University of Moratuwa

Department of Computer Science and Engineering

CS 4202 –Research and Development Project

Final Year Project Report

API for Diaspora Distributed Social Network

Project Group – 16

Chandrasena D.H.D.M (090061R)

Herath H.M.A.B (090178G)

Herath S.W.H.M.S.P (090181J)

Iroshan A.K.A (090196J)

Project Supervisor

Dr. H. M. N. Dilum Bandara

Co-Supervisor

Prof. Gihan Dias

Coordinated By

Dr. Malaka Walpola

5th of December 2013

Abstract

Project Title : API for Diaspora Distributed Social Network

Authors : Madhawa Chandrasena - 090061R

Aruna Herath – 090178G

Sandaruwan Herath – 090181J

Akila Iroshan - 090196J

Coordinator : Dr. Malaka Walpola

Supervisors : Dr. H. M. N. Dilum Bandara

Co-Supervisor : Prof. Gihan Dias

Centralized Social Networks (CSNs) such as Facebook, Twitter, and Google+ have millions of users. However, personal information/data submitted by users to these sites are completely out of the users' control and the presentation of this information is determined by CSNs. Distributed Social Networks (DSNs) are emerging as a viable alternative to address these problems. Users in DSNs can choose a server, which they trust to host their personal data. The users have the ownership and are in complete control on who should see their personal information and what restrictions there are to be enforced while disseminating the data. However, the reach/usefulness of these DSNs is limited due to the absence of an Application Programming Interface (API) for third-party app development.

We address this key limitation by developing an Application Programming Interface (API) for third-party app development in DSNs. Using the API we also developed a real-name-based user search application, which addresses another key limitation in DSNs. Both the API and search application is developed for the "Diaspora" DSN. Diaspora was selected due to its large user base, number of deployed servers/pods, and developer support.

Third-party app developers can use our API to gain controlled access to users' information (e.g., profile information, friend list, and wall posts) to develop social apps such as Wunderlist and social games like Farmville. As a part of this work, we implemented a distributed app authentication scheme called "DAuth" for Diaspora based on the industry standard “OAuth”. This is the first time such an app authentication model is implemented for a DSN.

Many users in DSNs demand different features, sometimes creating conflicts among themselves. Using our API it is possible to provide those features in the form of a third-party app than integrating those features to the core of the DSN. This enables those apps/features to be used by only the interested users. One such example is a user’s interest to be searched by others or not. Currently DSNs only facilitates searching for friends using their DSN-specific user identifiers. This mechanism is restrictive and not user friendly because a user has to remember their friends’ DSN-specific IDs. Search app was implemented as a separate web application to provide more privacy to users, e.g., eliminating ones who do not like to be searched and it provides real name and location based user searching. Furthermore, the search app demonstrates the utility of our API.

Table of Contents

Abstract i

Table of Contents iii

List of Figures vi

List of Tables vii

1. Introduction 1

1.1. Motivation 1

1.2. Contributions 2

1.3. Outline 2

2. Literature Review 3

2.1. Distributed Social Networks (DSNs) 3

2.1.1. Buddycloud 3

2.1.2. Diaspora 4

2.1.3. Freenet 5

2.1.4. Friendica 6

2.2. Diaspora Architecture 7

2.2.1. Overall Architecture 7

2.2.2. Diaspora Models 9

2.2.3. Diaspora Security Architecture 10

2.2.4. Diaspora Code Architecture 11

2.3. API for Third-Party Apps 13

2.3.1. Existing APIs 13

2.3.2. API Design Guidelines 15

2.4. Information Discovery in Distributed Systems 18

2.4.1. Peer-to-Peer (P2P) Information Discovery systems 18

2.4.2. Diaspora Contact Search 19

2.4.3. Information Discovery Mechanisms in Distributed Networks 21

2.4.4. Trust Management in Online Interactions 26

3. Problem Statement 28

3.1 Goals 28

3.2 Objectives 28

3.3 Deliverables 28
4. Diaspora API Design and Implementation 30
4.1 Authentication Model 30
4.1.1 Public and Private Data 30
4.1.2 OAuth for Distributed Resource Owners 30
4.1.3 DAuth 31
4.1.4 Token Management Entity Relationships 34
4.2 API Architecture 38
4.2.1 Class diagram 38
4.2.2 Sequence diagram 40
4.3 API Specification 41
4.4 API Implementation Details 42
4.4.1 Project Process 42
4.4.2 Version Control 42
4.4.3 Coding Standards and Best Practices Guidelines 43
4.4.4 Website 44
4.4.5 Testing 44
5. Diaspora Search Application Design and Implementation 45
5.1 Diaspora Search Application High Level Architecture 45
5.2 4+1 Model – Diaspora Search Application 46
5.2.1 Logical View 46
5.2.2 Implementation View 48
5.2.3 Process View 50
5.2.4 Deployment View 52
5.2.5 User Cases View 53
5.3 Implementation Details 54
5.3.1 Coding Standards and Best Practices Guidelines 54
5.3.2 Testing 55
6. Summary 56
References 59
Appendix I 62
Appendix II 71

Abbreviations..99

List of Figures

Figure 1 - Diaspora high level architecture......7
Figure 2 - Diaspora architecture 8
Figure 3 - Interaction of contact, person and user entities......9
Figure 4 - Diaspora security architecture......10
Figure 5 - OAuth model......17
Figure 6 - How a Diaspora pod finds a user on different pod......20
Figure 7 - Cache strategy of distributed caching and adaptive search......24
Figure 8 - Graph representation of multiple sequential search algorithm 25
Figure 9 - Example: can Alice trust Eric 27
Figure 10 - API authentication and authorization process......33
Figure 11 - E-R Diagram - Token management model 34
Figure 12 - Manifest creation for developers......37
Figure 13 - User give authorize to App 37
Figure 14 - API Architecture 38
Figure 15 - API class diagram 39
Figure 16 - Sequence Diagram – “get_user_person_list” API function......41
Figure 17 - Immanent GitHub repository 43
Figure 18 - Project website......44
Figure 19 - Search app high level architecture 45
Figure 20 - Class diagram - Search app......47
Figure 21 - Sequence diagram - Contact search 48
Figure 22 - Component diagram - Search app......49
Figure 23 - Activity diagram - Search app......51
Figure 24 - Deployment diagram - Search app......52
Figure 25 - Use case diagram - Search App 53
Figure 26 - Contact search page - Search app......54

List of Tables

Table 1 - Get Auth Token 72
Table 2 - Get Refresh Token 73
Table 3 - Get Access Token 74
Table 4 - Get user aspect list 76
Table 5 - Get followed tag list 77
Table 6 - Get user details 78
Table 7 - Get user contact handle list 79
Table 8 - Get user app scopes 81
Table 9 – Edit user email 82
Table 10 – Edit last name 83
Table 11 – Edit user location 84
Table 12 – Get status list 85
Table 13 – Get comment list 87
Table 14 – Get likes for a status message 88
Table 15 - Get number of comments for a status message 90
Table 16 – Create status message 91
Table 17 - Delete status message 92
Table 18 – Get given user comment list 93
Table 19 – Get likes count 95
Table 20 – Create comment 96
Table 21 – Delete comment 97
Table 22 - API error codes 98

1. Introduction

1.1. Motivation

Centralized Social Networks (CSNs) such as Facebook, Myspace, Twitter, and Google+ have tens of millions of users using them every day. Once the user submits his/her personal information/data to these websites, it is stored on the site's servers, completely out of the user's control. Presentation of this information largely depends on the design of the social networking service rather than the users' preferences. Moreover, these CSNs form information silos, as it is not straightforward to port personal information, friend list, and history from one CSN to another. Consequently, to experience functionalities of a different social networking service, users have to re-enter their personal information and re-declare their friends.

Distributed Social Networks (DSNs) are emerging as a viable alternative to address these problems. Diaspora [1], Friendica [2], BuddyCloud [3], and Freenet [4] are some of the example DSNs. Users in DSNs can choose a server, which they trust to host their personal data. The users have the ownership and are in complete control on who should see their personal information and what restrictions there are to be enforced while disseminating the data. Further, users would not lose their data in a situation where the proprietary service hosting their data decides to shut down without a prior notice. However, the popularity and usefulness of these DSNs are limited due to the absence of several key features compared to CSNs such as API for third-party app developers, real name-based search and online chat.

API for a DSN, is essential in expanding the services provided by a social network and its popularity. Further, many users in DSNs demand different features, sometimes creating conflicts among themselves. Both of these limitations in DSNs can be overcome by developing an API for third-party app developers. With an API, app developers can gain controlled access to users' information to develop social apps such as birthday calendars. Moreover using an API it is possible to satisfy different user demands in the form of an app, such that only the users who are willing can use it.

Searching friends is an essential feature in a social network. Currently DSNs only facilitate searching of friends by their DSN-specific user identifier (e.g., user@pod.com). This mechanism is restrictive and not user friendly because a user has to remember their friends' DSN specific IDs. When adding a new friend they may not even know their DSN-specific IDs. Using a search app it is possible to enable the searching of users based on their real name, location, etc.

1.2. Contributions

We have chosen Diaspora as the DSN to work in our project because currently it is the well-known DSN. Further it has more user base and friendly developer support compared to the other DSNs which were mentioned previous section.

Absence of an API for third-party app development was one key limitation in the Diaspora network. Hence we implemented an API for Diaspora DSN as the main part of the project. We have designed and implemented distributed app authentication scheme called “DAuth” for Diaspora API based on the industry standard “OAuth” [5], which is a centralized implementation. Our API hides the complexities of the internal distributed network and presents a seamless interface to the app, where app can perform actions on behalf of the user while getting controlled access to the user’s private data.

To improve the information discovery in Diaspora DSN, we implemented a search app as a separate web application such that it can provide more privacy to users, e.g., eliminating ones who does not like to be searched and it provides real name and location based user searching.

1.3. Outline

Chapter 2 reviews existing literature that is relevant to this project. Chapter 3 presents the problem formulation which discusses the project goals, objectives, and intended deliverables. Design of the API is presented in Chapter 4. Specific details about the search app are presented in Section 5. Concluding remarks are given in Chapter 6. API specification and Software Requirement Specification (SRS) can be found in Appendix I and II, respectively.

2. Literature Review

A Distributed Social Network (DSN) is an Internet-based social networking service that is decentralized and distributed across distinct providers [6]. These are networks in which user data is not stored in any centralized server. Instead users in DSNs can choose a server, which they trust to host their personal data. Then users have the ownership and are in complete control on who should see their personal information and what restrictions there are to be enforced while disseminating the data. In this chapter we analyze of existing literature relevant to our project.

Section 2.1 presents a summary of several existing DSNs. Section 2.2 presents a detailed description about the Diaspora architecture, which is the DSN for which we developed the API. In Section 2.3 we review the literature related to implementing an application API for social networks. Literature related to information discovery on distributed networks are presented in Section 2.4.

2.1. Distributed Social Networks (DSNs)

There are many social networks with different features and different levels of popularity. But most of the popular social networks are centralized ones. This is because CSNs were implemented first and compared to centralized social networks, DSNs lack many features and are much harder to develop. We researched about many DSNs with a special attention on how those systems can be improved, and their current user base. In this section we present four DSNs namely Buddycloud, Diaspora, Freenet, and Friendica. We evaluate them based on their technologies, features, and overall architecture.

2.1.1. Buddycloud

Buddycloud is one of the popular DSNs. Simon Tennant is the founder of Buddycloud [3]. Buddycloud is designed by combining good features of existing social networks like Facebook and Google+. Being a DSN it enables users to preserve their privacy. In Buddycloud users can create any number of their own channels. Users have full control over everything they post and share.

Buddycloud use several key words such as channels, moderators, and producers. Buddycloud Channel [7] is a user account. Users can post to it. There are two types of channels; personal channels and topic channels. Personal channels represents real persons.

Topic channels represents a group of people interested in something. Producer is the channel owner and he can grant followers to read and write on the channel and he can change moderators of the channel. He also can change followers' roles, remove followers as well as moderators. Moderators can delete content on behalf of the producers. Buddycloud web client is the web user interface of the BuddyCloud. It is built with backbone.js. Buddycloud has a great API built with node.js and JavaScript. Following are some of the major features in Buddycloud:

- With a single click users are able to edit properties like channel name, channel description, status, privacy, location, block or promote users and delete channels.
- All the notifications will be sent to user's email address. Users also can simply ignore those notifications by unmarking them from the list.
- Users can create many topic channels as they like. Users can give capabilities to their followers to read the channel and post in the channel. They can also simply discard the channel when necessary.
- Users can search channels as well as posts in those channels.
- Users can simply drag and drop videos and photos into the channel to post them.

