social networks, as it doesn't rely solely on location technologies to assign users to their respective neighborhood groups, opting to instead rely on users to report to the service their street address and postal code to better support verification of user authenticity. This assists the platform in guaranteeing their claim of Nextdoor being "A secure environment where all neighbours are verified", as the platform's primary method of address verification checks to make sure that the billing address of a user's mobile phone matches their reported address.

2.2.3 Yik Yak

Yik Yak is a location based social network, currently only available on the IOS platform with the promise of an Android release in the near future and is the most similar in functionality and concept of

the three apps we've discussed to the Droid system being detailed in this report. Yik Yak was co-founded by Tyler Droll and Brooks Buffington in 2013 and underwent significant user base growth in its earlier years, followed by controversy regarding the platforms inability to manage the cyberbullying problems taking place on the platform, eventually ceasing operations in 2017. In 2021, however, the service announced that it would be returning and has since returned to the IOS platform[26]. In terms of functionality, Yik Yak is an anonymous only platform that allows users to submit and reply to posts, with the restriction that posts are only visible to users within a 5-mile (8 km) radius of the post's geographic origin[25]. The purpose of both the anonymity and visibility restrictions are to enable a sense of a private community, whilst preserving the privacy of users. This implementation is very similar to the Droid system in that it shares the same visibility restriction concept, tagging posts with their geographic origin and restricting visibility to posts to within a radius from their origin, with a key difference being that Yik Yak uses a static 8km radius for all posts, whilst the Droid system allows posts to set their own privacy radius within a range of acceptable values. Another key difference between the two systems is that the Droid system uses two radiuses to restrict visibility, a privacy radius controlled by the user who created the post, and a filter radius controlled by users querying for posts. This slightly more complex visibility system is detailed further in section 3.2.1.

3 Methodology

The details regarding the design and development and the Droid system will be split into two pivotal sections making up the majority of this report. This section is the first of the two, which will cover the methodology used in designing and developing the system, and will be followed by the section 4, Implementation and Discussion, which will detail the final design and its implementation of the finished server and client. This section is further split into two subsections, the first of which will discuss the tools and resources used in building the Droid system, and the second discussing the design of the

intended behaviors of the system and the considerations which have informed said decisions. Before moving ahead to these sections, it may be worthwhile to remind the reader of the intended functions of the system. The system as a whole can be divided into two independent components, the server and its corresponding Android client. The goal of the system is to allow users of the Android app client to submit and view geo-locked posts, which are posts which can only be viewed by those who the system considers sufficiently nearby to the location which the post was originally submitted from. The way the system, or more accurately the server, makes such a decision will be discussed in greater detail in sections 3.2.1 and 3.2.2, but without discussing implementation details can be viewed as a set of constraints, created in part by the original poster and in other part by the user who is querying the server for the post, of which the querying user must satisfy in order to be granted access. While there are conditions that must be met to view the post which do not involve the user's location, the majority of these constraints are designed to ensure that the user is sufficiently geographically near the post. Now that we understand the intended core functionality of the system, we will move on to discuss the methodology used to create such a system.

3.1 Tools and Resources Used

3.1.1 Programming Environments

In the development of the Droid system two programming environments, or IDEs, were used. First, the VS Code programming environment in combination with the Python interpreter was used to develop the Python server. VS Code is a free and cross-platform code editor and allows the use of plugins to extend the program's functionality and save time in development.[27] An assortment of python-related plugins was used in the development of the server. The second programming environment used was the Android Studio IDE and was used to develop the Android app client[28]. Android Studio allows for both Java and Kotlin development and supports both the XML template language UI development style and the Jetpack Compose development style, both of were previously discussed in detail in section 2.1.

In this project, the Kotlin language was chosen as well as the Jetpack Compose UI development style. These decisions were made in part due to their supposed ability to speed up development time, and in other part due to prior experience with these technologies.

3.1.2 Front End Plugins and Packages

The development of the front-end client involved the use of many plugins, all to significantly varying extents. Rather than detailing every plugin used, many of which are used for small trivial purposes, we will instead discuss four plugins/packages which were integral to the development of the Android app. These four plugins are the Jetpack Compose UI framework, Retrofit[29], OkHTTP[30], and Gson[31] .

Jetpack Compose is a framework developed by Google's Android team, which offers a toolkit for UI development that allows developers to define user interfaces and their related components in the Kotlin language rather than using the XML templating language. This allows developers to use only one language for the majority of Android app development, with the only outliers being manifest files and build scripts. Jetpack Compose is discussed in greater detail in section 2.1.1. Following the MVVM architecture design pattern discussed in section 2.1.2, the back end of the Android client, or the "model", is developed using the other three notable plugins, Retrofit, OkHTTP, and Gson. OkHTTP is a Java package developed by Block, Inc which acts a HTTP and HTTPS 2 client, allowing developers who use it to make both synchronous and asynchronous HTTP requests. Similarly, Retrofit is another Block, Inc developed Java package which allows developers to define HTTP APIs as Java interface classes as an abstraction of the REST API that they wish to communicate with. This package, which integrates with OkHTTP, allows for API calls which to be executed as simple Java function calls, with function parameters acting as an analog to HTTP request parameters such as headers, form data, and query parameters. Lastly, Gson is a Java package developed by Google that allows for serialization and deserialization of JSON objects received from API calls to Java/Kotlin data classes. This allows developers

to simply convert received data into data classes which model the JSON responses and bodies of REST API calls.

3.1.3 Back End Plugins and Packages

The server was developed in Python, using built in Python packages, the Django framework, and Django framework plugins such as the Django REST framework and the Django REST authentication plugin. Django is self-described as a "high-level Python web framework that encourages rapid development and clean, pragmatic design"[32], and offers developers a framework to develop web servers or API servers without having to handle URL routing, low-level security, or other common web development hurdles. Django also offers a database-agnostic way to define database models directly in Python in an object-oriented fashion without the need to use SQL to define one's database, which was used heavily in the development of the Droid server. The Django REST framework extends Django to be useful as a REST API framework[33], making API definition easier and adding functionality such as serializers, which is similar to the Gson Java package used the development of the Droid Android app in that it too enables serialization and deserialization of JSON objects. Lastly, the Django REST authentication plugin simplifies authentication of user requests and identification of users[34].

3.1.4 Heroku

Heroku is a web platform that allows developers to securely host server software, such as the Droid system, within their servers[35]. In doing so, it saves developers server costs, machine maintenance, and the need to obtain a TLS certificate, which is needed to enable encrypted HTTPS connections between users and the server. The Droid server is hosted on the Heroku platform and is running at all times. The URL for the API server is https://jlong-droid.herokuapp.com/.

3.1.5 Nightingale REST Client

The Nightingale REST Client is a REST API client which, among other things, allows one to create, store, and execute requests to REST API servers[36]. This client was used extensively throughout the development of the Droid system, especially prior to the development of the Droid Android client, as the server was developed first. This REST client is especially useful due to the fact that it allows developers to create API requests to servers without the need for a fully functional client having yet been developed. This makes it possible to test and debug a server without its corresponding client, helping maintain a separation of concerns when testing and developing.

3.2 Design

3.2.1 Location Processing

The key functionality of the Droid system is its ability for it to restrict post visibility to those who are sufficiently nearby. In order to accomplish this, two competing approaches are considered. The first approach involves setting up a radius surrounding the origin of the post as a function of its origin coordinates and a user supplied range. With these two values, a circle surrounding a post can be formed, and if a user that is querying this post is geographically within this circle, they are permitted to view the post. Another approach involves consuming a post's coordinates upon creation and replacing them with a local community identifier, representing a community level division called a locality, such as "Rothesay" or "Saint John". Upon querying a post using this approach, a user would send their coordinates along with the query, as they would do in the first approach, but their coordinates would again be consumed and replaced with a local community identifier representing the community they're currently within. After this, the community identifiers of both the post and user can be compared, and if they match the user may be permitted to view the post.

Both approaches have their own advantages and disadvantages. The first approach, which sets up a radius around a post, allows for fine grain control of who can view a post, allowing the user who

submits the post to set up as small of a radius as they like. This enables a way for users to maintain a sense of privacy. The second approach, however, makes a post viewable to an entire community, which is assumed to be significantly larger than the most common range value used in the first approach. While this lessens privacy, it benefits discoverability. Additionally, it provides a more human readable format of a post's location than the first approaches combination of coordinates and a range. Rather than choosing just one of these approaches, we instead have implemented both systems in combination, with each complimenting the other. In this combination system, all posts receive both coordinates and local community identifiers, and posts may be filtered by both methods. In this section, we'll detail the component of this system concerning the methods of the first approach, with details regarding the second approach being found in section 3.2.2.

