Abstract

Over the recent years, social media has achieved huge popularity through users sharing original multimedia with their friends, family, colleagues and even the public. When people are depicted in media, mainly photos, they assume co-ownership of that media and should be able to manage their privacy accordingly. Social media sites currently only offer *all or nothing* approaches of untagging or requesting that a photo is deleted. These approaches hinder sharing media to different groups of people (e.g. one user may have a privacy preference of "I don't want Bob to see this photo", but not mind if other friends see it.). Following this, a conflicting privacy preference may occur (e.g. another tagged user has a preference of "Why not? I want Bob to see this photo.") and thus this also needs resolving via a mediator. This project explores creating a tool for users on Facebook to collectively manage their privacy.

Acknowledgements

My supervisor, Dr. Jose M. Such. His guidance and expertise helped me deliver this functional project and understand the existing research that has gone into this topic.

My friends and family for supporting me throughout the duration of this project and to those who contributed feedback for the study.

Table of Contents

List of figures

Introduction

Privacy plays an important role in society as it describes how people protect their information by choosing with whom they share it with. It is desirable to protect your own information as adversaries could use it to commit crimes such as fraud or cyberstalking against you, or lead to unintended consequences such as losing your job or tarnish your self-representation (Such and Criado, 2016).

With the rise of Social Media and Social Network Services (SNS), billions of people are now involved in maintaining online relationships through sharing information with each other. Currently, in most popular SNS, the uploader of media manages the privacy policy for that media. However, what happens when a picture depicting multiple people or a different person is uploaded? e.g. Alice uploads a photo containing herself and Bob. In this scenario, Alice and Bob are co-owners of the photo, where something they are responsible for (themselves) is depicted (Fogues *et al.*, 2015). Currently Bob has little power over whom gets to see the photo with current tools available on SNS' (Fogues *et al.*, 2015) as no popular SNS offer support for collective decision making (Misra and Such, 2016) and so co-owners are left to negotiate via other means or untagging.

Project Aims and Objectives

The aim of this project is to develop a tool utilising the Facebook Graph API that facilitates reaching an agreement between co-owners on whom can access shared photos. The tool will automatically generate groups using community detection and detect conflicts from user-defined privacy policies. It will use an automated negotiation strategy to suggest a resolution for any given conflicts that arise.

To meet these aims, this tool has the following objectives:

- Co-owners should be aware of the audience that has access to a shared item.
- Allow users to define a privacy policy for groups discovered by a community detection algorithm.
- Use an automated negotiation algorithm to suggest resolutions to conflicts found.
    - Discover what other variables might influence conflict resolution through user feedback.
- Use non-identifiable information to evaluate how successful the algorithms are and identify further improvements.

Background and Literature Survey

The articles read presented the initial problem of co-owners of photos having little option in managing their privacy for photos they had not uploaded. Most SNS enforce a policy where only the uploader can manage the privacy policy of shared items. Without complaining via other means, this leaves co-owners with the single option of un-tagging themselves from the photo (Such *et al.*, 2017). However, untagging only prevents your unique friends from viewing the photo. Mutual friends, along with the uploader's friends would still be able to view the photo depending on the uploader's privacy settings.

To resolve this, further research has been done on how to identify groups of people to apply a privacy policy to. Facebook is the only major SNS that offers an element of computer supported

grouping. This comes in the form of “smart lists” where friend lists (groups) are automatically created based on friends that have a similar location, institution or other non-personal information. Most other SNS offer user-defined or pre-defined groups (Misra and Such, 2016). Misra, Such and Balogun implemented and compared 8 community detection algorithms to how 31 real users would grant access to photos for each friend individually. They found that the Clique-Percolation Method (CPM) was most effective but computationally expensive and not good enough for everyone (Misra, Jose M. Such and Balogun, 2016).

Once users have defined privacy policies for groups and photos, some conflicts will likely occur. Such and Criado present a mechanism for detecting conflicts between co-owners’ privacy policy for a photo and suggesting a resolution by estimating a user’s willingness to concede based on their sensitivity to the item. They tested this along with generic mechanisms (uploader-overwrites, majority-voting and veto-voting) in how often they matched concession behaviours from a study of 50 people. The results show that their mechanism was the most successful, with an ~85% match rate, as the generic mechanisms were not flexible enough in different situations (Such and Criado, 2016).

Main Result

Theoretical Development

This section identifies what methods and tools will be used in the development of the project.

Functional Specification

The functional specification, identifying the stakeholders in an agile fashion, for the system follows:

- As a social media user, I want to log in with my Facebook account, in order to use the application.
- As a social media user, I want to see which photos I am tagged in, in order to be aware of which photos may breach my privacy.
- As a social media user, I want to add context tags to photos I am tagged in, in order to help a mediator and define my privacy policies.
- As a social media user, I want to define what contexts a group of friends can access, in order to define my privacy policies.
- As a social media user, I want a community detection algorithm to determine groups of friends, in order to reduce my burden.
- As a social media user, I want a mediator to suggest resolutions to conflicts in my friends and my privacy policies.

Non-functional specification

The non-functional specification and properties of this system are the following:

- The system must be highly available, in order to sustain lots of users.
- The system must be horizontally scalable, in order to be cost-efficient with varying amounts of users over time.
- The system must be asynchronous, in order to perform slow calculations without reducing availability for other users.
- The system must be accessible on different devices, in order for people to the application on the go or at a desktop.
- The application must be simple to use in order to engage most users.
- The system must be secure as it stores personally identifiable information about its users in order to comply with laws and protect user privacy.

Methodology

This section outlines the development practices used whilst developing the system.

Agile Kanban

Kanban is a popular agile development methodology that uses the Just-In-Time (JIT) process to complete features of a project. It allows for a continuous flow of software releases (continuous delivery) and the whole team takes responsibility for moving features along the workflow (Radigan and Altassian, 2017).

Model-View-Controller (MVC)

MVC describes an architectural structure of developing applications that have a user interface. A model represents the domain-specific simulation of an entity, for example a *Building*. A view presents a user interface that shows data from models and can be composed of other views. A controller interfaces between models and views, defining the actions a user can take upon a model through a view (Krasner and Pope, 1988).

Inversion of Control (Dependency Injection)

Writing modular code results in an easily maintainable code-base. By writing small classes, modules and functions that have dependencies injected, we can stub out the dependencies during testing without having to have finished the implementation of them (Fowler, 2004). This also means that those dependencies can be implemented as interfaces and the implementation becomes swappable. E.g. a storage interface could be implemented by an SQL database class or a NOSQL database class.

Domain Driven Design/Development (DDD)

With MVC and Inversion of Control, Domain Driven Design helps organise a system by considering the domain context and models involved. This allows a system to be expanded easily by adding onto an existing core model (Evans, 2003).

Micro Services

By splitting a system into smaller services that communicate with each other, we create an interchangeable system that is easier to understand architecturally as the services are simpler by being responsible for fewer tasks.

Message Queueing

In this project, there will be computationally expensive algorithms that will delay responses if they were implemented in a simple HTTP request-response scenario and the service availability would be reduced. To solve this problem, message queues provide an asynchronous communication protocol where a producer that places messages onto a queue does not require a response immediately (Johansson, 2014). A consumer service (worker) can then pick up the messages one-by-one and perform the tasks without being overwhelmed.

Publish-Subscribe Pattern

To compliment Message Queueing, the project uses the publish-subscribe pattern to deliver an asynchronous message when a consumer finishes a task. The web application can connect to a web socket to receive these updates and update its local information accordingly.

Continuous Integration

During development, it is useful to test and deploy newer changes and iterations as soon as possible so that stakeholders can provide feedback. Continuous integration software can be used to run our automated tests on new commits to a repository and automatically deploy if those tests are successful. This project will use the hosted Travis CI service for continuous integration as it offers free builds for open-sourced projects. An alternative self-hosted

solution would be to use Jenkins; however, this requires further set up, maintenance as well as cost.

Analysis and Design

This section considers the different ways of implementing the system, explaining why some implementations are more suitable than others.

Algorithms

To determine groups of users, the clique percolation method will be used as it was determined to be the most successful from Misra, Such and Balogun's research as mentioned in the Background and Literature Survey. They used libraries to determine cliques from an input, but these are made for traversing completed friendship graphs. Due to the Facebook Graph API only showing mutual friends whom use the application, the graph of friendships that the application is aware of grows whenever new users join. Existing users may also form new friendships with each other during the lifetime of the application, thus changing or merging existing cliques. This algorithm therefore must:

- Compare found cliques to existing cliques, where if
    - The clique already exists, make no changes.
    - An existing clique is a subset of a newly found clique, merge the subset cliques into the new clique.
- Only traverse necessary parts of the graph.

Clique persistence can be achieved with bi-partites, assigning a many-to-many relationship between clique identifies and users (Kumpula *et al.*, 2008).

To determine tie-strengths, profile information features can be compared, where friends that have more in common are deemed closer than friends that have less in common. This can be achieved by valuing each common feature as 1 (Misra, Jose M Such and Balogun, 2016).

To detect conflicts, every co-owner's policy can be compared to find co-owners that disagree with each other on target friends (Such and Criado, 2016). To determine item sensitivity, tie-strengths between users are used. The friendship with the highest tie-strength in a set of conflicts will have their resolution suggested.

Overall System

This section plans the implementation considering the Theoretical Development section.

Two ways that this system could be implemented are the client-server architecture or a peer-to-peer network. A peer-to-peer network offers better availability as it is decentralised and persistent storage could use distributed hash tables in a similar way to Bitcoin. The disadvantage of this architecture are that the client applications will be more complex and resource intensive, limiting the scope of users that can access the application. With the client-server model, 'thin' clients can be implemented as a lightweight application that submits tasks to much powerful backend computers. Persistent storage will contain personally identifiable information and should be destroyed at the end of the project. It is impossible to guarantee destruction in a peer-to-peer network whereas, in the client-server

model, we have authority over database replication thus can delete replicas as necessary. These points favour the client-server model which will be used.

A synchronous system in the client-server model can result in fluctuations of latency where a burst of requests to resource intensive operations will make the system respond slower to other requests and thus a Denial of Service (DoS) attack can be easily performed. By creating an asynchronous system, resource intensive tasks can be queued and performed at a constant rate without reducing the availability of other tasks for other users. We can also use bidirectional communication where the backend can send the client updates.

Frontend Client

The client could be implemented in arbitrary languages as a desktop or mobile application. To meet the accessibility requirement, the application should be available on as many devices as possible. JavaScript has gained massive popularity over recent years as most smartphones, tablets, desktops etc. have web browsers that can execute JavaScript based applications. This has allowed developers to build applications that are supported by a wide range of devices employing the *write-once, run-anywhere* slogan that Sun Microsystems pioneered with Java in the past.

Using frontend libraries and frameworks will help accelerate the development process and help conform to better programming practices. One of the most popular and feature filled frameworks is Angular, which uses TypeScript, a superset of JavaScript that adds static typing, classes and many other features of an object-oriented programming language. For the user interface, a User Interface framework will be used. Materialize CSS provides components that follow Google's Material Design standards as well as *card* components that can contain images which will be useful for displaying a grid of images. It also exposes a responsive grid for laying out components of a page which will be used for streamlining user experiences across different screen sizes (mobile to desktop).

The technology used for the frontend follows in Figure 1.

Technology	Homepage	Purpose
TypeScript	typescriptlang.org	Programming language
Angular 4	angular.io	MVC framework
Materialize CSS	materializecss.com	UI framework
Facebook SDK	developers.facebook.com/docs/javascript	Facebook API calls
TSLint	palantir.github.io/tslint/	Code linting

Figure 1 - Technologies used in Client

Backend

An asynchronous system needs bi-directional communication between the backend and clients. We can use the WebSocket protocol (Fette *et al.*, 2011) to achieve this. Whilst Python, PHP and Java dominate much of the current backend server software share, Go is a newer programming language maintained by Google that compiles into machine code with garbage collection. Its compilation makes it easy to place into minimal Docker containers.

For deploying services, Docker has taken the headlines in the dev-ops (developer operations) world as it allows containers to run with different configurations independently from each other on a host utilising Linux Containers (LXC).