2.1.2. Diaspora

Diaspora project and the Diaspora Inc. were founded by Dan Grippi, Maxwell Salzberg, Raphael Sofaer and Ilya Zhitomirskiy, students at New York University's Courant Institute of Mathematical Sciences. They started development on 2010 using donations raised. Early summer 2012 Diaspora Inc. announced they would transition Diaspora project to be community governed. Sean Tilley of Diaspora Inc took charge of setting up community governance tools. From that point, Diaspora is maintained by a community.
Diaspora allows its users to:

- Post texts, images, videos on the network
- Insert hashtags to categorize them, and mentions to call other users
- Comment, like and re-share posts of the others

But especially, Diaspora is oriented on privacy, hence Diaspora users can:

- Choose the server where you register, or deploy your own server to keep your data safe

- Choose who has access to your posts by affecting your contacts to groups called "Aspect"
- Switch your account from one server to another easily

Compared to popular social networks it lacks the following features:

- Groups
- Events
- Likes and comments on comments
- API for application developers

2.1.3. Freenet

Freenet is a free distributed social networking software [4] which lets users anonymously share files, browse and publish free sites which are accessible only through Freenet and chat on forums without fear of censorship. Freenet is decentralized to make it less vulnerable to attack. Communications by Freenet nodes are encrypted and are routed through other nodes to make it extremely difficult to determine requester and content of the information.

Users contribute to the network by giving bandwidth and a portion of their hard drive (called the "data store") for storing files. Files are automatically kept or deleted depending on how popular they are, with the least popular being discarded to make way for newer or more popular content. Files are encrypted, thus generally the user cannot easily discover what is in his data store, and hopefully cannot be held accountable for it. Chat forums, websites, and search functionality, are all built on top of this distributed data store.

An important recent development, which very few other social networks have, is the “darknet” feature. By only connecting to people they trust, users can greatly reduce their vulnerability, and yet still connect to a global network through their friends’ friends’ friends and so on. This enables people to use Freenet even in places where Freenet may be illegal, makes it very difficult for governments to block it, and does not rely on tunneling to the "free world".

Following are some of the major features in Freenet [8]:

- Freemail - Freenet own anonymous email service
- Frost - Messaging and file sharing tool

- jSite - It is a Freenet website
- Thaw - File sharing utility and upload/download manager with a GUI

2.1.4. Friendica

Friendica [2] is an open source DSN software which is developed using PHP, Apache and MySQL. 3.1 is the current stable Friendica version and it was released on December 4, 2012. Currently around 40 developers works in the Friendica project and more than 30 public portal servers runs Friendica.

Friendica provides good support for its developers. It maintains a code repository at Github [9], has a separate mailing list [10] for the developers and users and also it has an issue tracker [11]. But it does not contain a public IRC chat line and good documentation for developers. Following are some of the major features in Friendica:

- Friendica supports unlimited length text status with different styles, (e.g., bold, italics, colour, size, etc.) and multimedia (e.g., video, links, YouTube, Vimeo, SoundCloud, uploaded photos, file attachments, etc.). User can use default location or use browser tracking for location/geo-tagging of posts. Friendica facilitates common tag formats including @mentions and #tags. A post contains likes, and dislikes buttons and users can re-share public posts of others.
- Friendica has a better birthday notifications system which is time zone corrected.
- Friendica has an event feature and shared events with calendar.
- Friendica facilitate users to upload photo albums with full privacy support and photo tagging.
- Initially a default public profile is provided which may be restricted according to user’s preference. Multiple profiles can be created. Existing profiles can be cloned to create similar profiles which only differ in minor details. Profile information are categorized such as personal, contact, work, relational information and hobbies/interest.

2.2. Diaspora Architecture

2.2.1. Overall Architecture

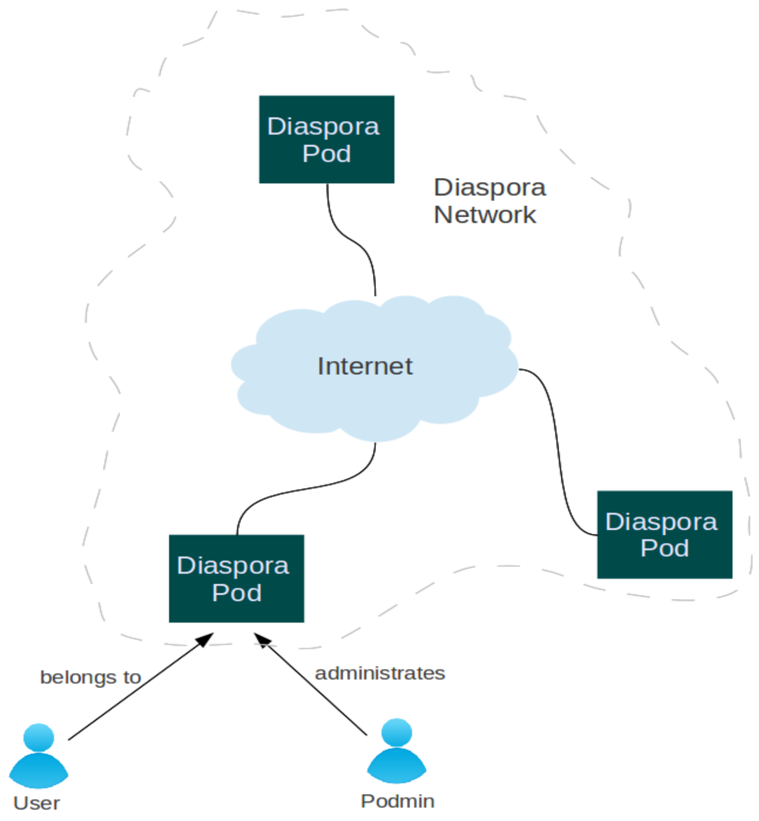


Figure 1 - Diaspora high-level architecture

Diaspora servers are known as *pods*. Diaspora pods communicate with each other making the Diaspora network (see Fig. 1). A *podmin* is the one who administrates the pod. Podmin updates the Diaspora software on the pod. A user is the one with a Diaspora account in the pod. A user can chose a pod to create an account. User trusts the podmin and the pod she has created the account in. Alternatively someone can setup a Diaspora pod, create an account in it, and become the podmin himself.

Podmin decides which version of Diaspora to use, hence pods in Diaspora network can have different versions. He also can change the code herself to give custom functionalities to the users of her pod. Some podmins run only the stable releases of Diaspora software while some run the most updated versions from the development branch. Some also change the looks of views slightly to improve the user experience.

Users can share with or follow users from their own pod or users from other pods. Currently to discover a user from another pod that users Diaspora handle is needed. Diaspora pods communicate with each other using the Diaspora Federation protocol.

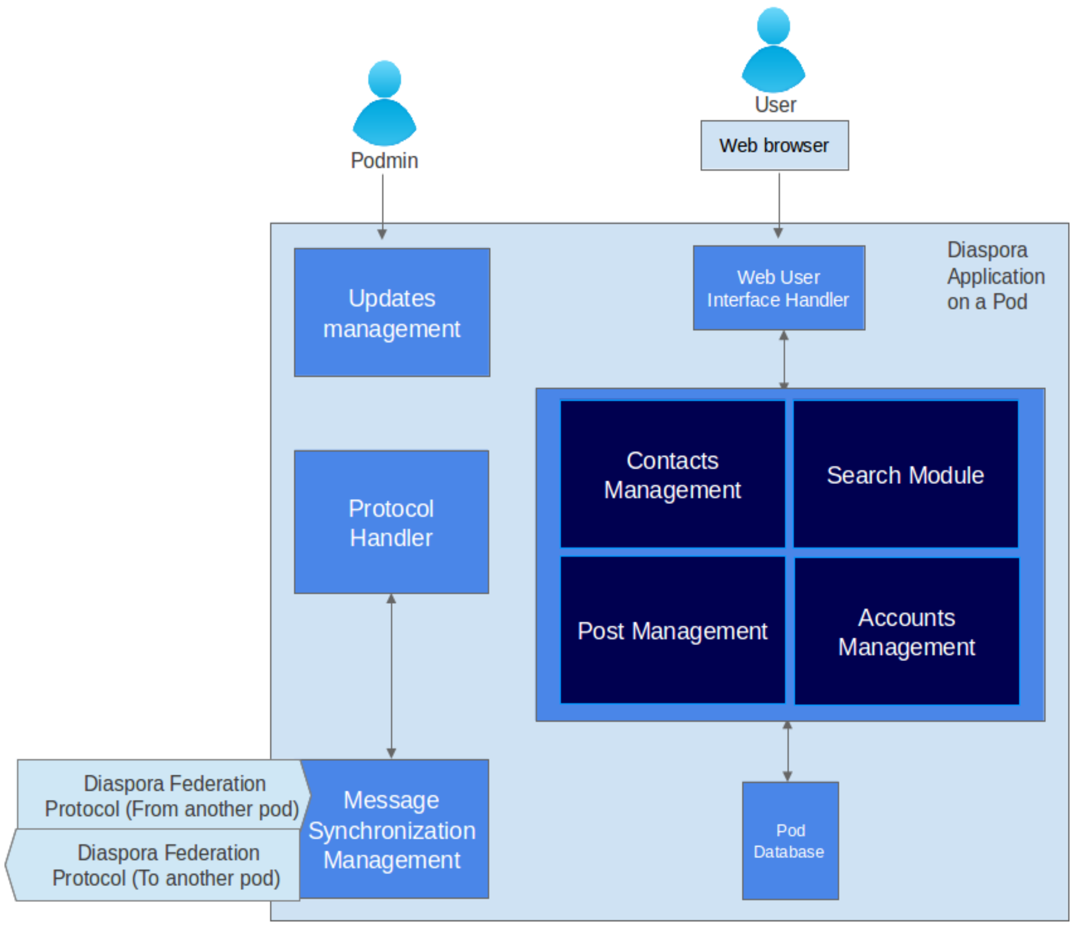

Podmin
User
Web browser
Diaspora Application on a Pod
Updates management
Web User Interface Handler
Contacts Management
Search Module
Protocol Handler
Post Management
Accounts Management
Diaspora Federation Protocol (From another pod)
Diaspora Federation Protocol (To another pod)
Message Synchronization Management
Pod Database

Figure 2 - Diaspora architecture.

Update manager represents the fact that the podmin is responsible to keep his pod up-to-date by upgrading the code when a new version is released. The *message synchronization management* keeps pods synchronized: each time a pod does a write in the database, this module transmits the information to the other pods in a message. Contacts Management, Search Module, Post Management, and Accounts Management are collectively known as Data Management. The modules are chosen depending to the type of data, but each has the same role, thus each has access to the UI, the database and the message synchronization. If access to some data is requested by the UI, the module corresponding will be called, and will interrogate the database. If requested data is not found inside the database, the module will call the message synchronization management to search the data in the other pods [12].

2.2.2. Diaspora Models

Following are the list of entities in Diaspora [1].

- User – represents the private information and capabilities of a user on that server. A User has a Person. Private Key of a user used in every communication is stored in the user object.
- Person – A Person is a User viewed from the outside. Person objects are replicated across servers, and they are where a User's public key lives. A Person has many Posts. A Person has a Profile.
- Contact – Is a "proxy" object for every person a User is friends with. Contact object contains details of the relationship between the User and the relevant Person. If Alice follows Bob and Bob does not share with Alice, only Bob's public posts are visible to Alice. If Bob shares with Alice, post Bob share with Alice will be visible to her.