When a post is submitted to the server (most likely from the Android client), the user's current coordinates are sent along side with it, along with a user set range, which describes how far out another user may be from the post's origin point while still being able to view the post. This range and the coordinates are attached to the post by the server and are immutable. When a user sends a request to the server requesting the latest posts that are nearby, the client also sends its current coordinates. The server then lazily performs the haversine algorithm on all posts in question (usually the most recent posts), which takes the coordinates of the post, compares them to the coordinates of the querying user, and returns a distance between the post's origin and the users current coordinates in meters. This distance can then be compared the posts range value that was provided when the post was created to determine if the post should be visible to the user.

3.2.2 Localities

When a user submits a post to the server, they are also given the choice to allow any users within the same local community, or locality (such as "Saint John"), to be able to view the post regardless of whether or not they're within range of the post. This choice will be referred to as the extend to locality

Boolean. If this is enabled, local community identifiers are generated from the coordinates or querying users and are compared to the local community identifier of the post to determine if the post should be visible. This is only done if the querying users themselves enable locality filtering rather than range filtering (which will be discussed further in 3.2.3). If this type of filtering is not enabled by the querying user, the posts range and coordinates are used with the haversine method instead, just as is done in the first location processing approach.

Local community identifiers are generated using the Google reverse geocoding API. When an identifier must be generated, the coordinates in question are sent along a request to the Google API server along with a Google API key and a flag set to only request localities (as apposed to country names, full addresses, or other data which the Google API is capable of generating). Returned from this is a unique identifier called a place ID (e.g. ChIJT1xXvKyUpkwRZ8STb135YZQ), and the human readable name associated with that identifier (e.g. Gagetown).

3.2.3 Query Scenarios

There are a total of 7 different scenarios than can be encountered when a user queries a single post. These scenarios are a function of the post's coordinates, the post's privacy range value, the post's extend to locality Boolean, and a fourth variable not yet mentioned, the user's filter range. The user filter range is used to further restrict results to a circle drawn around the user's coordinates similar to how a circle is drawn around the post's coordinates with the privacy range and is controlled by the querying user. These 7 scenarios can be seen in figure 1.

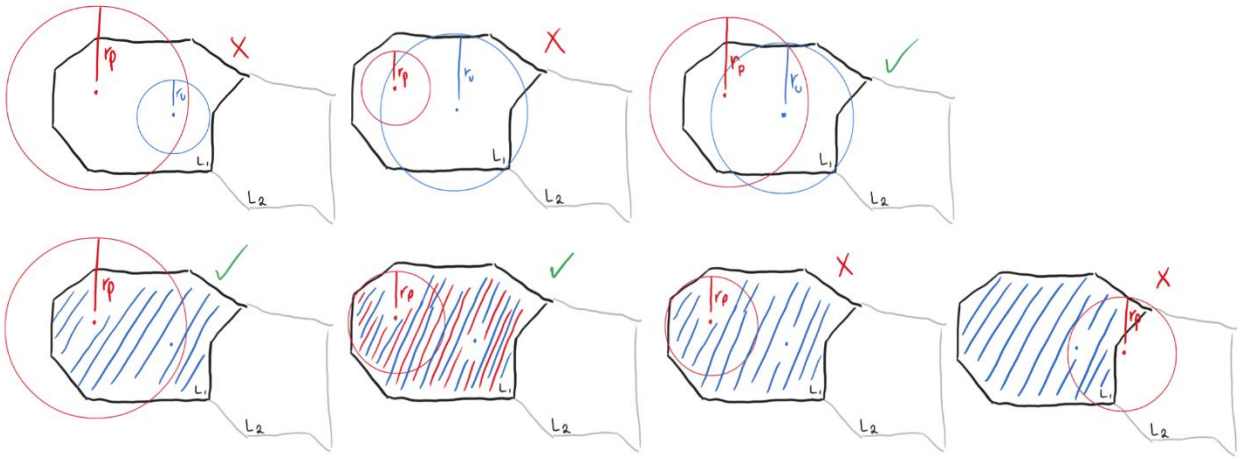

Figure 1 - The 7 different query scenarios, starting with scenario 1 (top-left) and ending with scenario 7 (bottom-right). r_p is the post privacy radius, and r_u is the user filter radius. L_1 and L_2 are different locality regions. A locality region shaded blue means that the user is filtering by locality, and a region shaded red means that the post has the extend to locality Boolean enabled.

In the first scenario in figure 1, the post is outside of the users filter radius, so it is not visible, and in the second scenario the user is outside of the post radius, and so the post is still not visible. The third scenario is the only coordinate and range only scenario that succeeds, as without localities, both the post privacy range and user filter range must be satisfied. In scenario four the post is in the user locality and the user is in the post range, making the post visible even though the post does not extend to locality. In scenario five, the user is not in the post range, but since the post is in the user locality and the post extends to locality, it remains visible. Scenario 6 fails due to the user not being in the post range and the post not extending to locality. Scenario 7 is the most interesting as it is the only locality scenario that lessens the number of posts returned as apposed to increasing it compared to the coordinate and range method. This happens even though the user is in the post range since the user filters on locality, ignoring the post as it is in locality 2.

3.2.4 Image Embedding

As mentioned in section 1.2, which details the requirements for the functionality of the Droid system, the system must offer users a way to embed images into posts. Conflicting with this requirement is the

fact that the Droid system is hosted on the Heroku platform, which is detailed in section 3.1.4, using the free base-level database plan, which tightly restricts the storage space allocated to the database. This conflicts with the image embedding requirement as images require significantly more storage space than plain text, and it is believed that simply storing images in the same database that is used by the rest of the server it would quickly outgrow its allotted storage capacity. To combat this, it was decided that the Droid server would use the Imgur image platform as a secondary database for images, via API calls made from the Droid server to Imgur servers. Imgur is an image hosting platform which offers a public and free API that allows developers to post, view, modify, and delete images hosted on the Imgur platform[37]. The Imgur is used in the image embedding system as follows:

- When a user submits a post with an image, the Android client sends the image to the Droid server encoded in base64.
- When the server receives the post request, it passes on the base64 along with an API key encoded in a new request to the Imgur API server to create an image.
- The Imgur system creates the image, hosted on its server, and returns a JSON object with information about the new image, including the URL that can be used to access the image, and a delete hash which is required to delete the image in the future.
- The image URL and its hash is stored in the Droid server database and is attached to the newly created post.

3.2.5 Authentication and Accounts

To ensure the correct users are given the correct permissions to make posts, edit their own posts, and delete their own posts, an authentication method must be used when such requests are made. Two added requirements that such an authentication method must meet is firstly the requirement that it must be computationally cheap, as it must be performed for most API requests, and secondly it must be

stateless, as the REST API used by the Droid system is stateless as well. The method chosen is the use of authentication tokens. When a user logs into the server though a request to the login API, they are passed an authentication token in the JSON response. This token must be passed in the header of any request sent to the server that requires identity verification, such as posting, editing, and deleting. When the server receives such a request, the token is checked against the tokens table in the server database, which then provides the class handling the request with the user object associated with the token.

3.2.6 Temporary Posts

Services like Snapchat provide users a way to make temporary public posts, which automatically get deleted or hidden from users after a certain amount of time has passed, through a feature called stories[38]. It is believed that a similar feature would be useful to users of the Droid system, for example in the case of an event, where a user could create a post advertising the event that automatically disappears once the event is over. This could also be seen as a privacy feature, allowing users to add an expiration date to sensitive information, ensuring that the information will not be permanently available. In the droid system, this is done by giving users an additional option to add an expiration date to posts when posting, after which such a post will no longer be visible to users querying for posts, even if they're within range of the post.

4 Implementation and discussion

In this section of the report, we will discuss the more fine-grain details of the implementation of both the Droid server and Droid Android client. As we have already discussed in great detail the basics of both Android and REST API development in previous sections, much of the code written in the development of the server and client will not be directly discussed, apart from that which is essential to the unique functionality of the system. The complete source code of both the server and client will be

made available as supplementary materials to this report, and discussion of said materials will be limited to a more abstract overview.

4.1 Server

4.1.1 Database Model Design

The database of the Droid server handles storage of users, user tokens, posts, image references, and precalculated localities. As discussed in section 3.1.3, the Django framework used in the development of the API server allows for the use of Python classes as models of the database, and so the models of the database tables used in the Droid system are defined in Python.