Technology	Homepage	Purpose
Go	golang.org	Programming language
Docker	docker.com	Linux Container Engine
Terraform	terraform.io	Infrastructure as Code
Amazon Web Services	aws.amazon.com	Platform as a Service
Travis CI	travis-ci.com	Continuous Integration
golint	github.com/golang/lint	Code linting

Figure 2 - Technologies used in the Backend

Representational State Transfer (REST) API

The RESTful API will be the only publicly exposed endpoint of the backend as it will be easier to harden the system and monitor access from the internet. It will only interact with the cache so that it can maintain a low response time for all requests.

It will only implement the required REST endpoints, for example friends will be synced via a worker in the background, thus only GET for friends will need to be implemented.

Workers

Workers will collect resource intensive tasks from the queue service to execute. They will store results in the database and cache when necessary, also telling publish/subscribe channels to update user-sessions if appropriate.

Storage

Persistent storage will be used in the form of a relational database. Storing normalised data in a relational database allows for arbitrary queries to take place. A cache will be used to serve commonly used structures of data.

Persistent

The database will only store the minimum amount of information that can be used to derive other information. For example, instead of storing user attributes, only store the user ID as other attributes are not necessary. As mentioned in Algorithms, cliques will be stored in the form of bi-partites and it would not be necessary to store the whole friendship graph in the database.

Cache

The cache will be used to accelerate performance on common sets of information used by the REST API. It will store transient information such as the friendship graph and user profile information that is used during tie-strength calculation to save querying the Facebook Graph API excessively.

Redis is an in-memory data-structure store that offers O(1) complexity on many of its operations making it ideal for a cache on common operations. It also offers publish-subscribe capability, which will be used for updating user-sessions.

Queues

As mentioned in the Overall System, queues will be used to hold collections of tasks to be carried out by worker services. The payload of a task should only hold enough information to identify what entities need processing so that fresh information is used when the task is carried out.

RabbitMQ will be used as it offers traditional message broking which the API will use to offload more resource intensive processes to be completed asynchronously (Humphrey, 2017). It also offers official support for Go.

Implementation and Experimental Work

This section explains how the system was constructed and deployed.

Service Overview

Figure 3 shows the overall system architecture on Amazon Web Services (AWS). The Application Load Balancer, Cluster management(containers) and Managed Database are provided from AWS as Software as a Service (SaaS). A managed database was desired because it offers simple backup and replication. The Application Load balancer from AWS integrates with the Elastic Container Service (ECS) to expose services behind a load balancer.

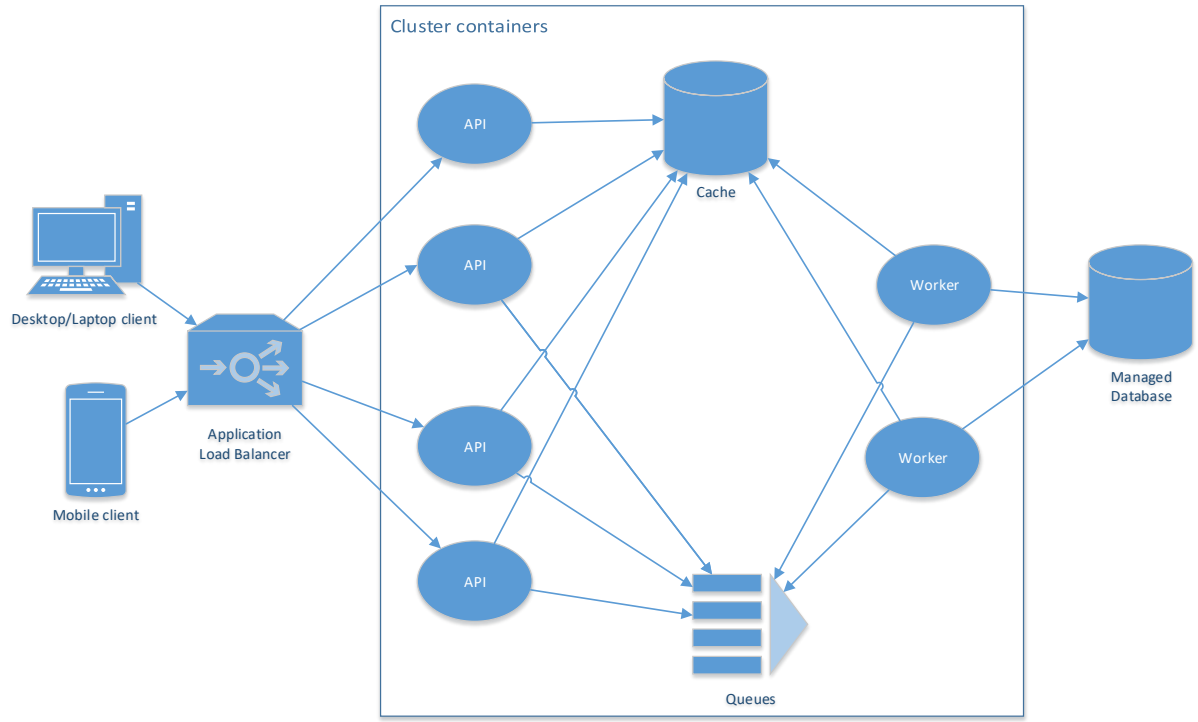

Figure 3 - System communication

The cluster consists of 2 m4.large (2 CPU cores – 8GB memory) EC2 spot instances. The m4 series of instances are preferred over the cheaper t2 series because they guarantee constant performance. The t2 instance class varies in performance based on past usage making them more suited for mostly idle configurations (Amazon Web Services, 2017). Spot instances are variably priced based on Amazon Web Services' spare capacity. These instances may be terminated and restarted in a different data centre at any time In Figure 3, the Cache and Queues also reside within this cluster as they store non-persistent data. They can also be configured to behave as a cache cluster and a queue cluster via Redis Sentinel (Sanfilippo, 2012) and RabbitMQ Highly Available Queues (Pivotal Software Inc., 2017).

Every service within the cluster and the clients communicate with JavaScript Object Notation (JSON) which is a lightweight data-interchange format (ECMA International, 2013).

Infrastructure and Deployment

This system has been deployed to Amazon Web Services (AWS) via Hashicorp's Terraform as infrastructure-as-code. Writing infrastructure-as-code allows us to pre-define complex infrastructures and spin them up or down on-demand. It also allows others to read how the

system works without searching through a control panel. The following Amazon Web Services have been used:

- Elastic Compute Cloud (EC2) – Virtual machines running Ubuntu Linux 16.04.
- Route 53 – Domain Name Service (DNS) for routing the domain names.
- Relational Database Service (RDS) – Managed relational database solution offering automatic backup and replication.
- Elastic Container Service (ECS) – Docker container management solution.
- Virtual Private Cloud (VPC) – Private virtual networks to isolate the application.
- Identity and Access Management (IAM) – User and service access.
- Certificate Manager – Transport Layer Security (TLS) certificate provision.
- CloudWatch Logs – Service output logging.
- Simple Storage Service (S3) – Object storage used for frontend web application and secure key storage.
- CloudFront Content Delivery Network (CDN) – Frontend deployment for high availability.

Travis CI has been used for continuous integration by running code style checks and tests before deploying to a test environment.

Frontend

The Angular 4 framework has been used to develop the frontend client. It is the simplest part of the system as it aims to be a 'thin-client' as mentioned in Overall System.

It has been deployed to Amazon Web Services' CloudFront CDN for high availability and it performs a small health check to verify that the backend services are reachable and coping with current load.

File type	Purpose
***.service.ts**	These interact with the backend to provide a repository that manages the lifecycle of entities on the client
***.model.ts**	These represent the entities used on the client
***.component.ts**	These are used to build a UI in Angular. There is one for each view, operating as a View/Controller in MVC.
***.module.ts**	These are used to configure a library

Figure 4 - Frontend file purposes

Promises

In the client, some object services provide a repository. These return an asynchronous guarantee for an object because a request to the backend server takes longer than accessing information already in memory. In JavaScript, this can be achieved using Promises (Mozilla Developer Network, 2017) and this works by programming the Promise object to first check a local in-memory structure for the existence of an object, resolving if it finds it otherwise querying the backend server for the object and resolving.

User Interface

This application has been developed using Materialize CSS which incorporates Google's Material Design standards into a user interface framework. Using this user interface framework aids in keeping a consistent style and experience throughout the application.

The navigation system allows a user to go to any of the main sections of the application in one click on desktop and 2 touches on mobile. The main sections identified are:

- Photos a user is tagged in.
- Cliques a user belongs to. This page also displays tie-strengths next to other members in a clique.
- Contexts a user has. This page also lists the global contexts.
- Submit Feedback for the application.

Figure 5 shows all the pages in the application and how a user interacts with the application to view them. Every page within the *Authenticated and Navigation System* container can navigate to Photos, Feedback, Friends and Contexts using the navigation system. This container also identifies pages which are only viewable once a user has been authenticated.

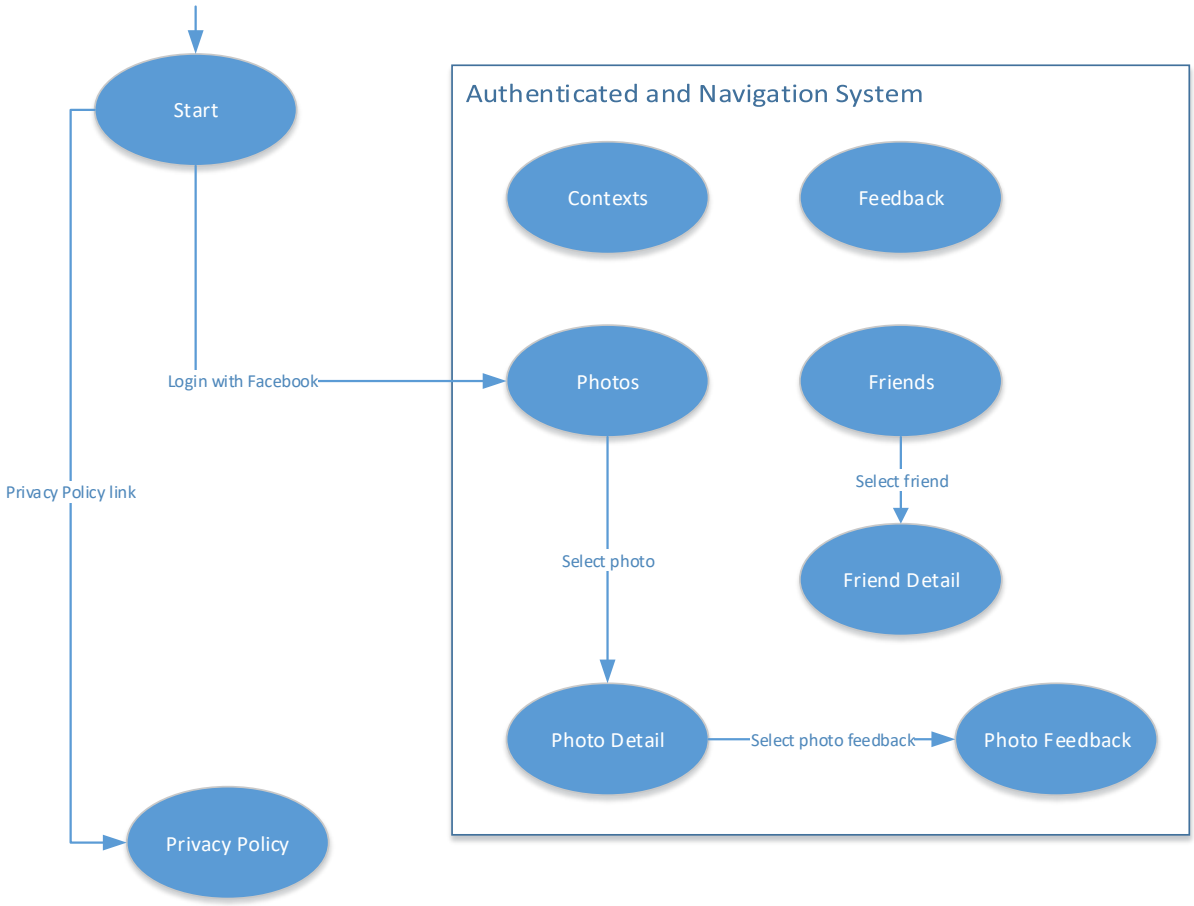

Figure 5 - Page state chart

The appendix provides screenshots of how the application looks under User Interface Screenshots.