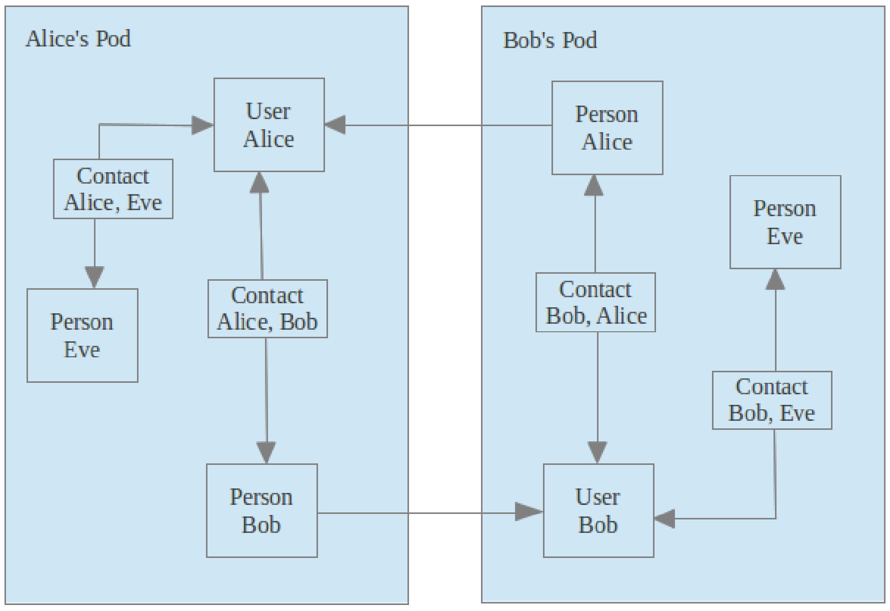


Figure 3 - Interaction of contact, person and user entities.

- Profile – This contains information about the person.
- Request – This is a friend request object that gets sent to another person.
- Aspect – This contains a list of people, and posts which are for that aspect.
- Post – A Post belongs to a Person. This is a parent class for different types of posts, it contains comment ids and a few other attributes common to all Posts.
  - Status Message inherits from Post
  - Album inherits from Post
  - Photo - inherits from Post

- Comment
- Like

2.2.3. Diaspora Security Architecture

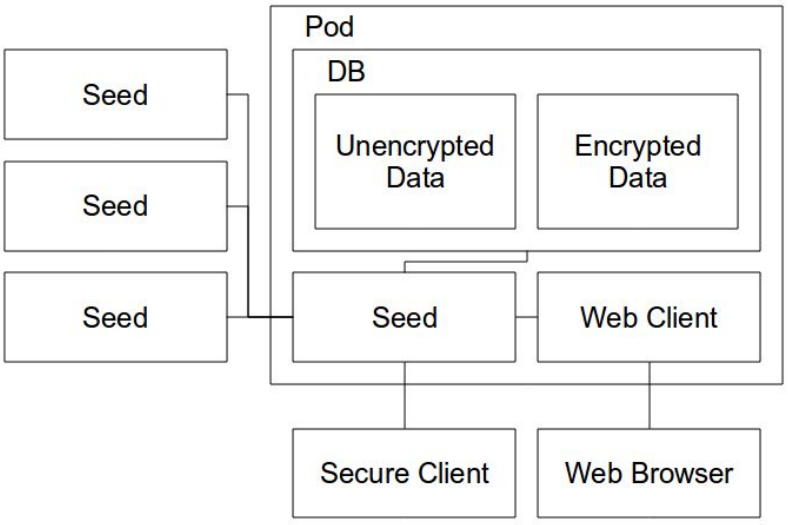


Figure 4 - Diaspora security architecture.

Figure 4 shows Diaspora security architecture [13]. Details description of the components given below:

- Web Client – The web client is a HTTP interface built into pods and so that users can easily connect to their seeds from anywhere.
- Secure Client - Secure client is an alternative to the web client for advanced users. It is a local client which communicates to the user’s seed via the client API. This will allow more secure communication.
- Seed - Seed is user’s Diaspora account. Data is pushed to it from the clients such as web client or secure client via the client API. It stores/retrieves data in/from the Pod’s database. Seed communicate with other seeds via the Seed API to notify of and exchange data.

Security Models

Diaspora introduces three different security levels that can be selected by the users. Details description of the security levels is given below:

- Security level with “None” - Post is not encrypted and available to anyone. They can be posted from any Client, stored by Seeds without encryption, and likewise communicated between Seeds without encryption. Any Seed can request the Post from its Owner's Seed.

- Security levels with “Low” – Post is encrypted for consumption by that Post’s audience at the Seed level. The Seed provides Audience Seeds with the Encrypted Post on request. Posts and their keys can be stored in Owner/Audience Seeds/Pods unencrypted or encrypted as required by computation/storage limitations.
- Security level with “High” – Each User has a key pair for use. The private key of this pair is kept secure by the User, on his Secure Client. A Post can be made from any Client that has been given the private key.

Communication between components

- User to Remote Client – Communication is encrypted. Dual authentication is required. Users authenticate via a login name and password.
- Client to Seed – Communication is encrypted for Low security Posts. Dual authentication (User and Seed) is required.
- Seed to Seed – Communication can be unencrypted (secure Posts are encrypted anyway). Owner authentication required.

2.2.4. Diaspora Code Architecture

Diaspora is a Ruby on Rails application [1]. Rails is Model View Controller (MVC) architecture-based web framework for Ruby language. Diaspora is currently based on Rails 3.2.11, requiring Ruby 1.9.2 as minimum (1.9.3 recommended). Diaspora pulls in a lot of Ruby gems, so has high amount of Ruby gem dependencies; 173 gems as of January 13, 2013.

Sidekiq in collaboration with Redis is used for background job processing in Diaspora. On and before version 0.0.3.4 Diaspora used resque as the background job processing engine. Sidekiq has replaced resque in the current development version and will be available in the next stable release. Redis is an open source, networked, NoSQL, key-value datastore. Redis has built-in persistence (snapshotting to disk) and more complex data types for key value pairs in addition to simple strings. For Diaspora, background processing of jobs is important, because a lot of network communication is done with other servers. To avoid UI being blocked while that is taking place, background job processing is used. When a post is written, the server saves it to the database and hands processing (e.g., sending the post to users on different servers) over to Sidekiq, which then takes place in the background as a separate process.

Usually templates in views of rails applications are written in ERB (HTML with inline Ruby). But instead of ERB Diaspora uses HAML. HAML is a lightweight, minimalistic templating language to generate XHTML. It aims to make markup as elegant as possible. HAML can function as a replacement for many inline page templating systems such as PHP, ASP other than ERB. HAML avoids the need for explicitly coding HTML into the template, because it itself is a description of the HTML, with some code to generate dynamic content.

Diaspora uses SASS framework for CSS. It can be thought of as the CSS equivalent of HAML. The aim of using SASS is to create style sheets that are easier to maintain. SASS adds features like nesting rules and variables to CSS. SASS needs to be converted to standard CSS by the web framework that uses it. In the case of Rails this conversion is done by the Rails asset pipeline. The new main syntax (as of SASS 3) is known as SCSS (for "Sassy CSS"). As SCSS is an extension to CSS3 every valid CSS3 style sheet is valid SCSS as well. Diaspora uses both SCSS and old SASS with old syntax. They are working on completely moving to SCSS.

Backbone.js and Handlebars.js frameworks are used on front end. Backbone is a JavaScript library with a RESTful JSON interface. It is based on the model-view-presenter application design paradigm. Its only dependency is underscore.js so its lightweight. Backbone is intended to be used in implementations of single page applications. It can be used in any application to avoid unnecessary page r. Backbone is a way to structure an application better. It makes clear decoupling between data and views that render that data. Handlebars.js is a client side templating engine that compiles data into web elements. The data handled by Backbone is rendered using handlebars. Diaspora plans to do most of the templating on the client side in the future. Federation between pods is implemented using salmon protocol.

Diaspora code contains fully automated Rspec, Jasmine and Cucumber tests. Diaspora's aim to have a full coverage of tests, is enforced by not accepting code that does not have tests with it. Ruby tests are written in Rspec while UI tests involving JavaScript are done using Jasmine framework. Cucumber is used to write integration tests. Every test is automatically run in Travis Continuous integration against every pull request.

2.3. API for Third-Party Apps

An Application Programming Interface (API) is a protocol which is intended to be used as an interface by software components to communicate with others. Software which releases the API allows developers to access its services more efficiently. Software developers use these APIs to design products that are powered by the services of that APIs. We researched about Facebook, Twitter, and Google+ APIs, and how to develop apps using those APIs and how to implement an API to the Diaspora. We also researched literature relevant to API design guidelines.

2.3.1. Existing APIs

Facebook API

Facebook has several APIs [14]. They help each other to build Facebook apps. Graph API is the primary way to get data in and out of Facebook's social graph. It is a simple HTTP-based API. It allows developers to query data, post new stories and any of the other tasks that an app might need to do. Most other APIs at Facebook are based on this Graph API. Facebook Query Language (FQL) allows developers to use a SQL-style interface to query the data exposed by the Graph API. Dialogs provide a simple interface to provide social functionality to app users. Dialogs require user interaction hence they do not require additional permissions. There are number of dialogs available for developers to use like post a story to their Timeline, send a friend request to another user, send a Facebook Message to one or more of their friends etc. Developers can integrate Facebook Chat into their products. Here Instant messaging client connects to Facebook Chat via the Jabber/XMPP service. Facebook supports internationalization for applications. Developers can take advantage of Translations framework immediately, so they can enjoy the benefits that translation can bring to their applications. Other one is Ads API. It is a restricted platform which may only be accessed by whitelisted apps. It allows developers to build their own app as a customized alternative to the Facebook Ads Manager and Power Editor tools. With those APIs developers can easily create Facebook apps.

Twitter API

Twitter API [15] is the programming interface to Twitter. Programmers use the Twitter API to make applications, websites, widgets, and other projects that interact with Twitter. Applications invoke the Twitter API over HTTP. V1.1 most recent version of the Twitter REST API and it support JSON only and discontinue support for XML, Atom, and

RSS. Applications are requested to authenticate all of their requests with OAuth 1.0a or Application-only authentication in v1.1. Twitter also provides Streaming APIs which facilitate streaming data from twitter without calling Twitter API over and over. This API becomes useful when developing real time applications.

Google+ API

Google+ API is the programming interface to Google+ [16]. Developers can use the API to integrate his/her app or website with Google+ which enables users to use Google+ features from within that application. Google+ API allows applications to fetch public data from Google+ and currently it provides read-only access to public data. Most of the Google+ API follows a RESTful API design which means that developer use standard HTTP methods to retrieve and manipulate resources. In app development, application is allowed a limited number of API calls. This ensures that an app does not consume more resources and any app will not impact of the performance of the other apps which are running on same app engine. Most of the API calls require that the user to grant permission to access their data. Google+ uses the OAuth 2.0 protocol to allow authorized applications to access user data. Google+ uses JSON data format to represent the resources in the API. Google+ provides developers to build mobile applications on android, iOS platforms and web applications on C#/.Net, Java, JavaScript, PHP, Python, Ruby platforms.

Google App Engine (GAE) API

The Google App Engine allows web applications to be hosted on a Google server, to use the Google Datastore to store data and to gain controlled access to user's Google account data [17]. The API for the GAE is available in Java, Python and Go languages. It is formed by a several set of services that Google offers. User service is the service mostly used by developers. By using the Users service, facility to log in using a Google account can be provided to the users of the web app. After logging them in the app can access the users name and email address.

There are many other services like the Channel API which allows the application to create a persistent information channel between the app and user's browser, thus the app can push data to the browser. The XMPP service which allows the app to communicate with users - or even other applications - over XMPP instant-messaging protocol. The Mail service allows the app to send e-mails on behalf of the logged in user.

The Google Datastore can be accessed by apps using the GQL query language. Like FQL of Facebook this language has many similarities to SQL. But the Datastore is not a table based database. Data in the Datastore are stored in entities. Each entity has a type and a unique key. Entity groups are formed by defining an entity as the child/parent of another entity.

App development for Google App Engine API

A software development kit which allows developers develop apps locally and later upload to GAE, is available. SDK simulates the Datastore, Google accounts, fetching URLs, sending e-mails and many other services. SDKs are provided for all the 3 languages the App Engine supports.

App developers have to register their apps under a unique application ID, using a Google account. One Google account is allowed to create up to 10 applications for free. After registering an app, developers can upload their apps to the App Engine using the SDK. After uploading, developers can update the uploaded application anytime. A dashboard for the developers is available which gives information like datastore entities, number of hits, resources used by the app, etc. for developers. There are some integrated development environments and tools that supports Google App Engine projects. Pydev plugin for Eclipse is an example.

2.3.2. API Design Guidelines

Behind most of the successful web applications there is an easy-to-use and feature-rich API. Having a well-organized API will help the main application to spread into others and reached by more users. Also, an API-enabled application can be easily enhanced further using the API itself. There are some good characteristics which should take into consideration while designing an API [18].