The following is the definition of the Post model, which models the posts made by users.

```
class Post(models.Model):
    title = models.CharField(max_length=6000, blank=False)
    content = models.TextField(blank=False)
    ref_url = models.URLField(blank=True)
    author = models.ForeignKey(User, on_delete=models.CASCADE, blank=False)
    created_on = models.DateTimeField(auto_now_add=True, blank=False)
    coord_longitude = models.FloatField(blank = False, default = 0)
    coord_latitude = models.FloatField(blank = False, default = 0)
    range_meters = models.PositiveBigIntegerField(blank = False, default = 5000, validators =
[MinValueValidator(10), MaxValueValidator(10000)])
    locality = models.ForeignKey(Locality, on_delete=models.CASCADE, blank=True, null=True)
    extend_to_locality = models.BooleanField(blank = False, default=False)
    image = models.ForeignKey(Image, on_delete=models.SET_NULL, blank=True, null=True)
    end_time = models.DateTimeField(blank = True, null = True)
```

Each line after the class signature defines a field or column of the database table, and in this model definition are mostly self explanatory, for example the title field on the second line represents the title of the post.

The following is the definition of the image model, designed to support embedding images into posts.

```
class Image(models.Model):
    id = models.CharField(max_length=10, primary_key=True, blank=False)
```

```
    url = models.URLField(blank=False, null=False)
    delete_hash = models.CharField(max_length=15, blank=False, null=False)
```

The fields of this post are slightly less self explanatory compared to the post model, and so they will be explained. The field “id” represents the unique image hash returned from the request to the Imgur API for uploading an image. The URL field represents the URL at which the image is hosted, which the Android client requires to display the image. The “delete_hash” field represents the hash required to delete the image from the Imgur server upon post deletion.

The following is the Python definition for the Locality model, which represents the human readable local community identifiers, previously discussed in section 3.2.2.

```
class Locality(models.Model):
    name = models.CharField(blank=False, max_length=400)
    google_place_id = models.CharField(blank=False, primary_key=True, max_length=400)
```

Other models used by the server include the user model and the token model, which are built into the various Django framework packages used by the server.

4.1.2 REST API Design

The server API, which is fully implemented in Python code, is made up of two separate APIs. Those are the user-focused APIs, which concern authentication and registration, and post-focused APIs, which concern the APIs involved in creating and querying for posts, among other post-related functionalities. All of these API endpoints will be briefly described in the two sub sections below, and a more complete documentation of the APIs can be found in Appendix A.

4.1.2.1 User APIs

The following is a list of the user-focused APIs implemented in the Droid server.

- POST /api/login/ - a request to this endpoint including one’s username and password returns the API token required for authentication.

- POST /api/register/ - a request to this endpoint including one's username and password creates a corresponding account on the server.
- POST /api/logout/ - a request to this endpoint including an authentication token logs out the tokens corresponding user, effectively deleting the token from the server.
- GET /api/test-auth/ - a request to this endpoint including an authentication token will succeed if the token provided is valid, allowing the client to know if its stored token is still valid.
- GET /api/self/ - a request to this endpoint including an authentication token returns all stored user information, including the user's current locality if the request also contains the user's current coordinates.
- GET /api/self/posts/ - a request to this endpoint including an authentication token returns all posts created by the token's corresponding user.

4.1.2.2 Post APIs

The following is a list of the post-related APIs implemented in the Droid server.

- POST /api/post/ - a request to this endpoint including an authentication token and all required post model fields including user coordinates creates a post on the server.
- GET /api/post - a request to this endpoint including an authentication token and location data will return a list of posts nearby to the user, sorted by newest posts first. This endpoint supports both coordinate-range filtering and locality filtering. This endpoint also supports pagination, allowing the client to request only a few posts at a time rather than all valid and visible posts on the server.
- GET /api/post/{id: Int}/ - a request to this endpoint including an authentication token and location data will return the post with the unique identifier provided in the id field if the user is in range of the post or is the owner of the post.

- DELETE /api/post/{id: Int}/ - a request to this endpoint including an authentication token that matches the user that created the post with the unique identifier provided in the id field deletes that post, and any image embedded in the post.
- PATCH /api/post/{id: Int}/ - a request to this endpoint including an authentication token that matches the user that created the post with the unique identifier provided in the id field edits any valid fields of the post included in the body of the request.

4.1.3 Post Filtering Algorithm

When a user sends a request for nearby posts to the server, the server must first determine which posts are within range of the user. After that, it must also determine if the post is expired or not if the post has an expiration date. A high level-overview of the process used to determine if a post is in range of the user can be found in sections 3.2.1 and 3.2.2, and the concept of temporary posts is described in section 3.2.6. The following is a more in-depth look at how post filtering is accomplished.

The request for nearby posts can take one of two forms, either locality mode or coordinate-range mode. As both modes require the user's coordinates, the mode chosen by the user is determined by whether a filter range is included in the request. If it is, then the coordinate-range algorithm is used, and if not, the locality mode algorithm is used. We will first discuss the coordinate-range algorithm.

1. Firstly, posts in the database are temporarily annotated with their distance from the user. This is done by calculating the haversine distance between the coordinates of the user and the coordinates of each post.

```
posts = Post.objects.annotate(
        distance = haversine(F('coord_longitude'), F('coord_latitude'),
        location['coord_longitude'], location['coord_latitude'])
)
```

2. Then posts are filtered by whether their distance from the user is less than the user's provided filter range.

```
posts = Post.objects.annotate(
        distance = haversine(F('coord_longitude'), F('coord_latitude'),
        location['coord_longitude'], location['coord_latitude'])
).filter(
```

```
        distance__lte=location['range_meters']
)
```

3. Posts are then filter by whether their distance from the user is less than the post's privacy range.

```
posts = Post.objects.annotate(
        distance = haversine(F('coord_longitude'), F('coord_latitude'),
        location['coord_longitude'], location['coord_latitude'])
).filter(
        distance__lte=location['range_meters']
).filter(
        distance__lte=F('range_meters')
)
```

4. Finally, posts are filtered by whether their expiration date is after the current server time. If we were to only filter on this, though, we would lose all posts without an expiration date, so we join the results from this filter with posts that have no expiration date. After this, we order the posts by newest, and paginate the results.

```
posts_active = (
        posts.filter(end_time__gt = datetime.now(timezone.utc)) |
        posts.filter(end_time = None)
).distinct().order_by('-created_on') [page*limit:page*limit+limit]
```

The posts in the "posts_active" object are then sent to the user. Now that we understand the coordinate-range algorithm, we'll now move to the locality mode algorithm.

1. Firstly, we create an object holding all posts with a matching locality identifier that have the extend to locality Boolean enabled.

```
posts_extend = Post.objects.filter(
        extend_to_locality = True, locality__google_place_id = location['locality']
)
```

2. We then create another object holding posts without the extend to locality Boolean enabled, filtered by whether the user is within the post's privacy range and whether the post has a matching locality identifier.

```
posts_no_extend = Post.objects.annotate(
        distance = haversine(F('coord_longitude'), F('coord_latitude'),
        location['coord_longitude'], location['coord_latitude'])
).filter(
        extend_to_locality = False,
        locality__google_place_id = location['locality'],
        distance__lte = F('range_meters')
)
```

3. We then join these two objects.

```
combined_results = (posts_extend | posts_no_extend).distinct()
```

4. We then repeat the same process found in step 4 of the coordinate-range algorithm, filtering for expiration dates, sorting the posts, and paginating the results.

4.1.4 Admin Model

The admin module is fully implemented on the server and can be accessed as a web GUI at https://jlong-droid.herokuapp.com/admin. On this web portal, users with admin privileges (which can be added and removed by the super user through either the backend terminal or the web portal. Once logged in, posts, users, and other models defined in the Droid server can be viewed, modified, deleted, and created. Much of the admin module is built into the Django web framework used by the Droid system, with a few lines of code in admin.py tailoring the module to the server. The following is the contents of the admin.py file.

```
from django.contrib import admin
from .models import Image, Locality, Post
# Register your models here.
admin.site.register(Locality)
admin.site.register(Post)
admin.site.register(Image)
```

The last three lines are of interest, which registers the custom database models to be accessible via the admin module. The user interface of the admin web portal can be seen in figure 2.