Backend

All the following components are executed in separate Docker LXC (Linux Containers) as individual, scalable micro services.

The backend services share a lot of code thus have been developed in the same Go package. This package is built into a Docker container which is then started with different configurations across the cluster for the different API and worker services.

Go API

The API is started by setting the *TYPE* environment variable to 'API'. This exposes a HTTP server on port 80. The API is RESTful and implements the following endpoints in Figure 6.

Endpoint	Method	Purpose
/v1/auth	POST	Authenticate a short-lived access token and userID, returning a different authentication for use with the API.
/v1/categories	GET	Returns all categories including user-defined ones.
/v1/categories	POST	Creates a new user-defined category.
/v1/friends?ids=["x","y","z"]	GET	Returns user objects for given friend IDs where the session-user is a friend.
/v1/cliques	GET	Returns the cliques the session-user is a member of.
/v1/cliques/{id}	PUT	Updates the session-user's privacy preferences for a clique.
/v1/photos?ids=["x","y","z"]	GET	Returns photo objects for photos that exist where the session-user is tagged or the uploader.
/v1/photos	POST	Registers a photo with the application.
/v1/photos/{id}	PUT	Updates a photo's categories.
/v1/surveys	POST	Submitting surveys.
/v1/users?ids=["x","y","z"]	GET	Returns the request user IDs that exist on the system for determining if a photo is negotiable.
/v1/ws?authToken=x	GET	Upgrades the connection a WebSocket allowing the API to send updates for the session-user in real-time.

Figure 6 - Backend API Endpoints

Where a session-user is involved (apart from the websocket endpoint /v1/ws?authToken=x), the session-user is identified by the `Authorization` header sent with requests.

Go Workers

A worker is started by setting the *TYPE* environment variable to 'WORKER' and setting the *QUEUE* environment variable to a valid queue name.

Long Lived Authentication Token Worker

This worker is started by setting the *QUEUE* environment variable to 'auth-long-token'.

By default, Facebook returns short-lived access tokens that often only have a lifetime of two hours (Facebook Inc., 2017b). This worker obtains a long-lived access token from Facebook using the short-lived access token for use in further API calls. It also obtains and caches the IDs for each profile vector that is used in tie-strength determination.

This worker runs every time a user logs in to the application, obtaining a long-lived access token for that user and updating the cache with profile information. It then adds a message to the community detection queue for the logged in user's friendship groups to be updated.

Synchronise Photo Tags Worker

This worker is started by setting the *QUEUE* environment variable to 'photo-tags'.

The API exposes an endpoint for users to create photo objects on this application. Arbitrary information could be sent through this endpoint therefore the backend should obtain information about a photo from Facebook's Graph API itself as it cannot always trust the input from a client. The client sends the photo ID and Facebook user ID which this worker then uses to obtain information about the photo using the Facebook user's long-lived access token.

This worker runs whenever a new photo has been added.

Community Detection Worker

This worker is started by setting the *QUEUE* environment variable to 'community-detection'.

As mentioned in Algorithms, a new implementation of the Clique-Percolation Method is used for determining communities of users.

Algorithm

This algorithm performs the following steps for a given user:

1. Find and save new friends for a user by,
    a. Using the user's long-lived access token to query the Facebook Graph API for the user's friends.
    b. Iterates through all of the returned friends, checking if they already exist in the system and adding new bidirectional relationships for ones that do not exist.
2. Builds a local graph of only the user's friends and their friends.
3. Obtain the user's existing cliques and its members.
4. Find cliques by comparing the user's friends to each of their friend's friends using an array union operation. If the resulting array if longer than 3 items, a clique is found. If a clique is found,
    a. Verify that each member of the clique is in every other member's friends list.
    b. Verify that the clique is maximal by performing an array union on the members of the clique's friends.
    c. Compare the new clique to existing cliques

i. If an existing subset clique exists, migrate the existing clique to the newly found one.
ii. If an identical clique exists, ignore the new one.
iii. Otherwise, form a new clique.

Pseudocode

Algorithm 1 Community Detection

```
1:  function CommunityDetectionForUser(u)
2:      E ← getExistingFriendsForUser(u)
3:      F ← getAllFriendsFromGraphAPIForUser(u)
4:      for all f ∈ F do
5:          if f ∉ E then
6:              E ← E ∪ f
7:      U ← E ∪ u
8:      for all e ∈ E do
9:          d ← getExistingFriendsForUser(e)
10:         G[e] ← d ∪ u
11:     C ← getExistingCliquesForUser(u)
12:     for all e ∈ E do
13:         M = U ∩ G[e]
14:         if |M| >= 3 then
15:             if isValidClique(M) ∧ isMaximalClique(M) then
16:                 for all c ∈ C do
17:                     if isSubset(c, M) ∧ |c| = |M| then
18:                         break
19:                     else if isSubset(c, M) ∧ |c| < |M| then migrateOld-
    CliqueToNewClique(c, M)
20: function isMaximalClique(N, F, G)
21:     for all n ∈ N do
22:         F ← F ∩ G[n]
23:     if |F| = |N| then return true
        return false
24: function isValidClique(N, G)
25:     for all n ∈ N do
26:         for all m ∈ N do
27:             if n = m then
28:                 break
29:             if n ∉ G[m] then return false
        return true
30: function isSubset(A, B)
31:     for all a ∈ A do
32:         if a ∉ B then return false
        return true
```

Figure 7 - Community Detection pseudocode

Figure 7 shows the pseudocode for the implemented community detection algorithm. Lines 2-6 are used for updating the locally cached graph of user friendships. U represents a subgraph of the users involved in this round of community detection by appending the current user, u, to the set of their friends, E. Lines 8-10 build a map (2-dimensional set) of user's friends that represent the graph of user friendships. M is the result of finding mutual friends using set unions as described in step 4 of the Algorithm.

Tie Strength Worker

This worker is started by setting the *QUEUE* environment variable to 'tie-strength'. This worker gets ran for every friend found during the Community Detection Worker.

In Algorithms, the tie-strength worker was to be implemented similarly to Misra, Such and Balogun's work. During this implementation, it became apparent that many permissions would need to be requested from users which significantly reduces the amount of completed logins as users perceive the app to be more intrusive (Facebook Inc., 2017d). Some attributes such as similar groups are unable to be read from the Facebook Graph API entirely. The closest match is the *user_managed_groups* permission which gives read/write access to groups that a user manages, but no access to see which groups a user is a member of (Facebook Inc., 2017e).

As well as the public permissions, 9 of the 11 permissions requested for the application are used for obtaining direct attributes for profile similarity. This takes into account the following 17 profile attributes:

- Gender
- Age group
- Family
- Hometown
- Current Location
- Educational Institutions
- Events
- Favourite Teams
- Inspirational People
- Languages
- Sports
- Work
- Music
- Movies
- Likes
- Political Alignment
- Religion

All similar attributes are valued as 1, apart from Family which is valued at 500 considering that families are usually much closer. To determine better values for each attribute, the survey asks users to weigh how important profile attributes are in their friendships.

Conflict Detection and Resolution Worker

This worker is started by setting the *QUEUE* environment variable to 'conflict-detection-and-resolution'.

This algorithm is based off Such and Criado's work as mentioned in Algorithms and split into 3 stages:

1. Build maps of tagged user allowed and blocked users by,
    a. Iterating through the tagged user's cliques and comparing their policy to the photo's contexts. This considers that a friend can exist in multiple of their cliques, thus may not be granted access in one clique but granted access in another.
2. As in Such and Criado's work, compare each tagged user's policy to other tagged users looking for conflicting policies on each friend.
3. If conflicts are found, suggest a resolution by using tagged-user tie-strengths as voting weights. Where tagged-users who want to BLOCK the friend have their tie-strengths subtracted from the tagged-users who want to ALLOW the friend. The mediator suggests that the friend is allowed if the result is positive and blocked if the result is negative.

Pseudocode

Algorithm 1 Conflict Detection and Resolution

```
    function ConflictDetectionForPhoto(p)
2:      T ← getTaggedUsersForPhoto(p)
        R ← getCategoriesForPhoto(p)
4:      for all t ∈ T do
            C[t] ← getExistingCliquesForUser(t)
6:          for all c ∈ C[t] do
                D ← getUserCategoriesForClique(t, c)
8:              for all d ∈ D do
                    if d ∈ R then
10:                     A[t] ← A ∪ c
                    else
12:                     B[t] ← B ∪ c
        for all t ∈ T do
14:         for all a ∈ A[t] do
                for all s ∈ T do
16:                 if a ∈ B[s] then
                        Y ← Y ∪ NewConflict(t, s, a)
         return Z
18: function NewConflict(t,s,a)
        y ← getTieStrength(t, a)
20:     z ← getTieStrength(s, a)
        x = y − z return (t, s, a, x)
```

Figure 8 - Conflict detection and resolution pseudocode

Figure 8 shows the most important pseudocode for how conflicts are detected and resolved. A represents a map of users allowed by tagged users in the photo and B represents a map of users blocked by users tagged in the photo. Lines 13-16 find conflicts by comparing all user's policies for the photo. Line 21, shows the recommendation being determined by the differences in tie-strength.

Persist Workers

There are 4 worker services and queues that are dedicated to simply persisting different entities to the database on creation. These are listed in Figure 9.

Worker Service	Name	Purpose
Category	persist-category	Add new Categories to the database
User-Clique	persist-user-clique	Update user privacy policy for a Clique they belong to
Photo	persist-photo	Add new photos to the database
Survey	persist-survey	Add new completed surveys to the database

Figure 9 - Persist workers

Data Storage and Caching

As per the Storage section, this system uses the Postgres Relational Database Management System (RDBMS) with a Redis cache. The Postgres RDBMS is provided by AWS RDS for easy backup and replication, whereas the Redis cache is part of the container cluster.

Entity Relationship

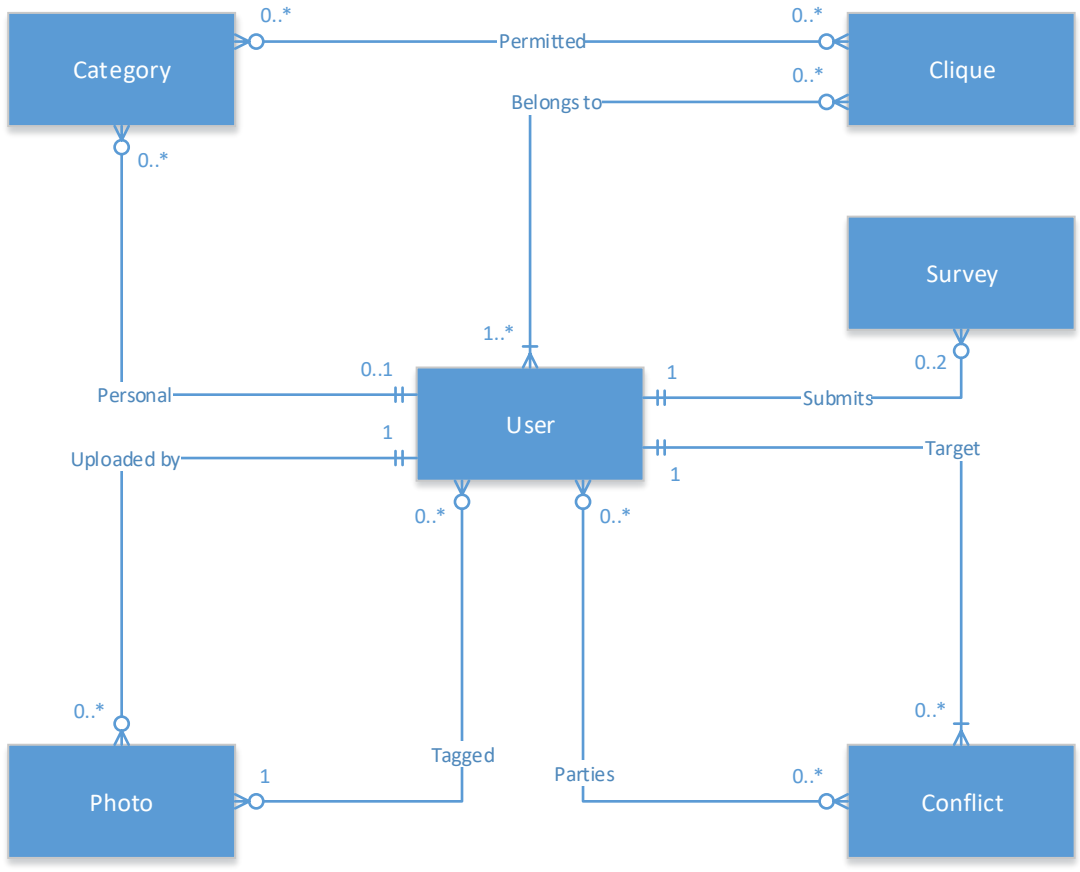

Figure 10 - Entity Relationship

Database Schema

The database schema consists of 11 tables including join tables and normalised entities.

```
categories (name, user_id)
cliques (id)
conflict_parties (conflict_id, user_id)
conflicts (id, photo_id, target_id, result)
photo_categories (photo_id, category_name, category_user_id)
photo_users (photo_id, user_id)
photos (id, uploader)
surveys (user_id, photo_id, raw_json)
user_clique_categories (category_name, category_user_id,
user_clique_clique_id, user_clique_user_id)
user_cliques (clique_id, user_id, name)
users (id, long_lived_token, token_expires)
```

Figure 11 - Relational Database Schema

Figure 11 shows the relational schema for the database. Tables names are in **bold**, primary keys are underlined with some tables having composite primary keys. For surveys, a go script (backend/survey-converter/main.go) has been written to convert a CSV (Comma Separated Values) file from the database into further CSV sheets after parsing the raw JSON so that feedback can be analysed. This was done so that the feedback survey could be changed without losing previous responses.