- Easy to learn.
- Easy to use, even without documentation.
- Hard to misuse.
- Easy to read and maintain code that uses it.
- Sufficient powerful to satisfy requirements.
- Easy to evolve.
- Appropriate to audience.

General Principles of API Design

First thing in the API design process is requirements gathering. Sometimes we might get proposed solutions instead of the requirements. In that case we have to find the better solutions among the proposed solution.

Second fact is starting the developments before finalizing the specifications so that it will save us from implementing a bad API. If you first specify and then begin to implement and decide that this is garbage, you have wasted lot of time specifying requirements.

Since most of the time API is used by the third-party app developers, all the functions should be easy to explain to the API users. API should be as small as possible with satisfying its requirements. In order to maximize information hiding, use suitable access control by minimizing accessibility of everything. All the classes and members should be private as possible. Public classes should not have public fields. Allow modules to be used, understood, built, tested, and debugged independently.

API design is tough. It is not a solitary activity. It is a noble rewarding craft. Perfection in unachievable but designing it right way will improves the lot of programmers, end-user experience.

Authentication

One of the challenging tasks in designing an API is the authentication and authorization process modeling. Users have the ability to maintain his/her privacy limitations. External applications requests to access users' data will be processed through this authentication model and provide the requested information depending on the users' privacy policy.

OAuth Protocol

As the social network community grows, more and more third-party applications will be developed on our planned API. To make those applications more interactive, user data will be needed. The problem is, in order for these applications to access user data, Diaspora ask for username and passwords. Not only does this require exposing user password to third-party applications, it also provides these applications unlimited access to do as they wish. They can do anything even changing the password and lock users out. As a solution OAuth protocol can be adopted for the authentication and authorization which will play a role as a valet key for the web [5]. OAuth give third-party applications limited access to users' account by providing a way to grant limited access like in scope, duration etc.

In the traditional client-server authentication model, the client uses its credentials to access its resources hosted by the server. OAuth introduces a third role to this model: the resource owner. In the OAuth model, the client requests access to resources controlled by the resource owner (resource owner acting behalf of the server), but hosted by the server.

In order for the client to access resources, it first has to obtain permission from the resource owner. The permission is granted in the form of a token and matching shared – secret. The purpose of the token is to make it unnecessary for the resource owner to share its credentials with the client. Unlike the resource owner credentials, tokens can be issued with a restricted scope, limited lifetime.

Following diagram illustrate how the OAuth really works. Instead using clients' credentials, the client is using the resource owner's credentials to make requests.

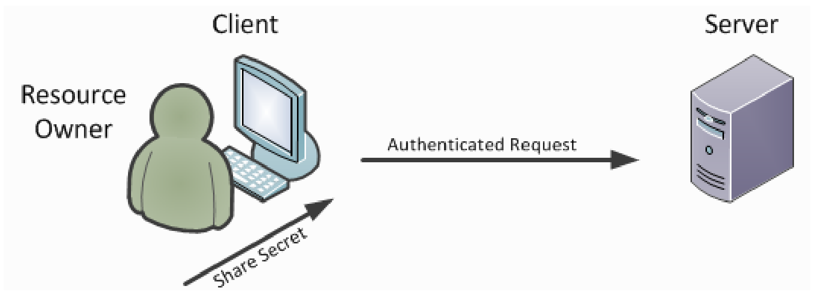


Figure 5 - OAuth model.

Credentials and Tokens

OAuth offers three types of credentials: client credentials, temporary credentials and token credentials. Client credentials are used to authenticate the client. Token credentials are used in place of the resource owner's username and password. Temporary credentials which are used to identify the authorization request.

Turning Rail Site into an OAuth Provider

There is an OAuth plugin and an OAuth gem which will help us to create both OAuth providers and consumers. Consumer is a third-party application which uses another application's data. A provider is a web application that the consumer wants to access. OAuth gem provide following features for the Provider [19].

- User can register their own applications to receive consumer key/secret pairs.

- Users can manage and revoke tokens issued in their name.

Diaspora security architecture defines who should have what data in what form. Most people use Diaspora expecting more privacy than other existing central and DSNs. Some advanced users want end to end encryption, as they do not trust their friends. However this design makes Diaspora complex and cumbersome for the average users. In fact, more secure a system, less user friendly and the more expensive, it becomes.

2.4. Information Discovery in Distributed Systems

Information discovery plays a major role in social networks. Popular social networks, e.g., Facebook, facilitate different searching options such as search by real name, locations, working place, etc., for their users. However, information discovery in DSNs is complex due to their distributed nature. We researched about P2P information discovery systems, diaspora contact and information discovery mechanism and algorithms in distributed systems.

2.4.1. Peer-to-Peer (P2P) Information Discovery systems

Today there are many P2P file sharing systems popular among the world. P2P file sharing allows users to access media files such as books, music, movies, and games using a specialized P2P software programs such as BitTorrent, Gnutella and Kazaa that search for other connected computers on the P2P network and locate the desired content. We researched how the peer discovery happens in those P2P file sharing systems.

BitTorrent

BitTorrent [20] offloads some of the file tracking work to a central server called a tracker. This is how BitTorrent file discovery happens, BitTorrent client software communicates with a tracker to find other computers running BitTorrent that have the complete file (seed computers) and those with a portion of the file (peers that are usually in the process of downloading the file). BitTorrent tracker identifies the swarm, which is the connected computers that have all of or a portion of the file and is in the process of sending or receiving it. BitTorrent tracker helps the client software trade pieces of the file he/she wants with other computers in the swarm. Local computer receives multiple pieces of the file simultaneously.

Gnutella

There is no central database that knows all of the files available on the Gnutella [21] network. Instead, all of the machines on the network tell each other about available files using a distributed query approach. First user enters the details of the file he/she want to find on Gnutella network. User machine should know of at least one other Gnutella machine somewhere on the network. In order to provide that facility Gnutella software has an IP address of a Gnutella host pre-programmed in or user can enter IP address of other Gnutella network. Local machine sends the file name to the Gnutella machine(s) it knows about. These machines search to see if the requested file is on the local hard disk. If so, they send back the file name (and machine IP address) to the requester. At the same time, all of these machines send out the same request to the machines they are connected to, and the process repeats. A request has a time to live (TTL) limit usually 7 or 8. A request might go out six or seven levels deep before it stops propagating. Assume one machine knows of 4 others machines on Gnutella network, and then request might reach 8000 machines on the network.

Kazaa

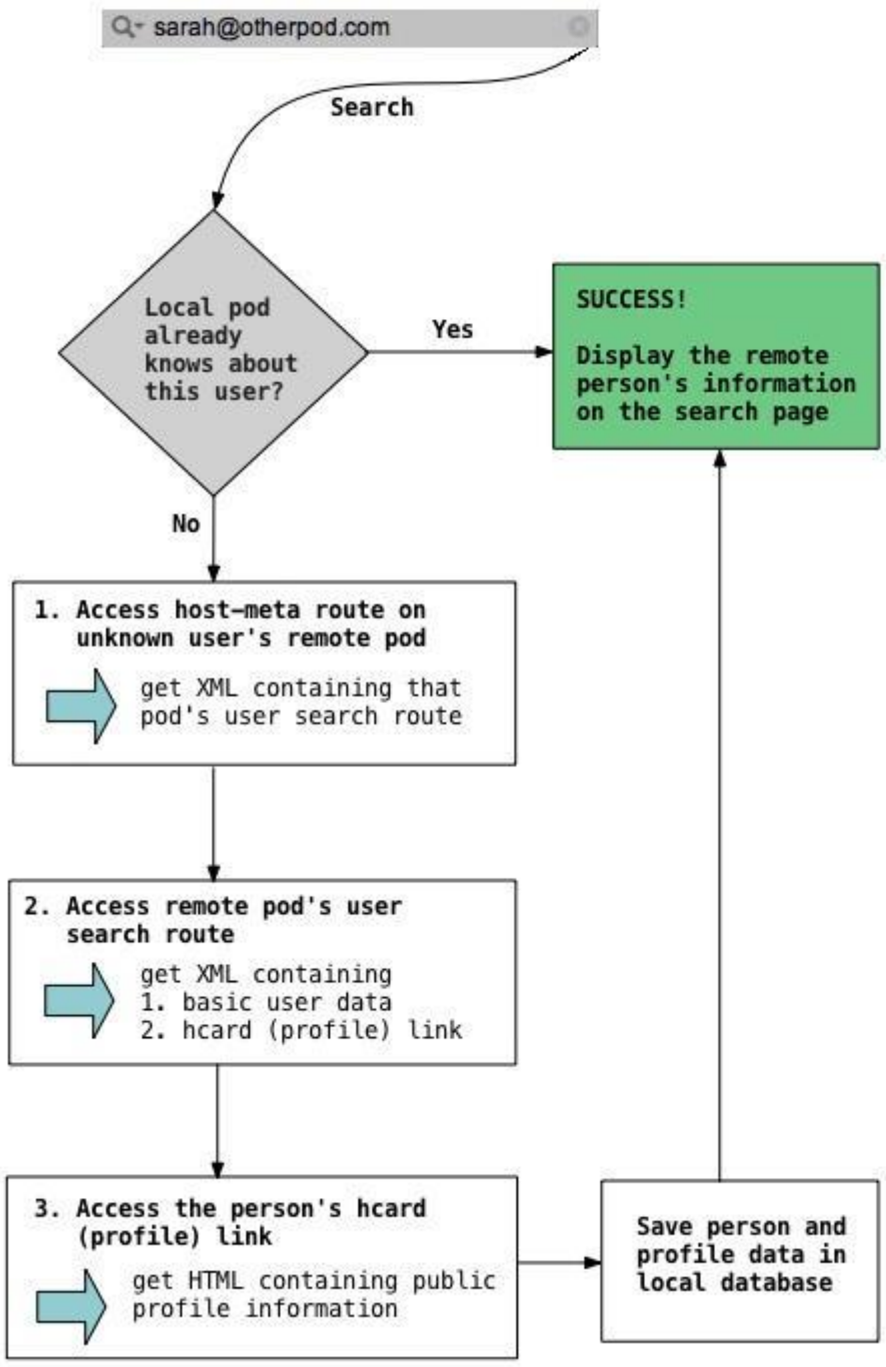
Kazaa [22] system contains two different types of nodes super nodes and ordinary nodes. Super nodes are powerful computers with fast network connections, high bandwidth and quick processing capabilities and which act a lot like traffic hubs, processing data requests from the slower ordinary nodes. Super node can serve between 60 and 150 ordinary nodes at one time.

Kazaa software preprogrammed with a list of super nodes. Every time the user launches the Kazaa application, his/her computer registers with the central server and then chooses from a list of currently active super nodes. When sending out a request for files to download or upload, the request is going through the super node. The super node communicates with other super nodes, regular node, even more regular to fulfill the request until the TTL of 7 runs out so the search request will extend seven levels into the network before it stops propagating. Once the correct file has been located, it is transferred directly from the file owner to the requester using HTTP and no need go through a super node.

2.4.2. Diaspora Contact Search

A single installation of the Diaspora software is a pod. Diaspora is made up of hundreds of pods. A pod may contain one (called as individual pods) or many (called as community pods and run by a person or organization) Diaspora users. But it facilitated to connect with other users on any other pod.

User discovery in Diaspora happens using Diaspora ID. A Diaspora ID is made up of a username, followed by an @ sign, followed by the pod name, e.g., saman@otherpod.com. Here are the step of user discovery process in Diaspora [23].



sarah@otherpod.com
Search
Local pod already knows about this user?
Yes
SUCCESS!
Display the remote person's information on the search page
No
1. Access host-meta route on unknown user's remote pod
get XML containing that pod's user search route
2. Access remote pod's user search route
get XML containing
1. basic user data
2. hcard (profile) link
3. Access the person's hcard (profile) link
get HTML containing public profile information
Save person and profile data in local database

Figure 6 - How a Diaspora pod finds a user on different pod.

When Diaspora gets a search request for a Diaspora ID, the first thing pod does is look in its local database to see if it already knows about requested user.

If requested user is not in the local database, pod URL is extracted from the Diaspora ID. Then check that the remote pod supports the Web Finger protocol. Web Finger is a protocol that aims to provide information about people by their E-mail addresses. To check whether that remote pod support the web finger protocol it appends a standard location called the host-meta route to get the Uniform Resource Locator (URL): e.g., http://otherpod.com/.well-known/host-meta. Route is part of the web finger standard which is the basic way to ask a server whether or not it supports web finger.