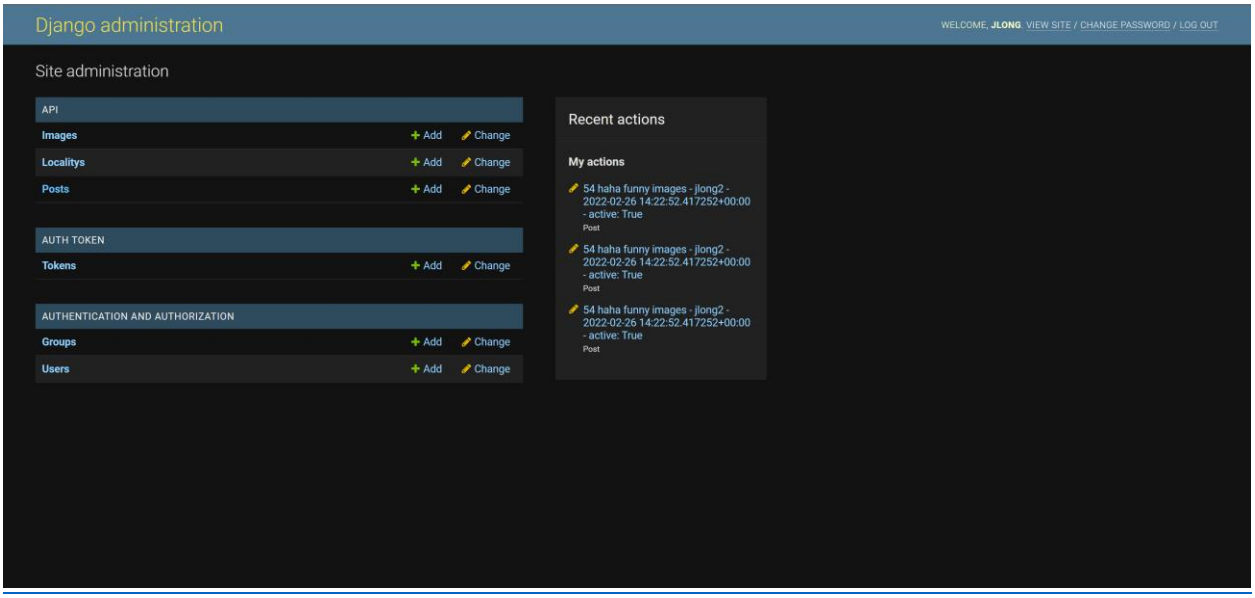

Figure 2 - Admin module UI.

4.2 Client

4.2.1 Communicating with the Server

As previously discussed in section 3.1.2, the Android client of the Droid system makes extensive use of the OkHTTP and Retrofit libraries, developed by Block, Inc., and the Gson library, developed by Google. To communicate with the server, the API endpoints defined on the server are matched with corresponding API interfaces on the Android client. These interfaces are implemented using the Retrofit API in such a way that allows for the JSON bodies of the requests made by the app and the JSON responses received from said requests to be automatically serialized and deserialized to Kotlin data classes, through use of the Gson library. Here we will look at how an endpoint interface such as the login API is implemented in the Android client.

The specification for the login API as implemented on the server can be described as follows:

POST /api/login/

Body (JSON Fields):
username:
required, String
password:

required, String

Successful Result:

```
{
  "key": "d5ec9fac767cc70d87f40ce43855266576eab0f0"
}
```

This can be read as a POST request to the server endpoint /api/login/ which includes a JSON body with a username and password will return an API token if successful. The corresponding app interface for this endpoint is as follows:

```
@POST("api/login/")
suspend fun login(@Body loginDto: LoginRegisterDto) : Response<TokenDto>
```

This interface represents a function call that will send a request to the login API, which takes a data class representing the body of the request and returns a data class representing the response from the request. The definition for both data classes can be seen below.

```
data class LoginRegisterDto(
    @SerializedName("username") val username: String,
    @SerializedName("password") val password: String
)

data class TokenDto(
    @SerializedName("key") val accessTokenVerify: String
)
```

4.2.2 User Interface

Developed using the Jetpack Compose UI framework, the user interface of the Droid app was designed in such a way as to provide users with all important post information, including a detailed look at where the post originates from. The user interface was developed in accordance with the Material Design version 3 specifications, also known as the Material You design language. One notable feature of the Material Design 3 specifications and APIs is its inclusion of the dynamic color palette, which when

implemented in one's user interface, derives its colors from the user's current phone wallpaper. The following figures are screenshots of the Droid Android client user interface.

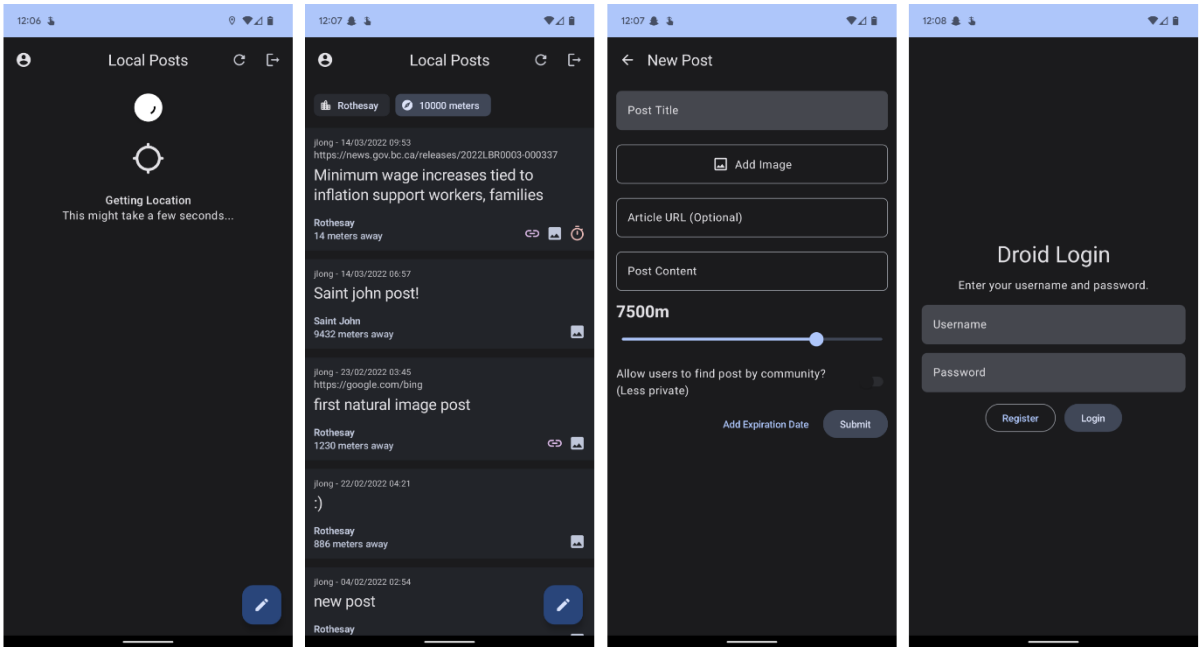

Figure 3 - Screenshots of the Android client user interface.

In figure 1, there are 4 screenshots of the Droid client UI. Image 1 of this figure (left most) is the landing page that the user sees once they've opened the app. This screen tells the user that it is currently fetching the clients GPS coordinates. Image 2 shows the landing page once it's loaded, showing nearby posts and options to change the user's filters (locality mode toggle, and range button). Image 3 shows the new post screen, where users can edit and create new posts to submit to their location. Image 4 (right most) is the login/register screen, which the user sees on first launch or after having logged out.

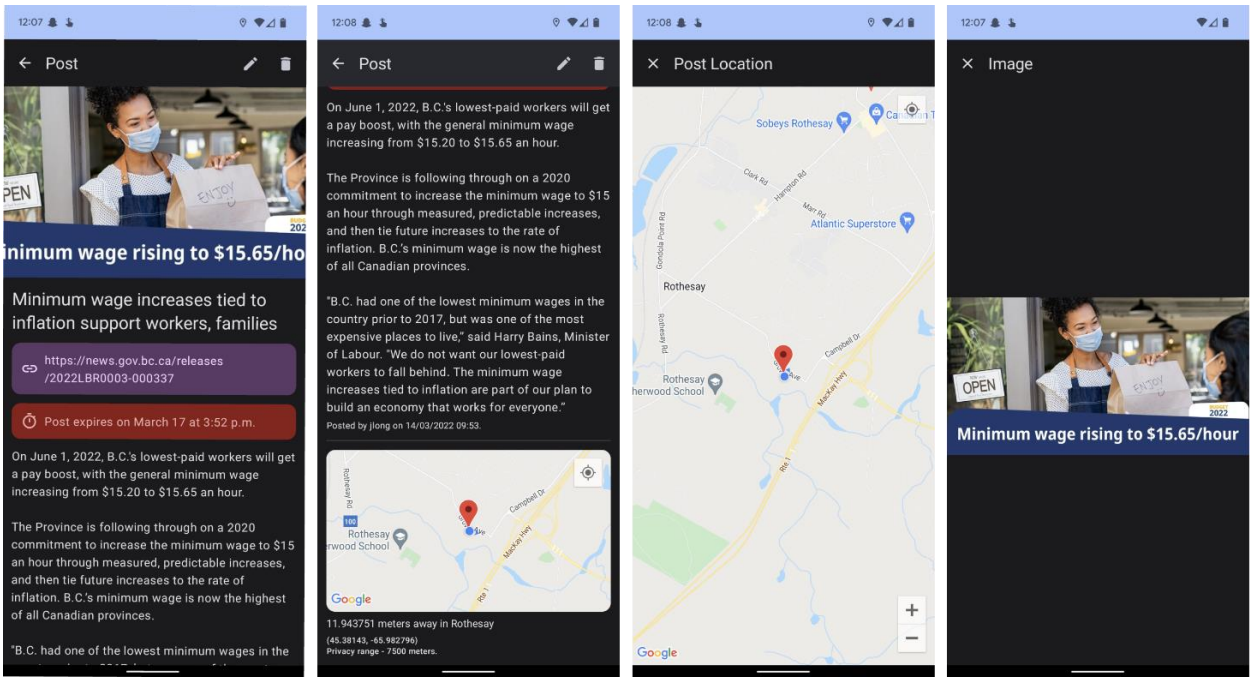

Figure 4 - More screenshots of the Android client user interface.