Cache Schema

Redis is an in-memory key-value data store, which is being used a cache. Caches tend to have very fast and simple query operations, but slower and more complex insert and update operations as many items will need to be updated in its de-normalised structure.

Query/Purpose	Key	Type
Global categories	categories	Set<String>
User-defined categories	u<userID>:categories	Set<String>
Friendship graph	u<userID>:friends	Hash<friendID, Object of relationship>
Cliques	u<userID>:cliques	Hash<cliqueID, Object of user clique>
Photo information	p<photoID>:info	Object of photo information
Photo categories	p<photoID>:categories	Set<String>
Photo user categories	p<photoID>:u<userID>	Set<String>
User information	u<userID>:info	Object of user information
User profile vectors	u<userID>:profile	Object of user profile vectors
User Pub/Sub	user:<userID>	Channel

Figure 12 - Cache structure

In Figure 12, the relationship object contains the tie-strength between a user and their friend. This is stored twice from user $A \rightarrow$ user B and user $B \rightarrow$ user A as friendships are bidirectional. The user clique object caches information about a user's privacy preferences for a clique. The photo information object caches information about whom is tagged and the who uploaded the photo. The user information object caches the expiry time for the long-lived access token. The user profile vectors object stores information about a user's

profile to be used in tie-strength calculation. The User Pub/Sub is used to update user clients. For example, every session will connect to their user's pub/sub channel, resulting in a unified experience as updates are emitted to every session where the user is logged in.

Security

This section covers the security considerations of this application as it handles personal information.

Authentication

For authentication of a client to the backend API:

1. Client performs Facebook login using JavaScript SDK to obtain a short-lived access token and the Facebook user ID.
2. Client sends POST request to /v1/auth on the backend API containing short-lived access token and Facebook user ID.
3. Backend API queries the graph API using the short-lived access token to find out what Facebook user ID it belongs to and compares it to ID that the client sent.
4. If the IDs match, return a new JSON Web Token that contains the Facebook User ID. This will be used by the client in further requests so the API know which Facebook User ID sent a request. Otherwise, return the Unauthorised status.

JSON Web Tokens (JWTs) are symmetrically encrypted strings with some extra meta-data (i.e. valid-from and expiry). In this case, the encrypted portion contains the Facebook User ID. If the symmetric key used by the backend is exposed, anyone could create arbitrary access tokens to masquerade as different users on the backend API. From here they could only affect this application's integrity e.g. changing the categories on a photo or changing the privacy preferences for a clique. This application doesn't write to the Facebook graph so information on Facebook won't be compromised, and the application only uses Facebook IDs so a user would still not be able to see photos themselves as the clients request further information directly from the Graph API which is authenticated separately using the Facebook JavaScript SDK. The symmetric key for this is parameterised to the application, making it easy to change.

Confidentiality

This application holds information about its users in the storage services (database, cache, queues). To protect user information, these services are contained within a private network alongside the backend workers and API so that they can communicate with each other. This private network is firewalled from the public network (the internet), only allowing the load balancer to communicate with the backend API services. The load balancer is the only component that is publicly accessible and all inbound requests are routed through it.

If a researcher wants to obtain information from the database such as completed surveys, a snapshot of the production database is created, then launched in a separate network that is firewalled to only allow access from the researcher's IP address. The researcher then connects to the database using TLS encryption on their preferred database client and can browse or download the snapshotted database. This procedure is recommended to prevent accidental changes to the real production database.

Privacy Policy

The privacy policy lets users know what and how information of theirs is used in order to provide the services exhibited by this application. It also states how data is secured in transit and that it will all be destroyed at the end of the project lifespan. See the full text in the Appendix under Privacy Policy.

Facebook Graph API

Facebook exposes an API called the Graph API which allows third-party applications, like this one, to query for certain information about users. This application relies on the Graph API and shares the same limitations that it exposes.

It is impossible present a fine-grained audience view to users as there is no field to read a photo's current privacy setting (Facebook Inc., 2017c). It is possible to obtain a limited privacy setting ("everyone", "all-friends", etc.) for an Album, but this cannot be updated via the Graph API. It's possible to restrict a photo's privacy setting further via the Graph API, so applying a resolution is possible. However, this has not been implemented because it will result in photos being too restrictive as it would only allow users who also use the application to see the photo.

When querying for a user's friends, the Graph API only shows friends that also use the application. This hinders community detection and restricts how useful the tool is when not all stakeholders use it. For example, If Alice does not want Bob to see a photo, but Bob does not use the application, she would not be able to define a privacy policy stating her preference as the application is not aware of Bob.

When retrieving tagged photos for a user, the Graph API will only return publicly viewable photos or photos from friends that also use the application.

Facebook Approval

For an application to be available publicly for any Facebook user to use, it first must progress through Facebook's App Review system. In the system, a screencast, usage steps and description of use must be submitted for each requested permission. The first 4 submissions were unsuccessful due to little feedback, however upon communicating with their team, the application was successfully approved after end-users could view how the tie-strength was calculated, which showed how the permissions for user profile were used. Figure 13 shows the permissions requested and approved.

Permission	Purpose
user_friends	See user's friends that also use the application to generate cliques
user_photos	See user's photos to present them and store contexts
user_posts	See user's photos that have been published as a post
user_education_history	See user's education history for tie-strength generation
user_hometown	See user's hometown for tie-strength generation
user_location	See user's current location for tie-strength generation
user_relationships	See user's relationships for tie-strength generation

user_religion_politics	See user's religion and political alignment for tie-strength generation
user_work_history	See user's work history for tie-strength generation
user_events	See user's events for tie-strength generation

Figure 13 - Facebook permissions requested

Recruitment

After the application was approved by Facebook, a post was published to encourage friends and family to use the application and share further. The initial post received 8 shares and 18 likes, however more people liked the post than used the application.

41 friends were directly messaged 4 weeks later which received a far better response rate. It also highlighted that some friends were not comfortable approving the requested Facebook permissions.

Facebook Promotions was also used with a budget of £40 over 4 days. It reached 6,165 users with 64 ad-clicks redirecting to the application.

Results

Over the course of 5 weeks, the application became aware of 948 people tagged across 2809 photos from 57 users logging into the application. 35 cliques were formed between those users who also assigned 19 categories to cliques they belong to. 23 categories (with 2 user-defined categories) were assigned to photos, resulting in 3 conflicts being detected. 33 pieces of submitted feedback were received.

Of the received surveys, all respondents were over the age of 21. 16(48.48%) of the respondents were female, with the remaining 17(51.52%) being male.

Feedback Result Summaries

Question 1

Has this tool made you more aware of things that undesirably breach your privacy?	
No	6 (18.2%)
Yes, I had no idea	9 (27.3%)
Yes, I had some awareness but didn't realise the extent.	18 (54.5%)

Figure 14 - Question 1 results

Question 2

What is your current sharing preference on Facebook?	
Everyone	6 (18.2%)
Friends only	24 (72.7%)
Only me	3 (9.1%)
I don't know	0

Figure 15 - Question 2 results

Question 3

How do you usually resolve conflicts with photos that you upload or are tagged in?	
I would untag myself/others	27
I would crop the photo	0
I would remove the photo entirely	11
I would blur the photo	0
I would do nothing	3

Figure 16 - Question 3 results

There were 3 "other" selections that detailed further steps before untagging or removing.

Question 4

Do you think the generated similarities reflect your relationships accurately?	
No	5 (15.2%)
Yes	28 (84.8%)

Figure 17 - Question 4 results

The 5 "No" selections brought the following points:

- Interacted with people via other social platforms.
- Not enough friends using the application to check accuracy.
- Likes are not up to date between parties and not fully representative.

Question 5

Do you think your Facebook profile represents yourself and your relationships accurately?	
No	11 (33.3%)
Yes	22 (66.7%)

Figure 18 - Question 5 results

The 11 "No" selections brought the following points:

- Tagged posts/photos do not necessarily represent thoughts and feelings about certain topics.
- Interact with people more via other social platforms.
- More careful with Facebook to manage self-representation.
- Not updated, stale information.

Question 6

How important do you think these are for representing your relationship with someone? (totals)	
Politics	91
Religion	56
Work	85
Sports	81
Family member	120
Location	117
Education	113
Favourite Teams	66
Inspirational People	73
Languages	85
Music	100
Movies	89
Likes	96
Groups	93
Events	112

Figure 19 - Question 6 total results

Splitting the results by gender, gave the following averages in Figure 20.

How important do you think these are for representing your relationship with someone? (average by gender)		
Attribute	**Male (avg.)**	**Female (avg.)**
Politics	2.65	2.88
Religion	1.76	1.63
Work	2.71	2.44
Sports	3.18	1.69
Family member	3.59	3.69
Location	3.47	3.63
Education	3.41	3.44
Favourite Teams	2.41	1.56

Inspirational People	2.18	2.25
Languages	2.47	2.69
Music	3.29	2.75
Movies	2.82	2.56
Likes	2.76	3.06
Groups	3.00	2.63
Events	3.53	3.25

Figure 20 - Question 6 results by gender (mean average)

Question 7

10 people submitted feedback on this question. They suggested the following improvements:

- Not entirely clear on how to use the application, which could be resolved with further descriptions of what parts mean and/or a tutorial.
- Manually editable cliques or individual users.
- Have the ability to filter by only publicly viewable photos.
- Inform the user who can currently see photos.
- The interface is friendly, but ugly.

Question 8

7 people submitted feedback on this question. They were all positive and made the following points:

- Can see the tool being really useful if more people used it.
- Accessibility is easy as there is no need to download and install.
- It's good for keeping track of your online presence and hiding things from future employers.
- It's great for awareness.

Other

Feedback via other means from people who did not use the application included:

- They were uncomfortable sharing their information.
- The application looked illegitimate.

Performance

Community detection takes ~800ms for a user with 40 friends.

The average latency for the API was 6ms. The API handled up to 10,000 requests in one day without any noticeable change in resource use.

AWS CloudFront handled 6,571 requests for the frontend. The devices these came from are presented in Figure 21.

Device type	Number of requests
Desktop	2865 (43.6%)
Mobile	2798 (42.58%)
Bot/Crawler	431 (6.56%)

Tablet	136 (2.07%)
Unknown	341 (5.19%)

Figure 21 – AWS CloudFront requests by device

User Interface

Using the Goals, Operators, Methods and Selections (GOMS) rules we can predict how long different tasks will take a user (John and Kieras, 1996).