After accessing above URL it returns an XML. This XML document contains information of how to construct the query for user discovery in that remote pod.

Then according that instruction it generate a query URL and request user information from the remote pod. Here is an example for query URL: https://otherpod.com/webfinger?q=saman@otherpod.com

This query will returns XML with basic information about the user, and contains links to find more detailed information. The hcard location of the profile is extracted from that XML in order to retrieve the profile information. An hcard is a standard, structured way to represent profile data in HTML. Then accesses the hcard location, and gets back HTML with additional profile details for the remote user.

Finally, having searched for the user and then retrieved his/her hcard, local pod extracts the profile details and saves them in local database.

2.4.3. Information Discovery Mechanisms in Distributed Networks

Today many peer-to-peer systems [24] are sharing huge volumes of data. The usability of these distributed systems depends on effective techniques to find and retrieve data. Those systems are using different information discovery mechanisms to find information within the system. When Implementing Diaspora information discovery feature, we have to use an efficient information discovery mechanism to deal with that.

Iterative Deepening

In iterative deepening search a depth limited search is run repeatedly. Increase the depth limit with each iteration until the query is satisfied or it reaches the depth of the shallowest goal state. This is equivalent to multiple breadth first searches which were initiated with successively larger depth limits. To implement iterative deepening method there should be a system wide policy that tells at which depths the iterations should occur. If it look likes p = {*a*, *b*, *c*}, then source node initiates a breadth first search of depth *a*. Nodes at depth *a* process the message and send response messages to the source node. Source node does nothing if query satisfied. Otherwise source node starts the next iteration at depth *b*. To start that iteration source node send a resend message to the nodes at the depth with a TTL of a. They simply forward the query message to its neighbors. The process continues in a similar fashion to the other levels of the policy. That is the procedure in Iterative deepening method which is used to search for relevant queries. Time elapsed to get the relevant data is small if it finds relevant nodes within initial iterations, that containing useful data. Otherwise response time will be large because of the time taken by multiple iterations.

Directed Breadth First

In directed Breadth First (BFS) Search the source node forwards query message to a selected subset of nodes. Nodes who received query messages forwards the messages to its all neighbors as BFS. To forward the query to a subset of neighbors, source should maintain a table of quality of results of its neighbors. Selecting neighbors there are few heuristics to help. Then source can forward query messages to the best neighbors using those heuristics.

- Select the neighbor who has returned highest number of results for past queries.
- Select the neighbor who has returned response messages that have taken lowest number of hops. It represents that node is close to the nodes containing useful data.
- Select the neighbor who has received largest number of messages from source node. It represents that the neighbor is stable and can handle large flow of messages.
- Select the neighbor with a shortest message queue. It represents that the neighbor is not dead.

Using those heuristics source node can select the best subset of its neighbors to forward the query message. This method will reduce the cost of the system by reducing number of nodes who process the query.

Local Indices

In Local Indices method, each node maintaining an index over the data of all nodes within r hops by itself, r is said the radius of the index. So when a node receives a query message it will process the query on behalf of the all nodes within that r hops. Likewise few nodes can process the query and search for data of many nodes.

There should be a system wide policy which specifies the nodes at which depth should be processed the query. Nodes at depths which are not included in the policy simply forward the query to the next depth without processing them. To maintain indices at each node an extra step needed when nodes leaves, join and updates its data. When a node joins it sends a join message with a TTL of r containing metadata over its collection. When others got that join message they also send a join message containing metadata over their collections. When a node leaves the system other nodes that index its nodes collection will remove its metadata after a timeout. When a node updates its collection it will send a simple update message with a TTL r containing the metadata of the affected data. Nodes that receive that message will update their index.

Distributed Caching and Adaptive Search

Distributed caching and Adaptive search mechanism [25] performs very efficient role in searching information. Mechanisms like "Uniform Index Caching" query results are cached in all peers along the inverse query path. Hence it creates large number of duplicated and unnecessary caching among neighboring peers. But this mechanism "Distributed caching and Adaptive search" prevents such unnecessary things and makes searching process very efficient. It distributes caching results among neighboring peers and forwards query messages to the peers with the high probability of providing the desired cache results. All peers within a network randomly choose an ID from a certain range and when connecting new peers into an existing system they also select an ID randomly from a certain range [0-M]. If peers satisfy below equation, only those peers cached query response.

$$\text{Peer Group ID} = \text{hash (Query) mod } M$$

There are many request forwarding algorithms [26]. When forwarding query messages we can use them according to the importance of their usage. Those algorithms were,

1. Random – Peers does not store any information about the neighboring peers and forwards query message to the randomly selected peers.
2. Learning-Based – Peers store information about the other nodes who responded for requests in the past. Hence query messages forwards to the peers who have answered similar requests previously. The query message forwards to the randomly selected neighbors, if there were not such neighbors.

3. Best-Neighbor – Peers record number of responses received by each neighbors without recording type of response. So forwards query messages to the peer who has answered largest number of queries.
4. Learning-Based + Best-Neighbor – More like to the Learning-Based strategy. When peers cannot identify such experiences, then forwards query messages to the best neighbor.

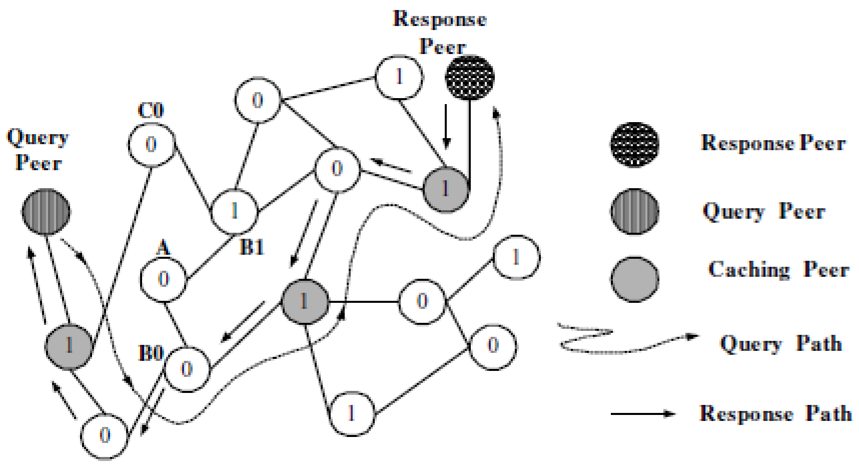


Figure 7 - Cache strategy of distributed caching and adaptive search [25].

In this searching protocol it forwards query messages to the neighbors in a way same as Learning-Based. Like query responses caching, query forwarding also restricted to matched peers who have satisfied above equation. If some peers do not have peers with matching IDs, then forwards query messages to the peer who has the highest connectivity degree with it. So query forwarding will not be blocked at early. Above figure represents the way that Distributed caching and Adaptive search works. With the time this protocol will accumulate responses which are lexicographically closed into the same node. This may also increase responsiveness of the system gradually. Concerning with other mechanisms, there are few advantages in this mechanism, for example:

- Reduce the number of nodes that process the query - When query messages propagated through the network nodes spend processing resources to forward and process them, and bandwidth to send and receive query messages and responses. Those are the main cost of queries. By forwarding query messages intelligently to appropriate peers with matching IDs, those costs can be reduced.
- Quality of results - Enhancement of an effective search mechanism will increase the quality of results which were expected by the user. The size of the total result set for a

requested message should be sufficient. With this mechanism it will be satisfied. And also it will take an acceptable time to get results, the time that has elapsed from when the query is first submitted by the user to when user receives the results set.

- Reduce number of query message count within the system - When forwarding query messages, peers forward them to the peers with the high probability of providing relevant results. Unlike in Gnutella networks which have flooding techniques, this mechanism reduces number of query message count within the system.
- Reduce replication within peers by caching responses in selected peers - Unlike UIC [25] mechanism this mechanism uses distributed caching. Query responses cached within peers which have matched group Ids relevant to the queries. All the peers in the reverse query path will not cache the responses and reduce data replication.

Multiple Sequential Search Chains Algorithm

Distributed system contains many computers interconnection each other's. Nodes (computers) in the distributed systems are divided into groups and these groups are called chains hence each chain contains list of nodes [27]. Each chain performs a serial search, the nodes in the chain being visited in turn, while different chains are processed in parallel.

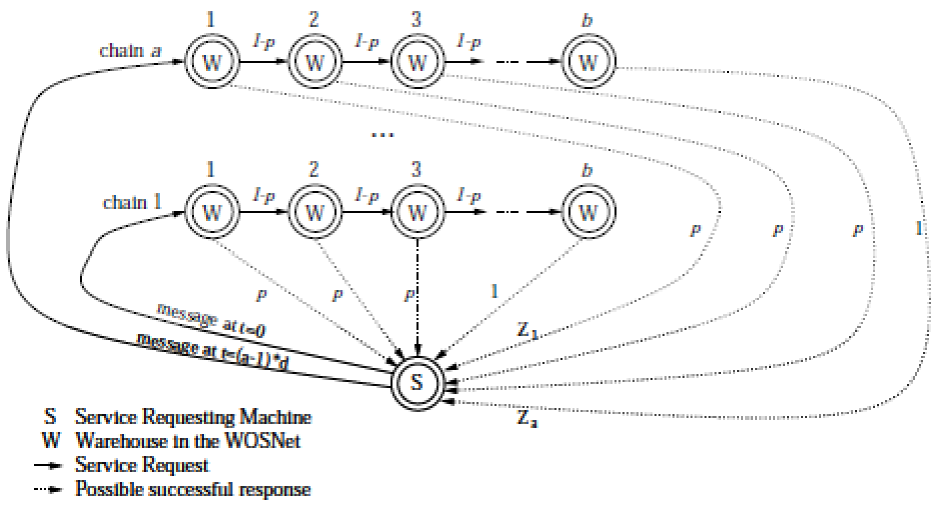

chain a
1
2
3
b
1-p
W
chain 1
p
1
message at t=0
message at t=(a-1)*d
S
Z1
Za
S Service Requesting Machine
W Warehouse in the WOSNet
Service Request
Possible successful response

Figure 8 - Graph representation of multiple sequential search algorithm [27].

There are many factors which determines the performance of this algorithm, they are number of chains, number of nods in the chain, the average transmission time of a message from one node to another, the average processing time of each node, the delay between the transmission of two message chains by the originating node, the probability that the requested information is available at node, response time of the chain. In order to get the best out of this algorithm it is better to perform some experiment and find out best number of chains for the system which will result in better search performance.

Use of Social Network Relationship Nature to Improve the Distributed Search

A social network is a social structure made up of a set of individuals, organizations and a complex set of relationships which ties these individuals, organizations. People develop relationships with other people in different contexts and they use these relationships to find information or services appropriately. These relationships may be differ respect to their strength. Some have wide social network with relatively weaker relationship, and others have narrow network of strong friendships. This nature of relationship in social network can be used to improve the information discovery in distributed network [28].

Like in social network, list of peer (respectively list of friend in social network) can be kept according to some criterion. Each peer can have different criteria and list of peers respective to those criterions. Peers should categories their own shared information according to criteria as above. Peer can learn another peer's interest from keeping track of peers who respond to a query in given category. In this way one peer can categories other peers and update its peer lists. When searching information search query is propagated through list of peers who has the highest probability of meeting the required information.

A peer attaches a strength value to each relationship with other peers from its peer-list in each category. The relationship strength between peers A and B respect to category C is equal to number of interactions by peer B to queries in category C issued by peer A divided by the total number of queries issued in that category by peer A. When relationship strength is high between two peers respect to particular category, then that peer may have higher probability of containing information. Hence the model of relationship strength can be used to improve the information discovery because it routes search query to the peer who has better probability of containing the search information.

2.4.4. Trust Management in Online Interactions

Today we cannot completely trust entities that mediate our online interactions. There should be a proper mechanism to ensure trust in online interactions. In a social network when we searching about something, we have to get information from many servers. So there should be a clear mechanism to ensure trust. Some traditional network security mechanisms are incomplete in their function to manage trust. Firewalls and cryptographic algorithms are currently using hard security mechanisms, with all or nothing property. Some mechanisms used centralized protocols, but with a large distributed system trusted intermediary can never be a good enough recommender for everyone. Recommendation protocol [29] may be a good protocol to manage trust in a distributed system.