In figure 2, there are 4 more screenshot of the UI, showing components of the focused post UI. In image 1 and 2 of the figure (left most), there is the post UI that the user sees once they're clicked on a post. Image 1 shows the top of the UI, showing the embedded image, title, included URL (which can be clicked to open the phone browser), expiration date, and contents of the post. Image 2 shows the bottom of this UI, showing the author, submission date, map of the location, and distance from the user of the post. Image 3 shows the full screen map UI which can be opened by clicking the map in image 2. On this screen, the user can see the post location marker (the red marker) and their own location (the blue dot). In image 4 (right most) the full screen image UI is shown, which can be opened by clicking on the picture in image 1.

4.2.3 UML Diagram

The following is a UML diagram which describes the file structure of the Android Client source code. Below this diagram, a description of each file can be found.

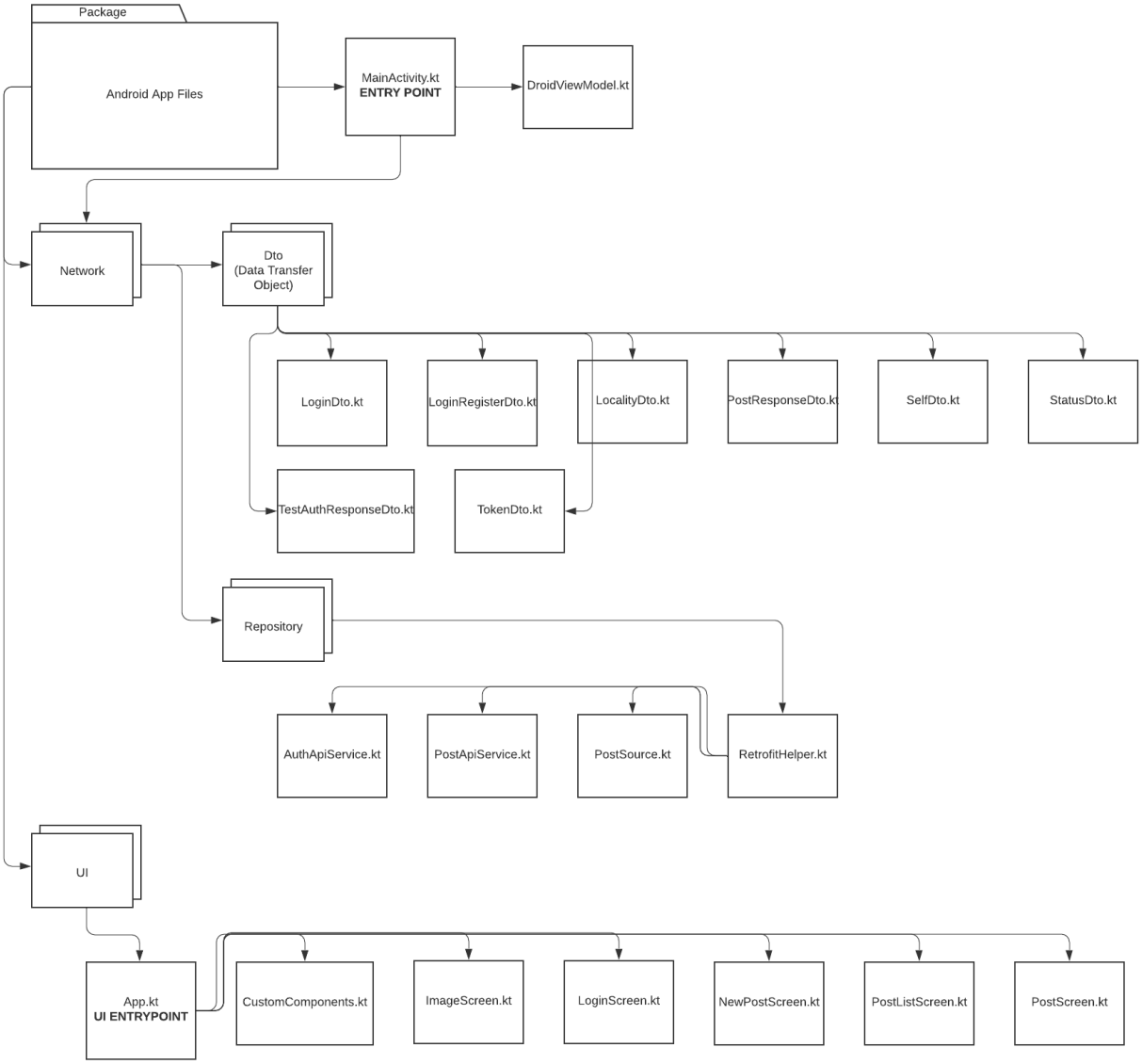

Figure 5 - UML Diagram of App Source Code

- **MainActivity.kt –** App entry point, which initializes the app's view model and UI.
- **DroidViewModel.kt –** Implements the app's view model, as per the MVVM design pattern, which is detailed above.
- **network [Folder] –** Files in this folder implement the data transfer objects (or DTOs) and API structure required to interface with the Droid server.
    - **dto [Folder] –** Files in this folder implement the DTOs which model requests and responses from the server.
        - **LocalityDto.kt** – Implements the DTO which models the Locality model from the server.
        - **LoginRegisterDto.kt –** Implements the DTO which models requests and responses for the login and register APIs.

        - **LogoutResponseDto.kt –** Implements the DTO which models the logout API response.
        - **PostResponseDto.kt -** Implements the DTOs which model the Post server model, and related requests and responses.
        - **SelfDto.kt -** Implements the DTOs which model the self API requests and responses.
        - **StatusDto.kt –** Implements the DTO which models a generic server response.
        - **TestAuthResponseDto.kt -** Implements the DTO which models the test auth API response.
        - **TokenDto.kt** - Implements the DTO which models the token response from the server, required for authentication.
    - **repository [Folder] -** Files in this folder implement API definitions and other network helper functions.
        - **AuthApiService.kt -** Implements the API for user-oriented APIs.
        - **PostApiService.kt -** Implements the API for post APIs.
        - **PostSource.kt -** Implements the pagination source for displaying large amounts of posts.
        - **RetrofitHelper.kt -** Implements the initialization of the networking files.
- **ui [Folder] –** Files in this folder implement the user interface of the Droid client.
    - **App.kt –** Entry point for the app's UI, including the navigation routing.
    - **CustomComponents.kt –** Implements a collection of UI components that aren't specific to any screen and so are reusable.
    - **ImageScreen.kt –** Implements the full screen view of post images.
    - **LoginScreen.kt –** Implements the login and register landing page.
    - **NewPostScreen.kt –** Implements the new post screen, where users write and submit new posts.
    - **PostListScreen.kt –** Implements the main app page, where users can scroll through nearby posts.
    - **PostScreen.kt –** Implements the post screen, where users can view a post and its contents.

5 Conclusions and recommendations

In this report, we discussed various concepts including Android development, API development, and location based social networks in great length. Following this, we detailed and implemented the Droid system, a location based social network which takes advantage of GPS technologies, REST API patterns, and the Android platform to enable a community focused and location aware method of communicating with one's neighbours and colleagues. By using the Android app developed throughout this report, users can make geo-locked posts which are only available to other users within range of said

posts, giving an enhanced sense of community and privacy to those who take advantage of the platform, fostering growth of local communities digitally.