Action	Time (seconds)
(K) Keying or clicking a mouse button	0.2
(P) Pointing	1.1
(H) Homing between keyboard and mouse	0.4
(M) Mentally preparing	1.35
(R) Response from application	0.25 (maximum)

Figure 22 - GOMS actions

The response from the application has a maximum of 0.25 seconds to avoid the user becoming uneasy. The following table exhibits all user tasks and their time taken using GOMS from Figure 22.

ID	Task	Actions (Desktop)	Time (seconds)
<1>	Log in	H PK	1.7
<2>	View tagged photos	<1> R	1.95
<3>	View photo	<2> M PK	4.6
<4>	Add context to photo	<3> PK M PK PK	9.85
<5>	View friends or cliques	<1> PK	3
<6>	Set policy for clique	<5> PK M PK PK	8.25
<7>	View contexts	<1> PR PK	4.35
<8>	Add context	<7> PK MK PK	8.5

Figure 23 - User task efficiency

Observation and Discussion

From Question 1 (Figure 14), of the 33 respondents, 27 (81.8%) found the tool showed them photos that breached their privacy. This highlights the need for tools like the one implemented in this project. They would be better integrated as part of social networks instead of external tools to avoid obstacles created from granting login permissions (and further trust); exposed API limitations; and keeping a consistent style. It might, however, be more desirable to have one external tool that can manage privacy across multiple social media sites, requiring exposed APIs to be compliant.

Question 2 (Figure 15) shows that the majority of people have their privacy setting set to show posts to their friends or the public. This results in people being far more careful about managing their self-representation manually as they may limit what they share and/or not approve requests from colleagues or parents, which further hinders sharing.

Question 3 (Figure 16) shows use of the currently available options people have in sharing. Untagging results in the photo not being hidden from mutual friends and removing the photo reduces sharing.

Question 4 (Figure 17) shows that 84.8% of respondents felt that the tie-strengths generated represented their relationships with friends accurately. This supports Misra, Such and Balogun's research by showing how effective similarity based access control is. The primary downfalls of this method were down to insufficient or stale profile information.

Question 5 supports some of the downfalls mentioned from Question 4, where a third of respondents felt that their Facebook profile was not representative of themselves due to insufficient or stale profile information. They also felt that they would manage their self-representation more on Facebook which presents the need for tools that can be aware of self-representation goals to audiences.

Question 6 presents the collated totals for each similarity vector. In general, this indicates that respondents rated family relationship, mutual location, educational institutions, events and music taste as the most definitive attributes to their friendships. However, this does not accommodate for every user's preference as there are some respondents that placed religion and work among their most important attributes. Given a different group of users, we may find that different profile attribute similarities are preferred in a friendship. It would be interesting to see what influences these preferences in order to predict tie-strengths more effectively. For example, Figure 20 shows that males, on average, valued sports and favourite teams in relationships more than females did. People with more similar profiles may share preferences to what attributes are most important to them.

Responses to Question 7 highlighted that the interface needed more polishing touches to guide a user on how to use the application. Unfortunately, the Facebook Graph API currently does not allow for showing what users can currently see a photo and retrieving the current privacy policy for photos. The request for manually editable cliques or at least individual user policies support that community detection should be suggested instead of forced to allow more flexible privacy policy definition.

Community detection took a long time with only 40 friends and will take exponentially longer with more friends. This is inevitable as community detection translates into a maximal clique problem which is NP-hard (Pardalos and Xue, 1994). This should continue to be done asynchronously, but further optimisation would be welcome, perhaps by examining existing cliques to discard parts of the graph.

Using queues to offload asynchronous work resulted in the API consistently having a low response time with minimal resource usage. The servers never showed any signs of higher resource usage showing that this application could handle much more load and cheaply scale to accommodate Denial of Service attacks.

It's important to note that in Figure 21, mobile devices had almost the same amount of requests as desktops/laptops. This shows how important it is for a tool like this to be accessible on mobile devices and the use of a responsive CSS framework mentioned in Frontend Client was valuable.

Whilst the User Interface evaluation showed that it was relatively quick and easy to perform tasks. A first time user would need to spend ~8.5 seconds per photo resulting in the average user for this application spending almost 7 minutes on just photos alone plus an additional ~7 seconds per clique policy. This shows that users have to invest a lot of time in this tool at the start and will likely stop using it. However, after the first-use, there will be significantly less photos and cliques to manage, therefore lowering the required time. Some feedback also showed that the styling of the application was ugly making it appear illegitimate. This could be because they expected a similar style to Facebook's website but integration into an existing social network would resolve these issues.

Conclusion

Social media's booming popularity is evident with over 1.3 billion active daily users on Facebook (Facebook Inc., 2017a). All types of media depicting many users are uploaded to social networks but these services only present privacy management options that favour the uploader. This leaves co-owners of media limited options that either reduce sharing (delete the photo) or don't respect their privacy completely (untag from photo).

This project has demonstrated that there is a need for collective privacy management and that profile similarities can be used in mediation. A problem that some profiles are not maintained or representative of users lives hinders how affective the algorithm is.

Collective privacy management has a long way to go before it becomes wholly mainstream, but this project could be used a starting point for future implementations as it exposes what is required, and further challenges, in building a tool to collectively manage privacy.

Future and Development

This tool would greatly benefit from integration into the social network directly. This would allow the tool to take advantage of using Facebook's infrastructure, bypassing the limitations of the Graph API, eliminating the limitations described in Facebook Graph API.

As mentioned in the Observation and Discussion, community detection would be better implemented as an aid upon user managed groups. A user could press a button that will suggest communities for friends that aren't currently in a group which can then be to applied or rejected.
Another enhancement would consider a user not minding who sees a photo they are tagged in. This could be implemented by having a special category for photos that exclude a user's usual privacy policy. This may be sufficient to meet Such and Criado's I Don't Mind and I Understand rules presented in their conflict resolution strategy (Such and Criado, 2016).

Further work could be done to implement further alternatives such as cropping or blurring pictures to blocked users in conflict. This would rely on tagged users being marked on the photo correctly as the Graph API does expose the coordinates of a tag.

The frontend client could be improved by incorporating a service worker (Gaunt, 2017) to deliver real-time notifications of conflicts and optionally resolutions. Local storage could also be used to cache user information from the Graph API, saving the need to re-make all of the requests again whenever a user revisits the application.

Another challenge is determining a context or scenario from a photo automatically. This tool currently asks users to define them manually, but future work could look at predicting what contexts or categories a user would apply to a new photo to reduce their burden further.

A tutorial could be implemented to train users on how to use the tool to help them manage their privacy more effectively.

Further security practices should also be considered. For example, JSON Web Tokens should expire after a short amount of time, requiring clients to re-authenticate automatically.

Bibliography

Amazon Web Services (2017) *Amazon EC2 Instance Types*. Available at: https://aws.amazon.com/ec2/instance-types/#burst.

ECMA International (2013) *The JSON Data Interchange Format*. Available at: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.

Evans, E. (2003) *Domain-driven Design: Tackling Complexity in the Heart of Software*. Addison Wesley.

Facebook Inc. (2017a) *Company Info*. Available at: https://newsroom.fb.com/company-info/.

Facebook Inc. (2017b) *Expiration and Extension of Access Tokens*. Available at: https://developers.facebook.com/docs/facebook-login/access-tokens/expiration-and-extension.

Facebook Inc. (2017c) *Graph API Photo Node*. Available at: https://developers.facebook.com/docs/graph-api/reference/photo/.

Facebook Inc. (2017d) *Optimizing Permissions Requests*. Available at: https://developers.facebook.com/docs/facebook-login/permissions/requesting-and-revoking.

Facebook Inc. (2017e) *Reference - Facebook Login*. Available at: https://developers.facebook.com/docs/facebook-login/permissions/#reference-user_managed_groups.

Fette, I. *et al.* (2011) 'The WebSocket Protocol'. Internet Engineering Task Force (IETF). Available at: https://tools.ietf.org/html/rfc6455.

Fogues, R. *et al.* (2015) 'Argumentation for Multi-party Privacy Management', *The Second International Workshop on Agents and CyberSecurity (ACySe)*, pp. 3–6.

Fowler, M. (2004) *Inversion of Control Containers and the Dependency Injection pattern*. Available at: https://martinfowler.com/articles/injection.html.

Gaunt, M. (2017) *Service Workers: an Introduction*. Available at: https://developers.google.com/web/fundamentals/getting-started/primers/service-workers.

Humphrey, P. (2017) *Understanding When to use RabbitMQ or Apache Kafka*.

Johansson, L. (2014) *What is message queueing?* Available at: https://www.cloudamqp.com/blog/2014-12-03-what-is-message-queuing.html.

John, B. E. and Kieras, D. E. (1996) 'The GOMS family of user interface analysis techniques: comparison and contrast', *ACM Trans. Comput.-Hum. Interact.*, 3(4), pp. 320–351. doi: 10.1145/235833.236054.

Krasner, G. E. and Pope, S. T. (1988) 'A Description of the Model View Controller User Interface Paradigm in the Smalltalk 80 System', *The Center for Research in Electronic Art Technology*, (October).

Kumpula, J. M. *et al.* (2008) 'Sequential algorithm for fast clique percolation', *Physical Review E*, 78(2), p. 26109. doi: 10.1103/PhysRevE.78.026109.

Misra, G. and Such, J. M. (2016) 'How Socially Aware are Social Media Privacy Controls?', *IEEE Computer*, 49(3), pp. 96–99.

Misra, G., Such, J. M. and Balogun, H. (2016) 'IMPROVE - Identifying Minimal PROfile VEctors for Similarity Based Access Control.', *Trustcom/BigDataSE/ISPA*, pp. 868–875. doi: 10.1109/TrustCom.2016.149.

Misra, G., Such, J. M. and Balogun, H. (2016) 'Non-sharing communities?: an empirical study of community detection for access control decisions', *Proceedings of ASONAM 2016*. doi: 10.1109/ASONAM.2016.7752212.

Mozilla Developer Network (2017) *Promise*. Available at: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise.

Pardalos, P. M. and Xue, J. (1994) 'The Maximum Clique Problem (MCP)', *Journal of global Optimization 4.3*, pp. 301–328.

Pivotal Software Inc. (2017) 'Highly Available (Mirrored) Queues'. Available at: https://www.rabbitmq.com/ha.html.

Radigan, D. and Altassian (2017) *Kanban*. Available at: https://www.atlassian.com/agile/kanban.

Sanfilippo, S. (2012) *Redis Sentinel Documentation*. Available at: https://redis.io/topics/sentinel.

Such, J. M. *et al.* (2017) 'Photo Privacy Conflicts in Social Media: A Large-scale Empirical Study', *ACM Conference on Human Factors in Computing Systems (CHI)*. Available at: https://kclpure.kcl.ac.uk/portal/en/publications/photo-privacy-conflicts-in-social-media-a-largescale-empirical-study(c5e9250b-c4ba-4a6a-9d38-33ae018aa1df)/export.html.

Such, J. M. and Criado, N. (2016) 'Resolving Multi-Party Privacy Conflicts in Social Media', *IEEE Transactions on Knowledge and Data Engineering*, 28(7), pp. 1851–1863. doi: 10.1109/TKDE.2016.2539165.

Appendices

General Survey Questions

1. At least every greyscale photo presented by this tool is publicly searchable and viewable by everyone. Has this tool made you more aware of things that undesirably breach your privacy? (i.e. A photo is publicly viewable that you weren't aware of) [Select one]
    a. Yes, I had no idea.
    b. Yes, I had some awareness but didn't realise the extent.
    c. No, I was already aware.
2. What is your current sharing preference on Facebook? [Select one]
    a. Everyone.
    b. Friends only.
    c. Only me.
    d. I don't know.
3. How do you usually resolve conflicts with photos that you upload or are tagged in? [Select multiple]
    a. I would untag myself/others.
    b. I would crop the photo.
    c. I would remove the photo entirely.
    d. I would blur the photo.
    e. I would do nothing.
    f. Other? [Text entry]
4. Do you think that the generated similarities section reflects your friendships accurately? [Select one]
    a. Yes.
    b. No. Why? [Text entry]
5. Do you think that your Facebook profile represents yourself and your relationships accurately? [Select one]
    a. Yes.
    b. No. Why? [Text entry]
6. How important do you think these are for representing your relationship with someone? (how close you are to someone). 1 is unimportant, 5 is very important.
    a. Politics [Select one between 1-5]
    b. Religion [Select one between 1-5]
    c. Work [Select one between 1-5]
    d. Sports [Select one between 1-5]
    e. Family member [Select one between 1-5]
    f. Location [Select one between 1-5]
    g. Education [Select one between 1-5]
    h. Favourite Teams [Select one between 1-5]
    i. Inspirational People [Select one between 1-5]
    j. Languages [Select one between 1-5]
    k. Music [Select one between 1-5]

   l. Movies [Select one between 1-5]
   m. Likes [Select one between 1-5]
   n. Groups [Select one between 1-5]
   o. Events [Select one between 1-5]
7. How could this tool be improved? [Free text]
8. Any further comments? [Free text]

Conflict Survey Questions

1. Were you asked for your permission before the photo was uploaded?
   a. Yes.
   b. No, but I don't mind.
   c. No, and I do mind.
2. Do you agree with the recommendation?
   a. Yes.
   b. No.
3. Do you understand the motives of opposing parties in the conflict?
   a. Yes, but I would not concede my preference.
   b. Yes, and I would concede my preference.
   c. No, but I may concede knowing their reasons.
   d. No, and I would not concede.