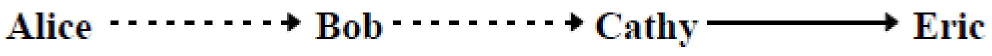


Figure 9 - Example: can Alice trust Eric.

In Figure 9 dotted arrow represents Recommendation Trust that means someone trust someone’s recommendations about something. Line arrow represents Direct Trust that means someone trusts someone. When Alice wanted to find Eric who does not have a connection with Alice, he asked from the Bob because he trusts recommendations of Bob. Bob also have not a connection with him and he asked from Cathy, because he trusts Cathy’s recommendations. Cathy knows Eric and he reply to Bob, then Bob to Alice. In that way it ensures and manages the trust between requester and the replier. This protocol can be applied in our information discovery feature on Diaspora. Then it will manage the trust between servers which runs Diaspora software. Hence it will be a good enhancement to the existing system and to the people who concerns about their privacy.

3. Problem Statement

Although there are many distributed social networks, none of them has a functional API that allows third-parties to create apps for the users. The challenge when implementing such an API is authentication and authorization of apps. Due to nonexistence of an API, Diaspora users are unable to experience third-party applications and native applications. Existing standards like OAuth are designed assuming a centralized resource owner; therefore, supporting tools like libraries and frameworks designed for those standards are not directly usable.

Purpose of this chapter is to present our project goals, objectives, and deliverables which we planned at early stage of the project. Section 3.1 states problem that we plan to address, Section 3.2 presents the goals of our project. On section 3.3 our objectives to achieve goals are presented. Finally section 3.4 presents the expected deliverables at the end of the project.

3.1 Goals

Our goal is to extend Diaspora DSN by adding following two useful features:

- Implement an API for Diaspora to authenticate and authorize third-party apps into Diaspora.
- Implement an information discovery app for Diaspora using the API with real name and location based user searching.

3.2 Objectives

Initial project objectives are:

- Design and implement a mechanism to authenticate and authorize third-party apps in DSN environment.
- Define an API specification.
- Design and implement API for Diaspora DSN.
- Design and implement information discovery app for Diaspora using the Diaspora API functions.

3.3 Deliverables

Our key deliverables are:

- API for Diaspora DSN

  This is a REST API that provides HTTP endpoints for applications allowing them obtain various information from the Diaspora pod. First apps must be granted authorization from the user and the pod must be able to verify the authenticity of apps. An Authenticator module is implemented with the API for this purpose. A document specifying the functionalities of the API and providing guidelines for developers is prepared.
- Information discovery application for Diaspora DSN

  This is a web based application which enhances the user experience in discovering other Diaspora users. Currently to search other users across pods Diaspora users have to know the Diaspora identifier of that user. With this application searching users with real names and other attributes like location will be possible. Further this demonstrates the use of the API.

Software requirement specification of API and the information discovery app can be found as appendix I.

4. Diaspora API Design and Implementation

This chapter describes about the design and implementation of the "DAuth" authentication and authorization model, and the Diaspora API. When designing an API there should be a proper mechanism to expose user's data securely with third-party apps. For that purpose we implemented an authentication model called DAuth which is functioning as the OAuth protocol. Then third-party app users will be able to use our RESTful API through this authentication model.

We discussed about the design and implementation of Authentication model in Section 4.1 including OAuth protocol, implemented DAuth protocol, token management database structure, and manifest file structure. In Section 4.2, we discussed about the API architecture, API class diagram and sequence diagrams. Introduction to the API specification is available at Section 4.3.

4.1 Authentication Model

4.1.1 Public and Private Data

When creating applications that only use public data, application does not need authorization from the end User. In these types of applications authorization occurs between two parties: an application (the Consumer) and the public data source (the Service Provider). The public data source can be a Web service or Web feeds such as RSS. Most Web Services offering public data require authorization, whereby an application and the Web Service exchange keys before public data can be transferred.

To access private data, an application is required to get the end User's authorization. In this type of applications, authorization occurs between three parties: the end user (User), the application (the Consumer), and the private data source (the Service Provider). End Users choose whether to give that application access to their personal data.

4.1.2 OAuth for Distributed Resource Owners

OAuth is an open protocol enabling an application to access end user information from a Web service when the application is authorized by the end user. The end user's information is securely transferred without revealing the identity of the user. For example, end users who want to run a photo sharing application on their profile pages need to allow the social network site to share personal data with that application. OAuth allows these end users to anonymously grant the photo sharing application access to their profile information. OAuth verifies that requests by an application are actually coming from that application and it has permission to access potentially sensitive data for users.

OAuth providers like Facebook, Google, and Twitter use a central registry of apps. The developer of the app registers the app on this central registry providing details of the app and obtains a consumer key and a consumer secret. The developer is responsible to keep this information secret. With every communication the app makes with the server (OAuth provider) this information is sent. Server uses them to authenticate the app and checks with the central app registry to get information about the app.

This approach works well for centralized servers (resource owners). But the requirement of an app registry makes it less suitable for a distributed resource owner. Apps need to be registered for each server with each server keeping its own registry of apps. Obviously above approach is not feasible. There could be many servers and new servers are added to the network with time. It is tedious for apps to be registered in each server. Another approach is keeping a central registry that every server communicates with. This is a possible solution but there are many complications associated with it. Secure communications need to be implemented with the central registry. It will have to handle multiple requests from many resource owners and apps. Further, many users in the distributed system does not appreciate using a centralized approach to solve this problem.

In the case of Diaspora the discussions about implementing an API has been carried out for about two years. In those discussions above solutions are discussed, and for reasons given above a better solution is needed.

4.1.3 DAuth

In centralized approach, OAuth protocol authorize an application by giving Consumer key (API key). But in distributed approach OAuth cannot be used since there is no central server which keep tracks on third-party applications. In DAuth protocol is an utilized version of the OAuth protocol which can be used to authorize third-party applications in a distributed systems.

In the DAuth protocol we introduce to Diaspora the trust anchor is shifted to the developer. The user is given a link to the profile of the developer. The profile is authenticated so the user can trust the information she is given in the profile. User use that information to audit the trustworthiness of the app. based on that the user can decide to whether to allow the app to use her information. The major difference in DAuth from OAuth is the user is asked to trust the developer of the app rather than the app itself.

In order to provide a trustable link of the developer to the user, requests from the app to the server have to be verified. To do this the information on the request has to be digitally signed using an asymmetric key pair.

Diaspora implementation assigns a key pair to each profile. The private key is stored in the database of the pod that contains the Diaspora account of the relevant user. It is treated as a secret and can't be obtained. The public key can be obtained from the public web finger profile of the user. This profile too is usually hosted in the pod where the user created his Diaspora account. Using this key pair information can be digitally signed enforcing non-repudiation.

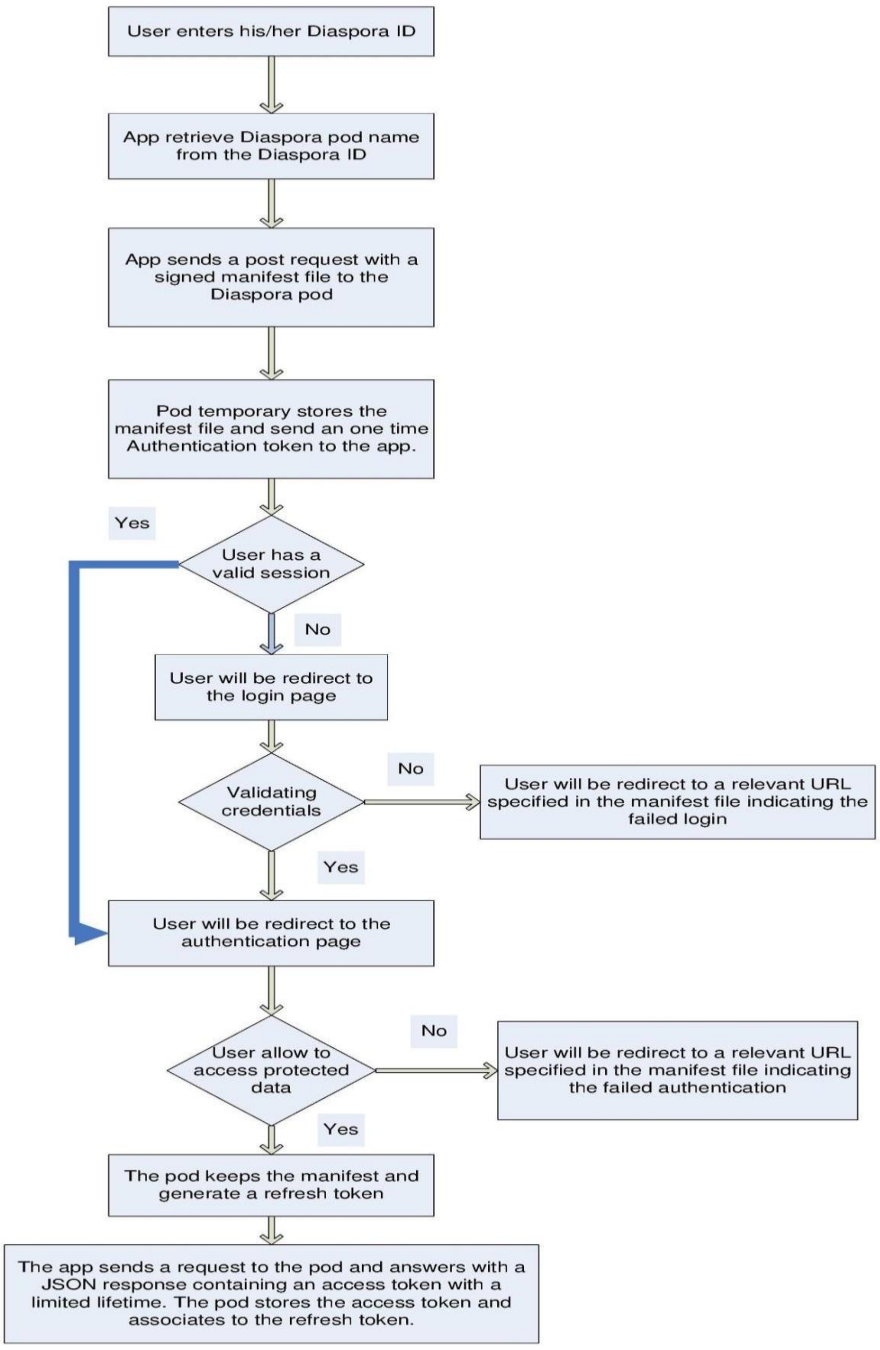

User enters his/her Diaspora ID
App retrieve Diaspora pod name from the Diaspora ID
App sends a post request with a signed manifest file to the Diaspora pod
Pod temporary stores the manifest file and send an one time Authentication token to the app.
Yes
User has a valid session
No
User will be redirect to the login page
No
Validating credentials
User will be redirect to a relevant URL specified in the manifest file indicating the failed login
Yes
User will be redirect to the authentication page
No
User allow to access protected data
User will be redirect to a relevant URL specified in the manifest file indicating the failed authentication
Yes
The pod keeps the manifest and generate a refresh token
The app sends a request to the pod and answers with a JSON response containing an access token with a limited lifetime. The pod stores the access token and associates to the refresh token.