At the core of the functionality of this system is its use of GPS and local community identifiers too offer both fine grain and community wide reach for users. Both of these methods of filtering rely on the coordinates of the devices which query from and post to the server, requiring that the service trusts that the coordinates being reported by devices are accurate. Unfortunately, GPS locations can easily be spoofed, or faked, on the Android platform through use of alternate GPS provider apps. This means that users can fake their locations to post to and read posts from areas they are not within, breaking the privacy that users may expect from the platform. Due to this vulnerability, which is often referred to as GPS spoofing and affects many location based applications such as Snapchat's snap map feature, we recommend further research into methods to combat GPS spoofing. One possible method that may be of interest for a service with a large enough and dense enough userbase would use the Bluetooth identifiers of nearby devices. These identifiers could be associated with user accounts and thus user location patterns, and if a user was able to see a Bluetooth identifier that was believed to be far away from the user's reported coordinates, the service would be able to doubt the user's reported location.

6 References

[1] Ngak C. Then and now: a history of social networking sites. Cbsnews.com. 2011 Jul 6 [accessed 2022 Mar 24]. https://www.cbsnews.com/pictures/then-and-now-a-history-of-social-networking-sites/2/

[2] Meta. Facebook MAU worldwide 2021. Statista. 2022 Feb [accessed 2022 Mar 24]. https://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/

[3] GitHub. Build software better, together. GitHub. 2013 [accessed 2022 Mar 24]. https://github.com

[4] Wikimedia Foundation. Wikipedia. Wikipedia.org. 2014 [accessed 2022 Mar 24]. https://wikipedia.org

[5] GoFundMe. Hurricane Ida Fundraisers. GoFundMe. [accessed 2022 Mar 24]. https://www.gofundme.com/c/act/hurricane-ida-fundraisers

[6] Google. Android. Android. 2016 [accessed 2022 Mar 24]. https://www.android.com/

[7] Statcounter. Mobile Operating System Market Share Worldwide. StatCounter Global Stats. 2021 [accessed 2022 Mar 24]. https://gs.statcounter.com/os-market-share/mobile/worldwide

[8] Google. Android Open Source Project. Android Open Source Project. [accessed 2022 Mar 24]. https://source.android.com/

[9] Amadeo R. Google's iron grip on Android: Controlling open source by any means necessary. Ars Technica. 2018 Jul 21 [accessed 2022 Mar 24]. https://arstechnica.com/gadgets/2018/07/googles-iron-grip-on-android-controlling-open-source-by-any-means-necessary/

[10] Google. Android's Kotlin-first approach. Android Developers. [accessed 2022 Mar 24]. https://developer.android.com/kotlin/first

[11] Jetbrains. A modern programming language that makes developers happier. Kotlin. 2020. https://kotlinlang.org/

[12] Google. Why Compose | Jetpack Compose. Android Developers. 2022 Mar 11 [accessed 2022 Mar 24]. https://developer.android.com/jetpack/compose/why-adopt

[13] Facebook. React – A JavaScript library for building user interfaces. Reactjs.org. 2019 [accessed 2022 Mar 24]. https://reactjs.org/

[14] You E. Vue.js. Vue JS. 2000 [accessed 2022 Mar 24]. https://vuejs.org/

[15] Fowler M. Presentation Model. martinfowler.com. 2004 Jul 19 [accessed 2023 Mar 24]. https://martinfowler.com/eaaDev/PresentationModel.html

[16] Microsoft. The Model-View-ViewModel Pattern - Xamarin. Microsoft.com. 2017 Aug 7 [accessed 2022 Mar 24]. https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm

[17] Mishra R. MVVM (Model View ViewModel) Architecture Pattern in Android. GeeksforGeeks. 2020 Oct 29. https://www.geeksforgeeks.org/mvvm-model-view-viewmodel-architecture-pattern-in-android/

[18] RedHat. What is an API? www.redhat.com. 2017 Oct 31 [accessed 2022 Mar 24]. https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces#history-of-apis

[19] Box D. A Young Person's Guide to The Simple Object Access Protocol: SOAP Increases Interoperability Across Platforms and Languages. docs.microsoft.com. 2019 Oct 25 [accessed 2022 Mar 24]. https://docs.microsoft.com/en-us/archive/msdn-magazine/2000/march/a-young-person-s-guide-to-the-simple-object-access-protocol-soap-increases-interoperability-across-platforms-and-languages

[20] SmartBear. SOAP vs REST. What. SmartBear.com. 2020 Jan 2 [accessed 2022 Mar 24]. https://smartbear.com/blog/soap-vs-rest-whats-the-difference/

[21] RedHat. SOAP vs REST. www.redhat.com. 2017 Oct 31 [accessed 2022 Mar 24].

https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces#soap-vs-rest

[22] Zheng Y. Location-Based Social Networks: Users. 2011. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/trajectorycomputing-lbsn-chapter8.pdf

[23] Ha A. Barry Diller Says Tinder Succeeded Because IAC Left Its Founders Alone. TechCrunch. 2014 Dec 3 [accessed 2022 Mar 24]. https://techcrunch.com/2014/12/03/barry-diller-on-tinder/

[24] NextDoor. Join Nextdoor, an app for neighborhoods where you can get local tips, buy and sell items, and more. ca.nextdoor.com. [accessed 2022 Mar 24]. https://ca.nextdoor.com/

[25] YikYak Inc. Yik Yak | The Yak is Back. yikyak.com. [accessed 2022 Mar 24]. https://yikyak.com/

[26] Hatmaker T. Yik Yak returns from the dead. TechCrunch. 2021 Aug 16. https://techcrunch.com/2021/08/16/yik-yak-is-back/

[27] Microsoft. Visual Studio Code. Visualstudio.com. 2016 Apr 14 [accessed 2022 Mar 24]. https://code.visualstudio.com/

[28] Android Studio. Android Studio and SDK tools. Android Developers. 2019 [accessed 2022 Mar 24]. https://developer.android.com/studio

[29] Retrofit. Retrofit. Github.io. 2013 [accessed 2022 Mar 24]. https://square.github.io/retrofit/

[30] Square I. OkHttp. Github.io. 2019 [accessed 2022 Mar 24]. https://square.github.io/okhttp/

[31] Google. GSON. GitHub. 2019 May 9 [accessed 2022 Mar 24]. https://github.com/google/gson

[32] Django. The Web framework for perfectionists with deadlines | Django. Djangoproject.com. 2019 [accessed 2022 Mar 24]. https://www.djangoproject.com/

[33] Christie T. Home - Django REST framework. Django-rest-framework.org. 2011 [accessed 2022 Mar 24]. https://www.django-rest-framework.org/

[34] Tivix. Deprecated. GitHub. 2022 Mar 23 [accessed 2022 Mar 24]. https://github.com/Tivix/django-rest-auth/blob/master/docs/index.rst

[35] Salesforce. Cloud Application Platform | Heroku. Heroku.com. 2020 Jan 25 [accessed 2022 Mar 24]. https://www.heroku.com/

[36] Nightingale. Nightingale. Nightingale. [accessed 2022 Mar 24]. https://nightingale.rest/

[37] Imgur. Imgur API. Imgur API. [accessed 2022 Mar 24]. https://apidocs.imgur.com/

[38] Snapchat. Snapchat Support. Snapchat.com. 2022 [accessed 2022 Mar 24]. https://support.snapchat.com/en-US/a/about-stories

Appendix A: REST API Documentation

POST /api/post/

Description: Create a new post
Headers:
AUTHORIZATION -> Token {token: String}
Description: API token, used to identify user and permissions
Body (JSON Fields):
coord_longitude:
Description: Client's current longitude, float value, required
coord_latitude:
Description: Client's current latitude, float value, required
range_meters:
Description: Privacy range, range in which users can query post from post's coordinates, Int value, required.
extend_to_locality:
Description: Determines if valid query area should be extended to locality, required, Boolean
title:
Description: Title of post, required, String
content:
Description: Content of post, required, String
ref_url:
Description: URL of related article, if applicable, optional, String (URL format)
end_date:
Description: Time and Date of post's expiration date, in ISO-zoned date/time string format and in the UTC time zone. Optional, string.
image_base64:
Description: Base64 representation of the embedded image. Optional, string.
Successful Return:


```
{
 "status": "success",
 "data": {
            "id": 19,
            "title": "Privacy Rights! Locality Mode.",
            "content": "first radius rights post.",
            "ref_url": "",
            "image": "",
            "created_on": "2022-01-29T16:03:12.880068Z",
            "coord_longitude": -65.986625286370909,
            "coord_latitude": 45.3888,
            "range_meters": 60000,
            "extend_to_locality": true,
            "author": 1,
            "locality": {
              "name": "Rothesay",
              "google_place_id": "ChIJ_2vsoe-sp0wRwi0gFylfEZs"
            }
 }
}
```


GET /api/post

Description: Retrieve posts in users location.