Privacy Policy

This tool utilises the following information from Facebook to generate relationship strengths by profile similarities:

- Education history
- Hometown
- Likes
- Location
- Relationship details
- Relationships
- Religion
- Politics
- Work history
- Events
- Groups

This information is stored securely in a cache and is only accessed when calculating relationship strengths.

This tool also utilises the following information from Facebook to present photos and perform community detection:

- Your tagged photos
- Your posts
- Your friends list

IDs, tagged users and the uploader are the only properties sought and are stored securely in a cache and database. This information is only accessed when generating communities and performing conflict-detection.

This tool also utilises the following information from Facebook to gather feedback on the application

- Facebook ID
- Facebook Photo ID

This is only used to prevent spam. You can only submit feedback for a unique photo once or the overall tool once.

Securing Data

This tool uses TLS to secure data sent over HTTP and Web Sockets.

Non-identifiable information (feedback) will be used to evaluate this tool. Afterwards, all information collected will be destroyed at the end of this project life in September 2017. Only non-identifiable information will be used for research in this project's scope

User Interface Screenshots

Mobile (1080 x 1920, 401 DPI)

Figure 24 - Mobile Index page

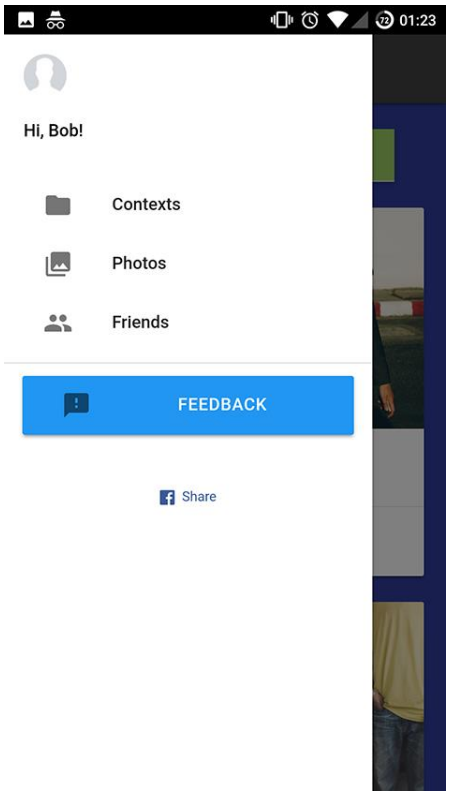

Figure 25 - Mobile Navigation

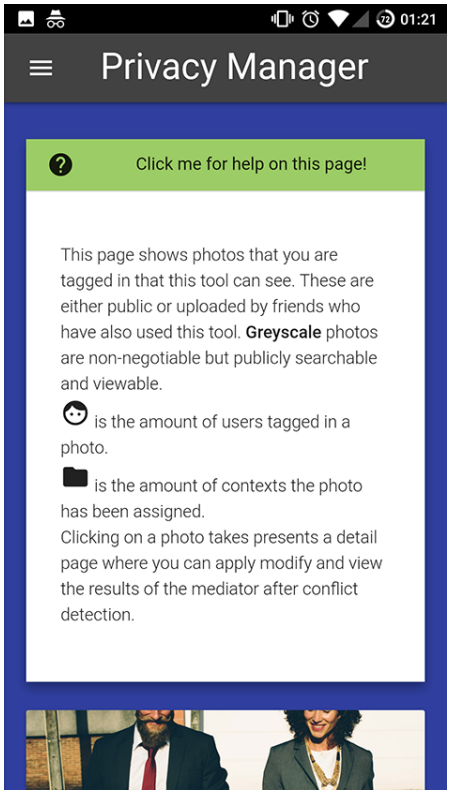

Figure 26 - Mobile Photos page 1

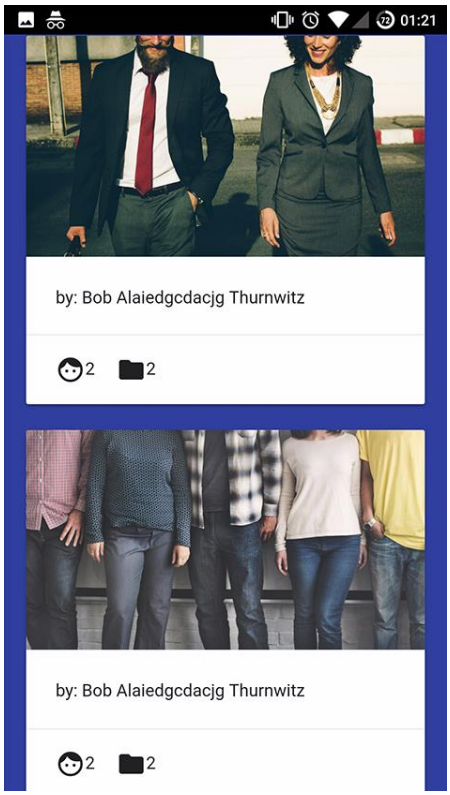

Figure 27 - Mobile Photos page 2

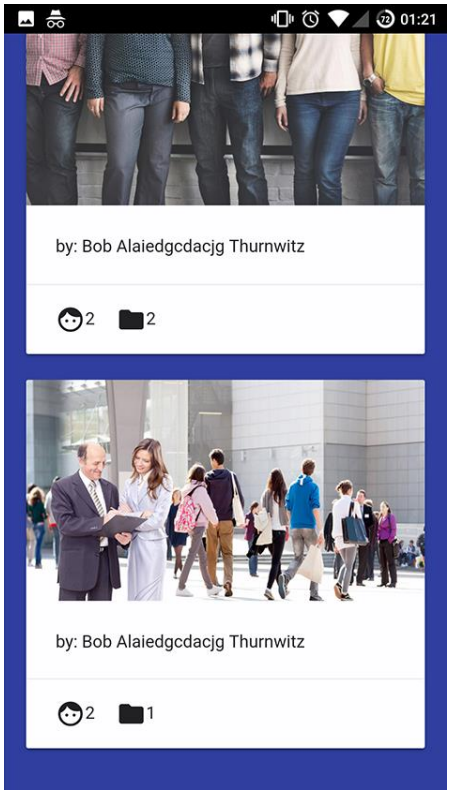

Figure 28 - Mobile Photos page 3

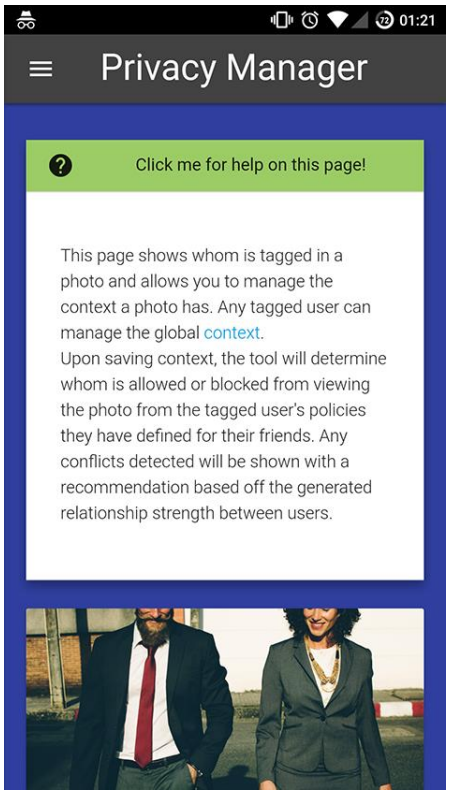

Figure 29 - Mobile Photo Detail page 1

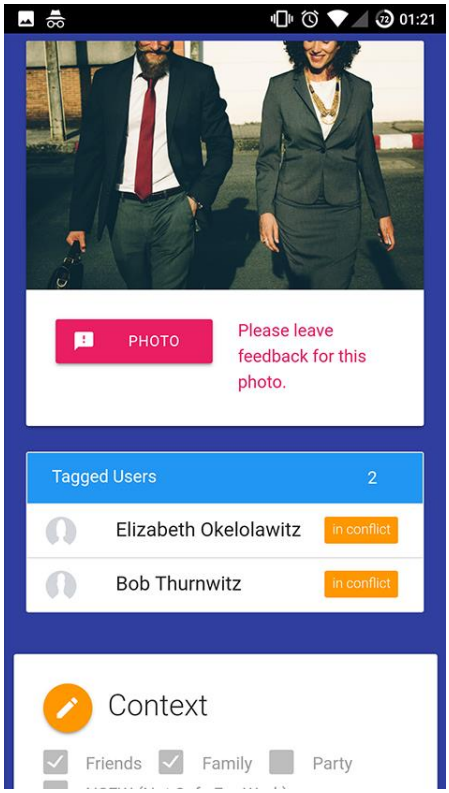

Figure 30 - Mobile Photo Detail page 2

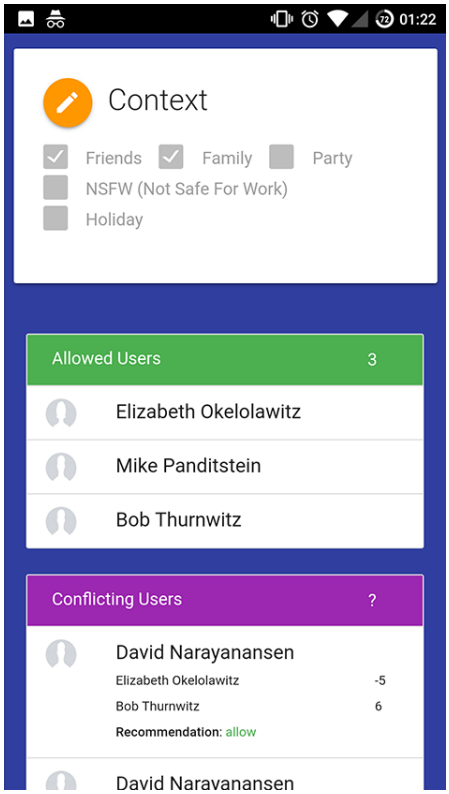

Figure 31 - Mobile Photo Detail page 3

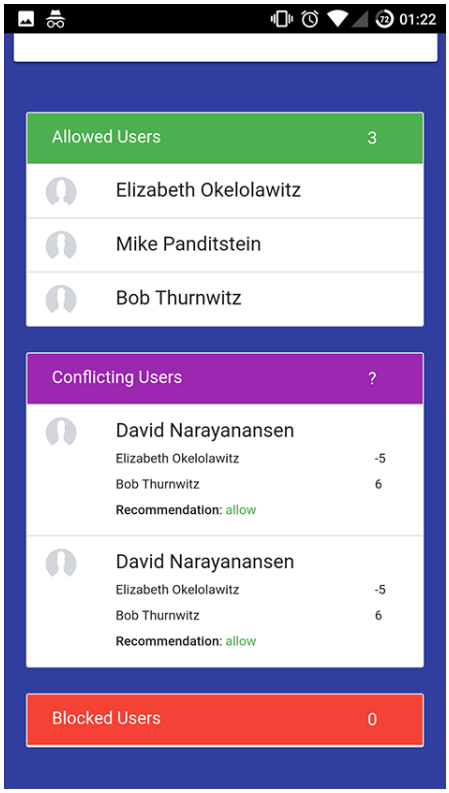

Figure 32 - Mobile Photo Detail page 4

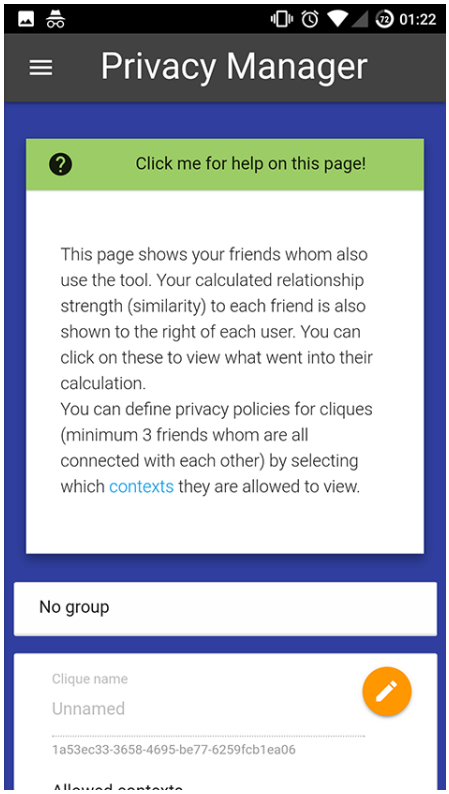

Figure 33 - Mobile Friends page 1

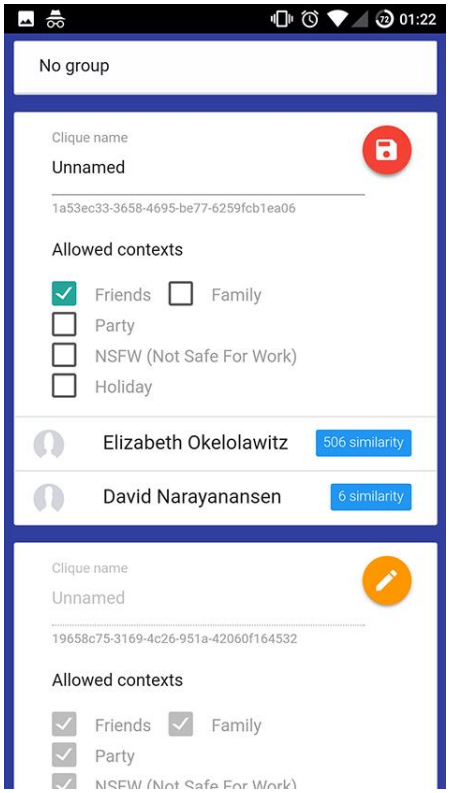

Figure 34 - Mobile Friends page 2

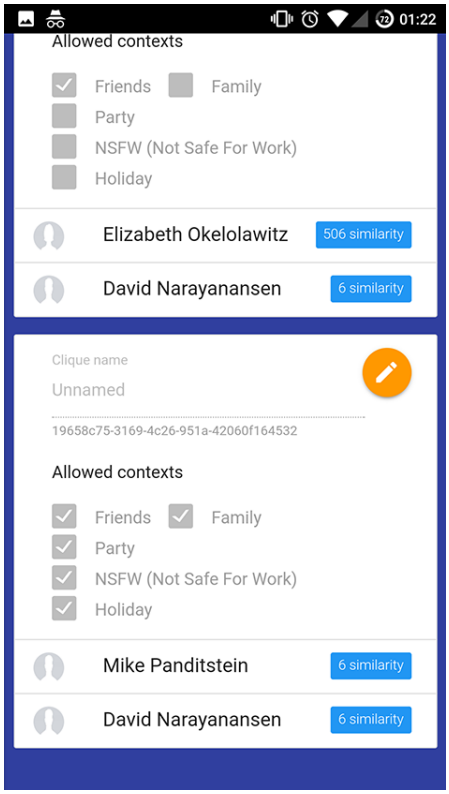

Figure 35 - Mobile Friends page 3

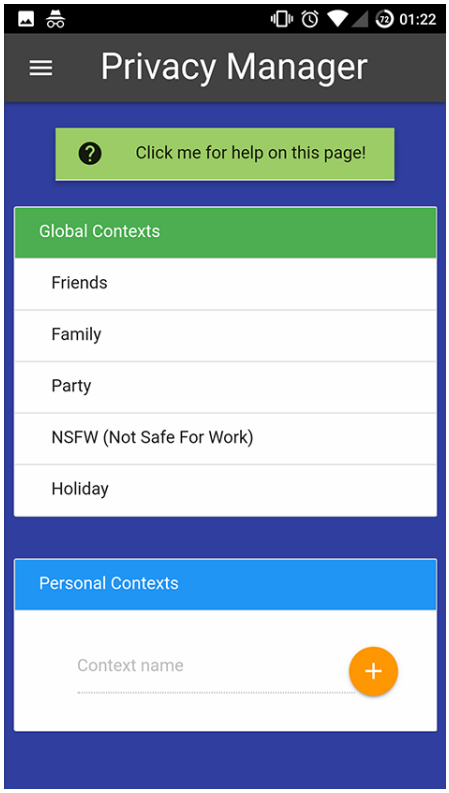

Figure 36 - Mobile Contexts page

Desktop (1920 x 1080, 100 DPI)