Figure 10 - API authentication and authorization process

4.1.4 Token Management Entity Relationships

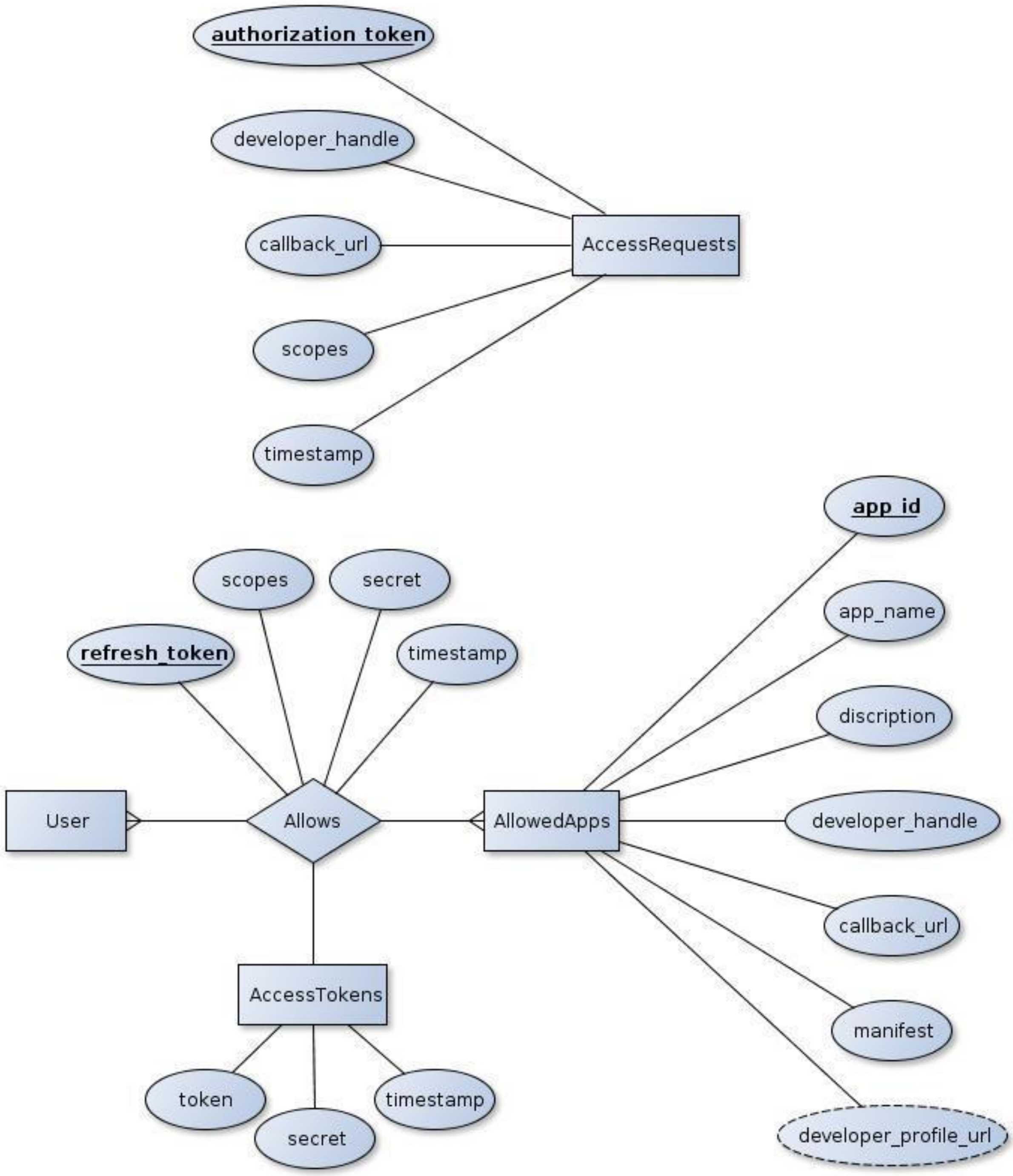


Figure 11 - E-R Diagram of the token management model.

Figure 11 gives an overview of entities in the token management. Their relations and operation is explained below.

AccessRequests

authorization_token :primarykey

developer_profile

callback

scopes

timestamp

- When the app sends a request to the “hostname/dauth/manifest” the pod will verify the manifest file. Then it will create an authorization token for this request. This will be a random string. Then store callback URL, scopes and a link to the developers’ profile. AccessRequests table will hold this data.
- Then the user is redirected to “hostname/dauth/authorize/authorization_token”.
- When user requests this URL the AccessRequests table will be accessed and a link to the developer's profile and requested scopes will be displayed to the user. user can adjust scopes in this page and view the developer profile

RefreshTokens

token :primarykey

secret

timestamp

app_id

user_guid

scopes

AllowedApps

app_id

app_name

discription

app_home_page_url

dev_handle

manifest

callback

- The user allows the app

- A refresh token is generated by the pod and stored in the RefreshTokens table. Other data in the table is taken from the manifest file and stored.
- An entry to the AllowedApps table is entered. This data is also taken from the manifest file.
- Now the entry in the AccessRequests table can be deleted.

AccessTokens

refresh_token :primarykey

token

secret

timestamp

- The callback in the AllowedApps table is used to inform the app of new access tokens.
- The app uses the Refresh token it is granted to get access tokens (with an expiry date).
- User can view all the apps allowed to use her information. This information can be found in the RefreshTokens table.
- When the user revokes a token the entry is deleted and the app is informed using the callback URL.
- AllowedApps table is not altered as other users maybe still have allowed this app.

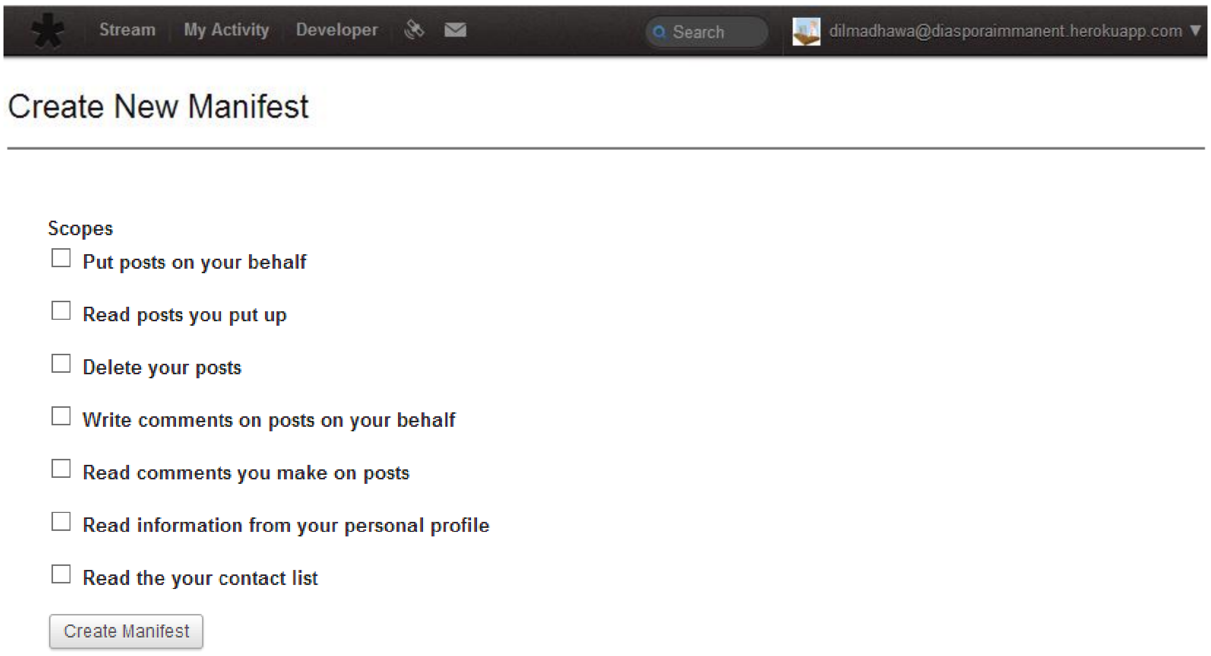
Manifest

Manifest file is responsible for keeping details about the third-party app. With the implementation of the API, there is a place for app developers to create their manifest file in a given format. Developers can enter following details manually by login in to their Diaspora accounts

- App name – Which contains the name of the third-party app
- App description - Short description which describe about the functionality of the app
- App version - Version number of the app
- Callback URL - URL which is used to inform the app of new access tokens
- Redirect URL - URL of a page in the third-party app which loads after completing authentication
- Scopes - Areas that user is going to expose to the third-party app

Other than above details manifest file contains its signed JSON web token using developer's private key.

Figure 12 shows manifest creation user interface for developers.


Stream My Activity Developer Search dilmadhawa@diasporaimmanent.herokuapp.com

Create New Manifest

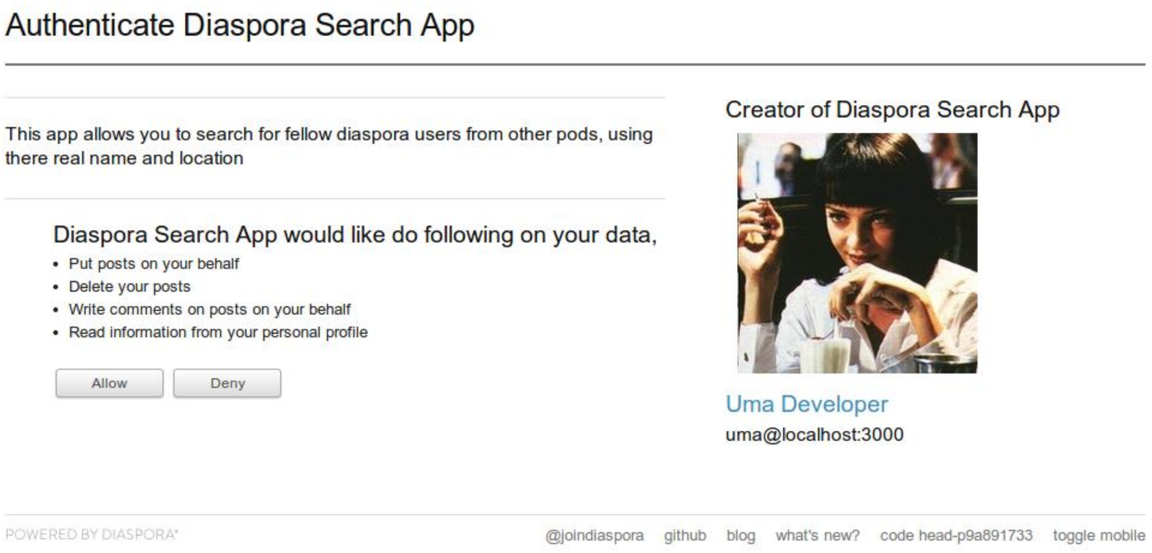
Scopes

- ☐ Put posts on your behalf
- ☐ Read posts you put up
- ☐ Delete your posts
- ☐ Write comments on posts on your behalf
- ☐ Read comments you make on posts
- ☐ Read information from your personal profile
- ☐ Read the your contact list

Create Manifest

Figure 12 - Manifest creation for developers.

Following figure shows 13 authentication page where user can give accept or deny an application.


Authenticate Diaspora Search App

This app allows you to search for fellow diaspora users from other pods, using there real name and location

Diaspora Search App would like do following on your data,

- Put posts on your behalf
- Delete your posts
- Write comments on posts on your behalf
- Read information from your personal profile

Allow Deny

Creator of Diaspora Search App

Uma Developer
uma@localhost:3000

POWERED BY DIASPORA* @joindiaspora github blog what's new? code head-p9a891733 toggle mobile

Figure 13 - User give authority to App.

4.2 API Architecture

We have developed an API for the use of third-party app developers of Diaspora. API is program which let the user to use methods of an application from outside the application. Here we created a REST API which allows CRUD (create, read, update, delete) operations.

The architecture of the Diaspora API is shown in the figure.