Headers:

AUTHORIZATION -> Token {token: String}

Description: API token, used to identify user and permissions

Query Parameters:

limit: `?limit={limit: Int}`

Description: Number of posts to return, optional, default = 10

page: `?page={page: Int}`

Description: Page of posts of size limit from the most recent posts to return, used for pagination, optional, default = 0

lon: `?lon={longitude: Float}`

Description: Client's current longitude, float value, required

lat: `?lat={latitude: Float}`

Description: Client's current latitude, float value, required

range: `?range={range: Int}`

Description: Filter range (meters), range from user's coordinates to return posts from, Int value, optional.

If not included, locality mode will be used. If coordinates don't result in valid locality, HTTP 422 error, code "invalidLocalityError"

Successful Return:


```
{
 "status": "success",
 "data": [
  {
   "id": 19,
   "title": "Privacy Rights! Locality Mode.",
   "content": "first radius rights post.",
   "ref_url": "",
   "image": "",
   "created_on": "2022-01-29T16:03:12.880068Z",
   "coord_longitude": -65.986625286370909,
   "coord_latitude": 45.3888,
   "range_meters": 60000,
   "extend_to_locality": true,
   "author": "jlong",
   "locality": {
    "name": "Rothesay",
    "google_place_id": "ChIJ_2vsoe-sp0wRwi0gFylfEZs"
   }
  },
  ...

 ]
}
```


GET /api/post/{id: Int}/`

Description: Retrieve post with id provided if post is visible to user or user is the author.

Headers:

AUTHORIZATION -> Token {token: String}

Description: API token, used to identify user and permissions

Query Parameters:

lon: `?lon={longitude: Float}`

Description: Client's current longitude, float value, required

lat: `?lat={latitude: Float}`
Description: Client's current latitude, float value, required
range: `?range={range: Int}`
Description: Filter range (meters), range from user's coordinates to return posts from
Int value, optional.

If not included, locality mode will be used. If coordinates don't result in valid locality, HTTP 422 error, code "invalidLocalityError"

Successful Return:


```
{
 "status": "success",
 "data": {
  "id": 19,
  "title": "Privacy Rights! Locality Mode.",
  "content": "first radius rights post.",
  "ref_url": "",
  "image": "",
  "created_on": "2022-01-29T16:03:12.880068Z",
  "coord_longitude": -65.986625286370909,
  "coord_latitude": 45.3888,
  "range_meters": 60000,
  "extend_to_locality": true,
  "author": "jlong",
  "locality": {
   "name": "Rothesay",
   "google_place_id": "ChIJ_2vsoe-sp0wRwi0gFylfEZs"
  }
 }
}
```


DELETE /api/post/{id: Int}/

Description: Delete post with id, only if user is the post's author.

Headers:

AUTHORIZATION -> Token {token: String}
Description: API token, used to identify user and permissions
User of the token must resolve to be the author.

Successful Return:


```
{
 "status": "success",
 "data": "Post deleted."
}
```


PATCH /api/post/{id: Int}/

Description: Edit post with id, only if user is the post's author.

Headers:

AUTHORIZATION -> Token {token: String}

Description: API token, used to identify user and permissions
User of the token must resolve to be the author.

Body (JSON Fields):

content:

optional, String

ref_url:

optional, String

range:

optional, Int

extend_to_locality:

optional, Boolean

title:

optional, String

end_date:

optional, String

image_base64:

optional, String

Successful Return:


```
{
 "status": "success",
 "data": {
  "id": 20,
  "title": "Privacy Rights! Locality Mode.",
  "content": "first radius rights post. Edited.",
  "ref_url": "",
  "image": "",
  "created_on": "2022-01-30T21:55:00.03382Z",
  "coord_longitude": -65.986625286370909,
  "coord_latitude": 45.3888,
  "range_meters": 60000,
  "extend_to_locality": true,
  "author": 1,
  "locality": "ChIJ_2vsoe-sp0wRwi0gFylfEZs"
 }
}
```


GET /api/self/

Description: Get user account and current locality information.

Headers:

AUTHORIZATION -> Token {token: String}

Description: API token, used to identify user

Query Parameters:

lon: `?lon={longitude: Float}`

Description: Client's current longitude, float value, optional

lat: `?lat={latitude: Float}`

Description: Client's current latitude, float value, optional

Successful result:


```
{
 "status": "success",
 "user": {
  "id": 1,
  "password": "pbkdf2_sha256$320000$4sgIdSbDUqD0DT8HrQiKf1$pPnGcd1+fNQR+VUyEGaFp/fiESywn4+/yKycZuzsslE=",
  "last_login": "2022-01-27T21:43:07.270883Z",
  "is_superuser": true,
  "username": "jlong",
  "first_name": "",
  "last_name": "",
  "email": "jlong.anthony@live.ca",
  "is_staff": true,
  "is_active": true,
  "date_joined": "2022-01-16T01:58:29.242886Z",
  "groups": [],
  "user_permissions": []
 },
 "locality": {
  "google_place_id": "ChIJ_2vsoe-sp0wRwi0gFylfEZs",
  "name": "Rothesay"
 }
}
```


GET /api/self/posts/

Description: Get all posts submitted by user.

POST /api/login/

Description: Login to user account, get a token for authentication.
Body (JSON Fields):
- username:
  - required, String
- password:
  - required, String

Successful Result:


```
{
 "key": "d5ec9fac767cc70d87f40ce43855266576eab0f0"
}
```


POST /api/register/

Description: Create a user account.
Body (JSON Fields):
- username:
  - required, String
  - Must be unique, error 400 if not.
- password:

required, String
Must not be empty, error 400 if so.

Successful Result:


```
{
 "status": "success",
 "user": {
  "id": 7,
  "password": "pbkdf2_sha256$320000$ef6YLTzihZ0cv56A3gLouy$ThLW9SY/BLofJ+EziR0MUJDKD85FOwH1WIkXhMjJqSc=",
  "last_login": null,
  "is_superuser": false,
  "username": "testUser4",
  "first_name": "",
  "last_name": "",
  "email": "",
  "is_staff": false,
  "is_active": true,
  "date_joined": "2022-01-31T00:11:08.192172Z",
  "groups": [],
  "user_permissions": []
 }
}
```


POST /api/logout/

Headers:

AUTHORIZATION -> Token {token: String}
Description: API token, used to identify user

Successful Result:


```
{
 "detail": "Successfully logged out."
}
```


GET /api/test-auth/

Headers:

AUTHORIZATION -> Token {token: String}
Description: API token, used to identify user