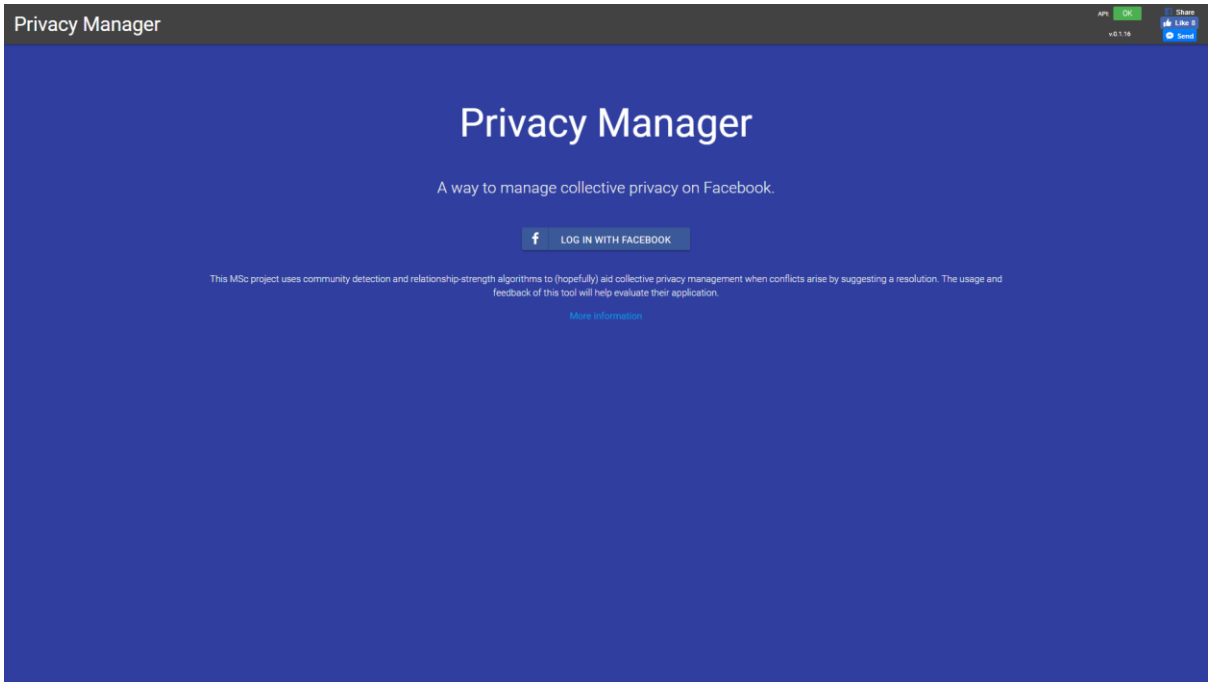

Figure 37 - Desktop Index page

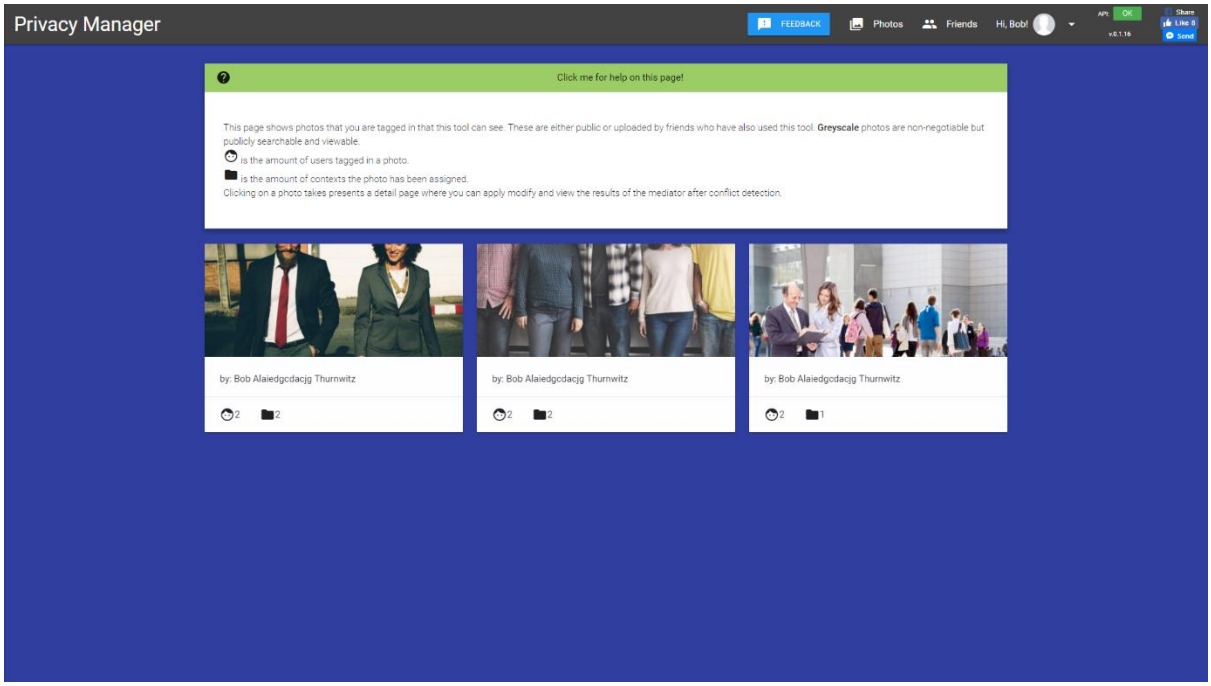

Figure 38 - Desktop Photos page

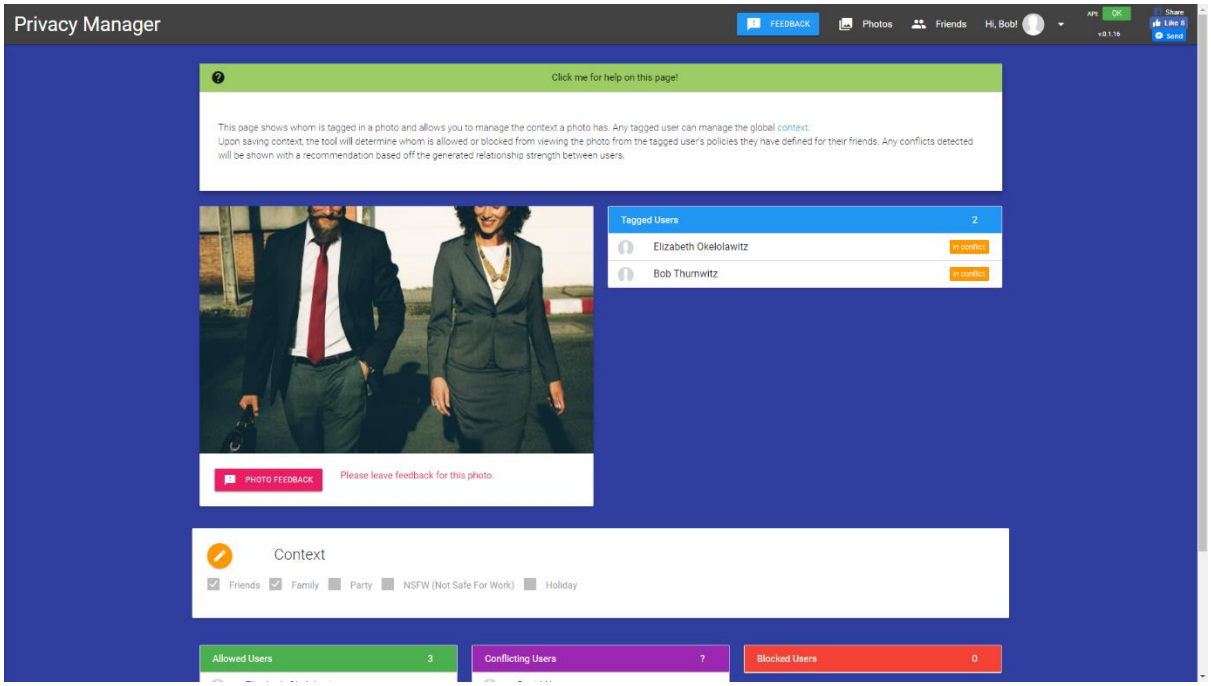

Figure 39 - Desktop Photo Detail page 1

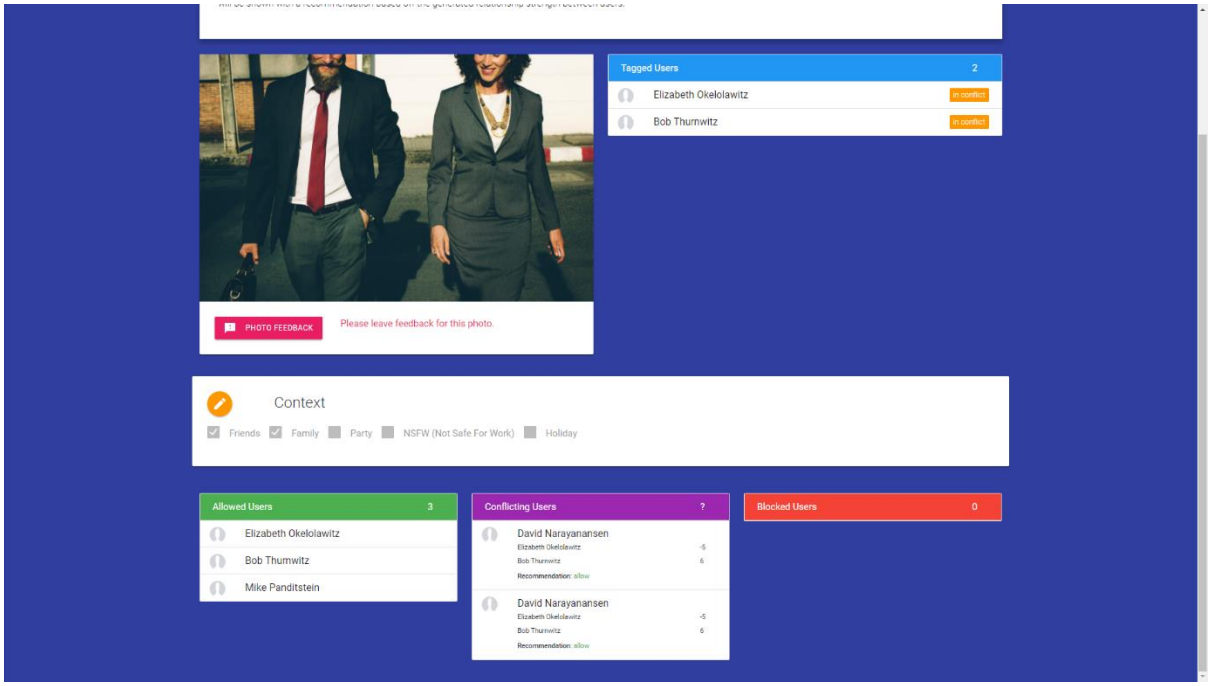