Successful Result:


```
{
 "status": "success",
 "data": "Hello jlong!"
}
```

Appendix B: Server Source Code Excerpt

While the full source code for both the server and app would be far too lengthy to be included in an appendix, it is believed may be useful to include an excerpt from the server source code. The selected section is the post_views.py file, which implements the post-oriented API endpoints. Full server and app source code is available upon request as a supplementary material.

```
from datetime import datetime, timezone
from http import HTTPStatus
from rest_framework.views import APIView
from rest_framework import permissions, authentication
from rest_framework.response import Response
from rest_framework import status
from api.imgur_requests import delete_image

from api.request_modifiers import author_pk_to_username

from .serializers import PostSerializer
from .models import Post
from django.db.models.query import QuerySet
from django.db.models import F
from django.shortcuts import get_object_or_404

from .distance import haversine

from .request_modifiers import add_image, add_ownership_from_id, add_user_as_author, user_is_user,
add_locality, place_id_to_name

class PostView(APIView):
    authentication_classes = (authentication.TokenAuthentication,)
    permission_classes = (permissions.IsAuthenticated,)
    serializer_class = PostSerializer

    def post(self, request):
        modified_request = request.data.copy()

        if 'coord_longitude' not in modified_request or 'coord_latitude' not in modified_request:
            return Response(
                {"status": "error", "data": "coord_longitude or coord_latitude not provided."},
                status=HTTPStatus.BAD_REQUEST
            )

        ### remove these fields from request if present
        modified_request.pop('author', None)
        modified_request.pop('locality', None)
        modified_request.pop('created_on', None)
        modified_request.pop('image', None)
        ###

        ### add database items that need a little processing
        modified_request = add_user_as_author(modified_request, request.user)
        modified_request = add_locality(modified_request)
        modified_request = add_image(modified_request) # this function will handle case of no image
        ###

        serializer = self.serializer_class(data = modified_request)
        serializer.is_valid(raise_exception=True)
        serializer.save()

        ### replace some fields with human readable fields
        modified_data = place_id_to_name(serializer.data)
```

```
        ###

        return Response(
            {"status": "success", "data": modified_data}
        )

    def get(self, request, id=None):
        ##
        # validate location from user request
        ##
        if request.GET.get("lat") == None or request.GET.get("lon") == None:
            return Response (
                {
                    "data": "Must provide both coord_latitude and coord_longitude, and optionally
range_meters.",
                    "status": "error"
                },
                status=HTTPStatus.BAD_REQUEST
            )
        location = {
            "coord_longitude": float(request.GET.get("lon")),
            "coord_latitude": float(request.GET.get("lat"))
        }
        ## validate range given by user
        if request.GET.get("range") != None:
            location.update({"range_meters": int(request.GET.get("range"))})
        if "range_meters" in location:
            if location["range_meters"] < 10 or location["range_meters"] > 10000:
                return Response (
                    {
                        "data": "Filter range must be within range [10, 10000] (inclusive).",
                        "status": "error",
                        "code": "rangeRangeError"
                    },
                    status=HTTPStatus.BAD_REQUEST
                )

        ## get user's locality data from google API
        modified_request = add_locality(location)

        ## route request based on location query type [range | locality]
        if 'range_meters' in location:
            return self.get_by_coords(request, location, id)
        else:
            modified_request = add_locality(location)
                # return error is no_location locality (not enough info to continue query)
            return self.get_by_locality(request, modified_request, id)

    def get_by_coords(self, request, location, id = None):
        limit = 10 if request.GET.get("limit") == None else int(request.GET.get("limit"))
        page = 0 if request.GET.get("page") == None else int(request.GET.get("page"))

        ## request can either be query for list of posts or for single post [id]
        if id == None:
            # filter based off of range, using haversine method
            posts = Post.objects.annotate(
                    distance = haversine(F('coord_longitude'), F('coord_latitude'),
location['coord_longitude'], location['coord_latitude'])
                ).filter(
                    distance__lte=location['range_meters']
                ).filter(
                    distance__lte=F('range_meters')
                )

            # filter further by post end_time
            posts_active = (posts.filter(end_time__gt = datetime.now(timezone.utc)) |
posts.filter(end_time = None)).distinct().order_by('-created_on') [page*limit:page*limit+limit]

            #create json serializer
```

```
            post_serializer = self.serializer_class(posts_active, many = True)

            # human readability modifiers
            modified_data = add_ownership_from_id(post_serializer.data, request.user, many = True)
            modified_data = author_pk_to_username(modified_data, many = True)
            modified_data = place_id_to_name(modified_data, many = True)

            return Response({
                "status": "success",
                "data": modified_data
            })
        else:
            post = get_object_or_404(Post, pk = id)

            # ensure post is within range of user
            in_range = len(
                Post.objects.filter(
                    pk=id
                ).annotate(
                    distance = haversine(F('coord_longitude'), F('coord_latitude'),
location['coord_longitude'], location['coord_latitude'])
                ).filter(
                    distance__lte = location['range_meters']
                ).filter (
                    distance__lte = F('range_meters')
                )
            ) == 1

            # if not in range or owned by requesting user, deny access
            if not user_is_user(request.user, post.author.username) and not in_range:
                return Response({
                    "status": "error",
                    "data": f"Post with id={id} outside of ranges (post/privacy range =
{post.range_meters}, user/filter range = {location['range_meters']}).",
                    "code" : "rangeError"
                }, status=HTTPStatus.UNAUTHORIZED)
            # create post serializer, annotating distance from user
            post_serializer = self.serializer_class(
                Post.objects.annotate(
                    distance = haversine(F('coord_longitude'), F('coord_latitude'),
location['coord_longitude'], location['coord_latitude'])
                ).get(pk = post.id)
            )

            # human readability modifiers
            modified_data = add_ownership_from_id(post_serializer.data, request.user)
            modified_data = author_pk_to_username(modified_data)
            modified_data = place_id_to_name(modified_data)

            return Response({
                "status": "success",
                "data": modified_data
            })

    def get_by_locality(self, request, location, id = None):
        limit = 10 if request.GET.get("limit") == None else int(request.GET.get("limit"))
        page = 0 if request.GET.get("page") == None else int(request.GET.get("page"))

        ##
        # at this point locality is just the google_place_id ( a char field ), and no_location has "0" for
this
        # if == 0, no available locality, so deny access
        if location['locality'] == "0":
            return Response(
                {
                    "status": "error",
                    "data": "By ommiting a range, locality mode was selected, but no valid locality could
be found for the provided locality.",
                    "code" : "invalidLocalityError"
```

```
                },
                status=HTTPStatus.UNPROCESSABLE_ENTITY
            )

        ## request can either be query for list of posts or for single post [id]
        if id == None:
            # check for posts with auto extend to locality on
            posts_extend = Post.objects.filter(
                extend_to_locality = True, locality__google_place_id = location['locality']
            )

            # check for posts without this but still within their privacy range
            posts_no_extend = Post.objects.annotate(
                distance = haversine(F('coord_longitude'), F('coord_latitude'),
location['coord_longitude'], location['coord_latitude'])
            ).filter(
                extend_to_locality = False,
                locality__google_place_id = location['locality'],
                distance__lte = F('range_meters')
            )
            combined_results = (posts_extend | posts_no_extend).distinct()

            # filter by post end_date
            results_active = (combined_results.filter(end_time__gt = datetime.now(timezone.utc)) |
combined_results.filter(end_time = None)).distinct().order_by('-created_on') [page*limit:page*limit+limit]

            #create post serializer
            post_serializer = self.serializer_class(results_active, many = True)

            # human readability modifiers
            modified_data = add_ownership_from_id(post_serializer.data, request.user, many = True)
            modified_data = author_pk_to_username(modified_data, many = True)
            modified_data = place_id_to_name(modified_data, many = True)

            return Response({
                "status": "success",
                "data": modified_data
            })
        else:
            post = get_object_or_404(Post, pk = id)

            # ensure post is within users range
            in_range = (
                post.extend_to_locality and post.locality.google_place_id == location['locality']
            ) or (
                len(
                    Post.objects.filter(
                        pk = post.pk
                    ).annotate(
                        distance = haversine(F('coord_longitude'), F('coord_latitude'),
location['coord_longitude'], location['coord_latitude'])
                    ).filter(
                        extend_to_locality = False,
                        locality__google_place_id = location['locality'],
                        distance__lte = F('range_meters')
                    )
                ) == 1
            )

            # if not in range or owned by requesting user, deny access
            if not user_is_user(request.user, post.author.username) and not in_range:
                return Response({
                    "status": "error",
                    "data": f"Post with id={id} outside of ranges (post/privacy range =
{post.range_meters}, user/filter range = {location['range_meters']}).",
                    "code": "rangeError"
                }, status = HTTPStatus.UNAUTHORIZED)

            # create post serializer, annotating distance from user
```

```
            post_serializer = self.serializer_class(
                Post.objects.annotate(
                    distance = haversine(F('coord_longitude'), F('coord_latitude'),
location['coord_longitude'], location['coord_latitude'])
                ).get(pk = post.id)
            )

            # human readability modifiers
            modified_data = add_ownership_from_id(post_serializer.data, request.user)
            modified_data = author_pk_to_username(modified_data)
            modified_data = place_id_to_name(modified_data)

            return Response({
                "status": "success",
                "data": modified_data
            })

    def delete(self, request, id):
        post = get_object_or_404(Post, pk = id)
        if not user_is_user(request.user, post.author.username):
            return Response(status=status.HTTP_403_FORBIDDEN, data = {"detail": "Refused: cannot delete
another users post :/."})
        if post.image != None:
            delete_image(post.image.delete_hash)
            post.image.delete()
        post.delete()
        return Response({
            "status": "success", "data": "Post deleted."
        })

    def patch(self, request, id=None):
        post = get_object_or_404(Post, pk = id)
        modified_request = request.data.copy()
        # remove these fields from request if present
        modified_request.pop('author', None)
        modified_request.pop('locality', None)
        modified_request.pop('created_on', None)
        modified_request.pop('coord_longitude', None)
        modified_request.pop('coord_latitude', None)
        modified_request = add_image(modified_request)
        if not user_is_user(request.user, post.author.username):
            return Response(status=status.HTTP_403_FORBIDDEN, data = {"detail": "Refused: cannot edit
another users post >:|."})
        serializer = PostSerializer(post, modified_request, partial = True)
        serializer.is_valid(raise_exception=True)
        serializer.save()

        modified_data = place_id_to_name(serializer.data)
        return Response({"status": "success", "data": modified_request})
```