Figure 40 - Desktop Photo Detail page 2

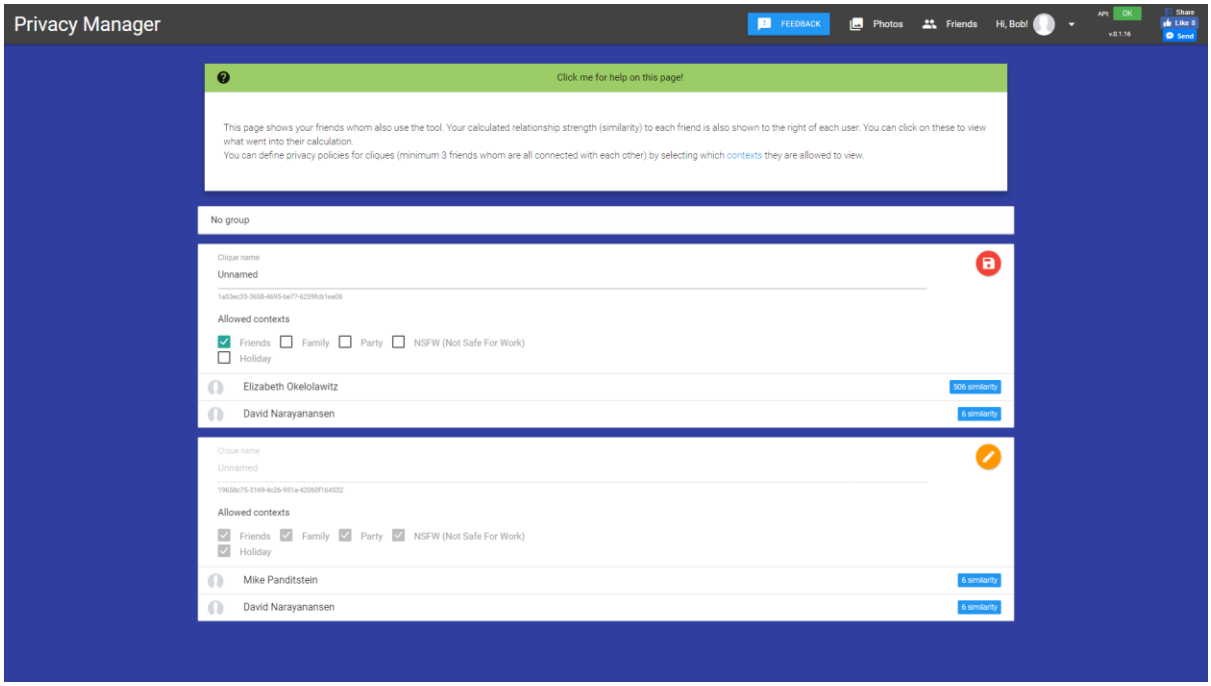

Figure 41 - Desktop Friends page

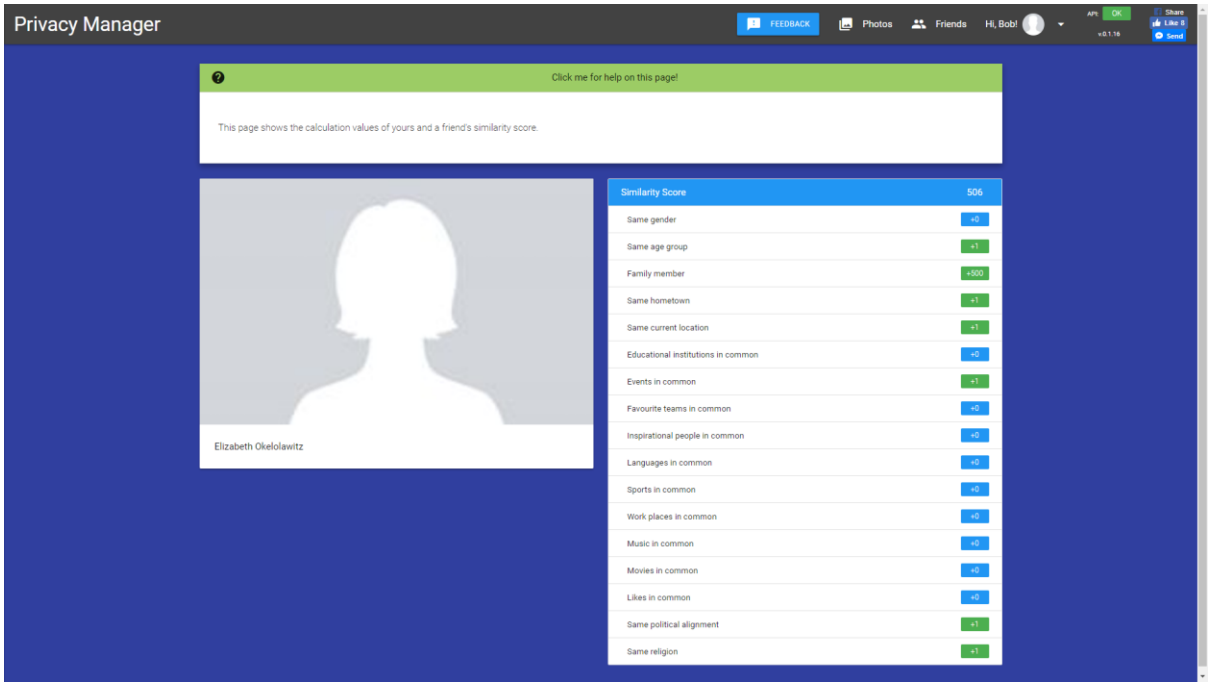

Figure 42 - Desktop Friend Detail page

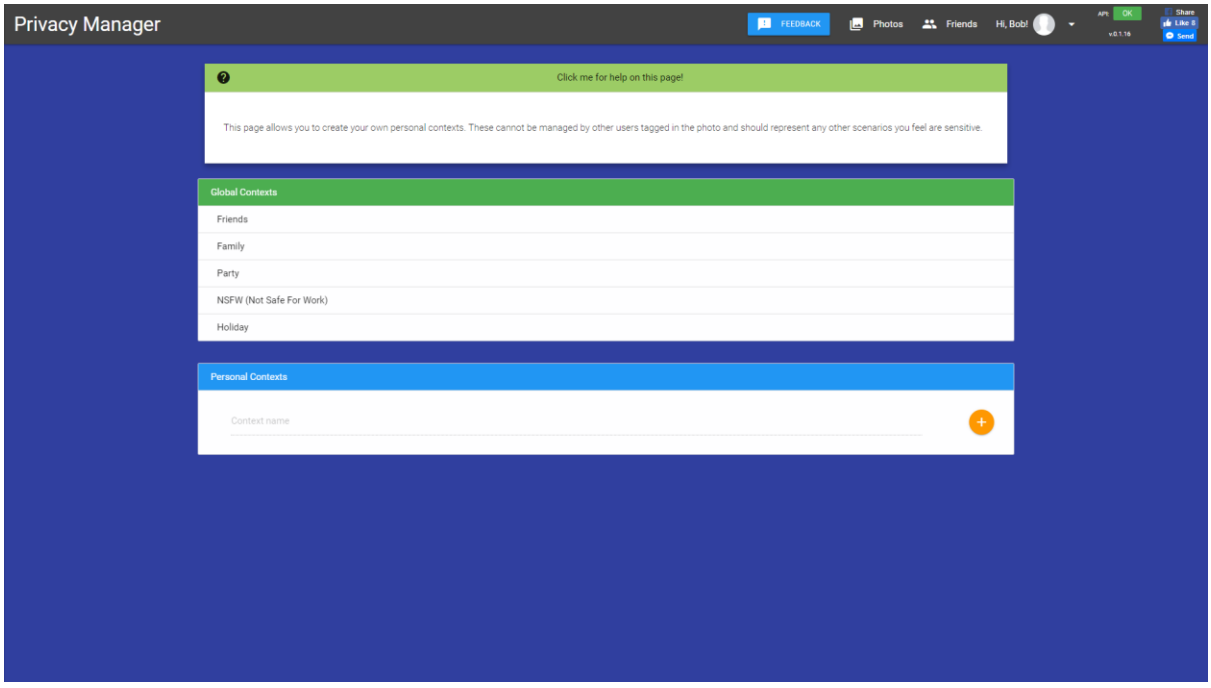

Figure 43 - Desktop Contexts page