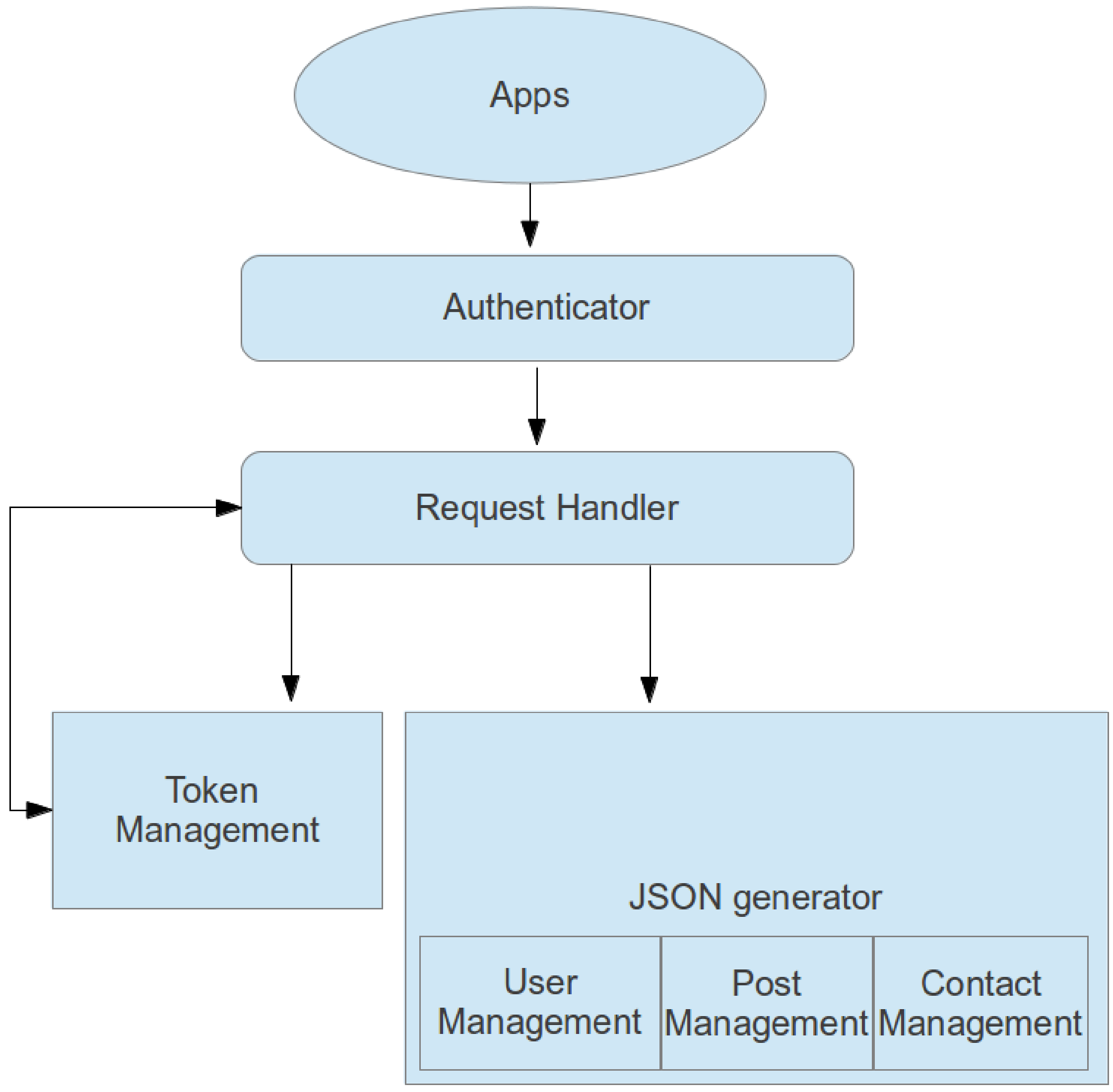


Figure 14 - API Architecture.

4.2.1 Class diagram

Diaspora API consists of set of classes which related to the several scopes like user profile, comments, posts etc. Users can access protected resources by calling API methods with relevant parameters. The class diagram of the Diaspora API is shown in Figure 15

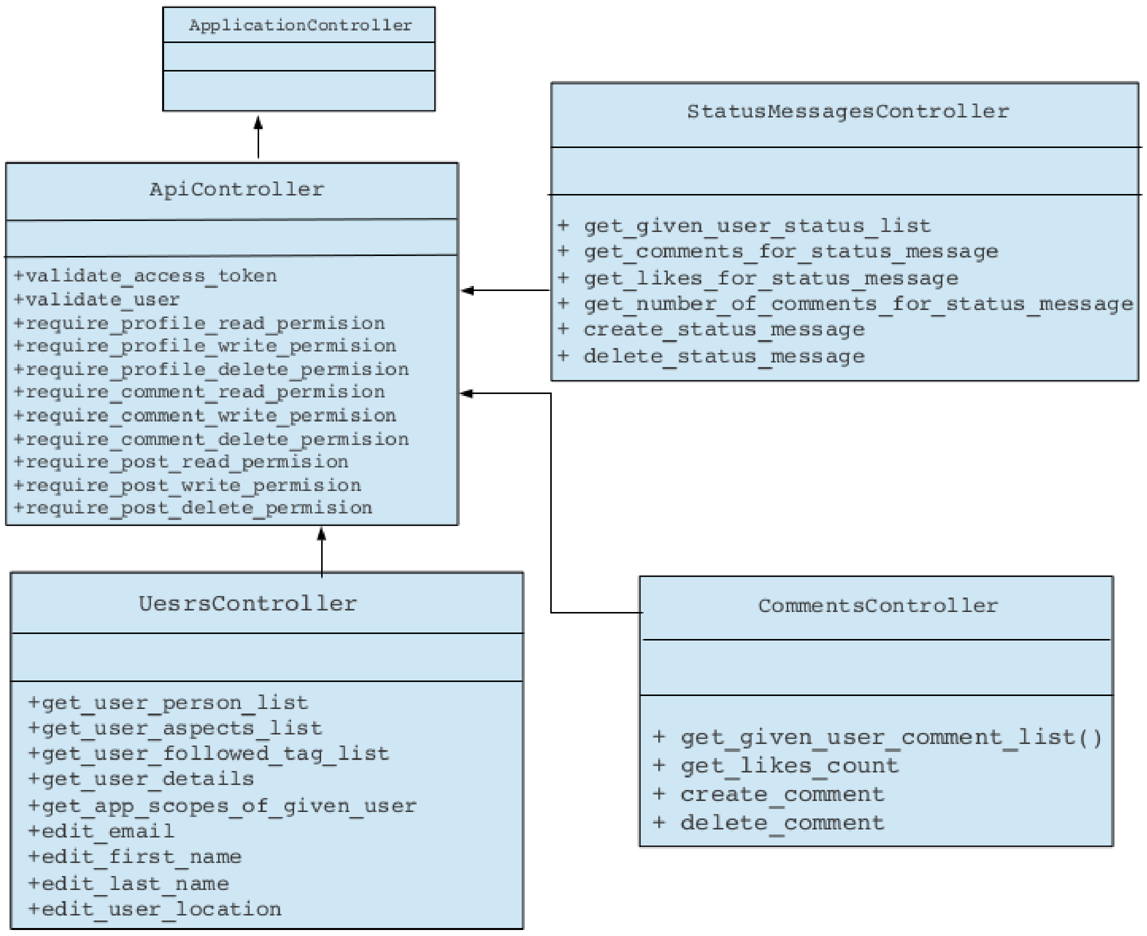

ApplicationController
ApiController
+validate_access_token
+validate_user
+require_profile_read_permision
+require_profile_write_permision
+require_profile_delete_permision
+require_comment_read_permision
+require_comment_write_permision
+require_comment_delete_permision
+require_post_read_permision
+require_post_write_permision
+require_post_delete_permision
StatusMessagesController
+ get_given_user_status_list
+ get_comments_for_status_message
+ get_likes_for_status_message
+ get_number_of_comments_for_status_message
+ create_status_message
+ delete_status_message
UesrsController
+get_user_person_list
+get_user_aspects_list
+get_user_followed_tag_list
+get_user_details
+get_app_scopes_of_given_user
+edit_email
+edit_first_name
+edit_last_name
+edit_user_location
CommentsController
+ get_given_user_comment_list()
+ get_likes_count
+ create_comment
+ delete_comment

Figure 15 - API class diagram.

api_controller

This class is responsible for giving controlled access to the protected resources. User have to declare scopes which he likes to share with the application when he is going to registering to a new app. Hence it is tested through this class at the time app accessing the Diaspora API. There are methods to check whether a user have permissions to read data, delete data and update existing data. When accessing resources user should have a valid access token. It is also checked within this class. There is another method to check whether the user is a valid user with valid access token which is assigned to that particular user.

users_controller

This contains API methods which related to the user details. Accessing profile details, edit profile details, update existing profile details, accessing user's private data as well as public data and user’s scopes related to each allowed applications can be access using its API methods.

comments_controller

This class contains API methods related to the comments. Users can access details of their comments, delete comments and create new comments using its methods.

status_messages_controller

This class contains API methods related to the users’ states messages. Users can access details of states messages, delete states messages which are published by them and create new states messages using its methods.

4.2.2 Sequence diagram

Sequence diagram in Figure 14 illustrates the sequence of the requests and responses for API methods. Figure 16 illustrates the sequence of “get_user_person_list” API method.

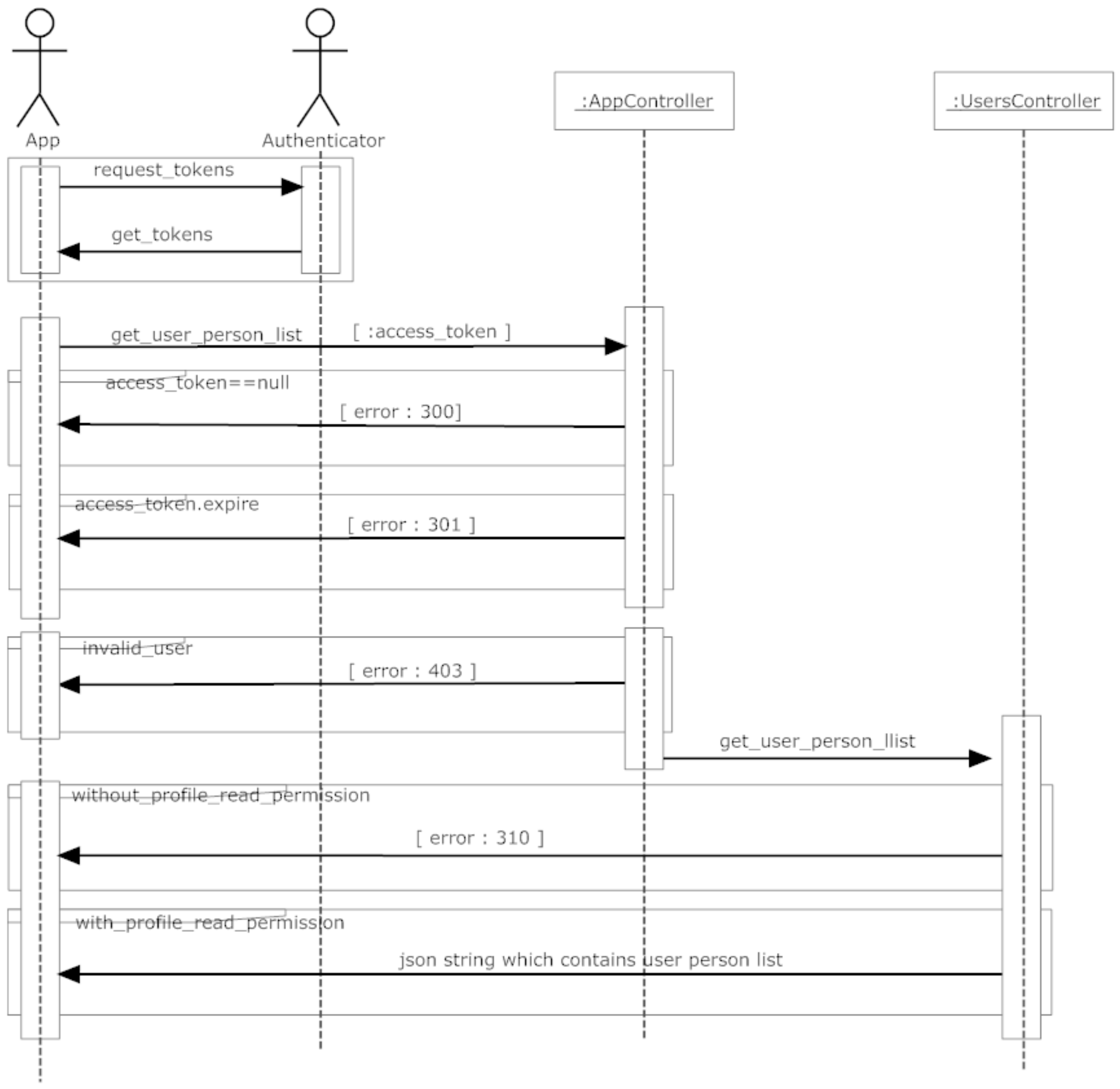


Figure 16 - Sequence Diagram –“get_user_person_list” API function

4.3 API Specification

Diaspora REST API securely shared users’ information with apps. There are many API methods which can be accessed by app developers in developing their apps. With the Diaspora API we also created its API specification for the use of third-party app developers. In that document we have given,

- Description about each and every API method
- API method request URL
- Method type (GET, POST, DELETE)
- Description of the relevant parameters and their types (ex. Diaspora handle, access token)

- JSON response structure
- Error codes and their explanations.

Full API specification can be found in appendix II.

4.4 API Implementation Details

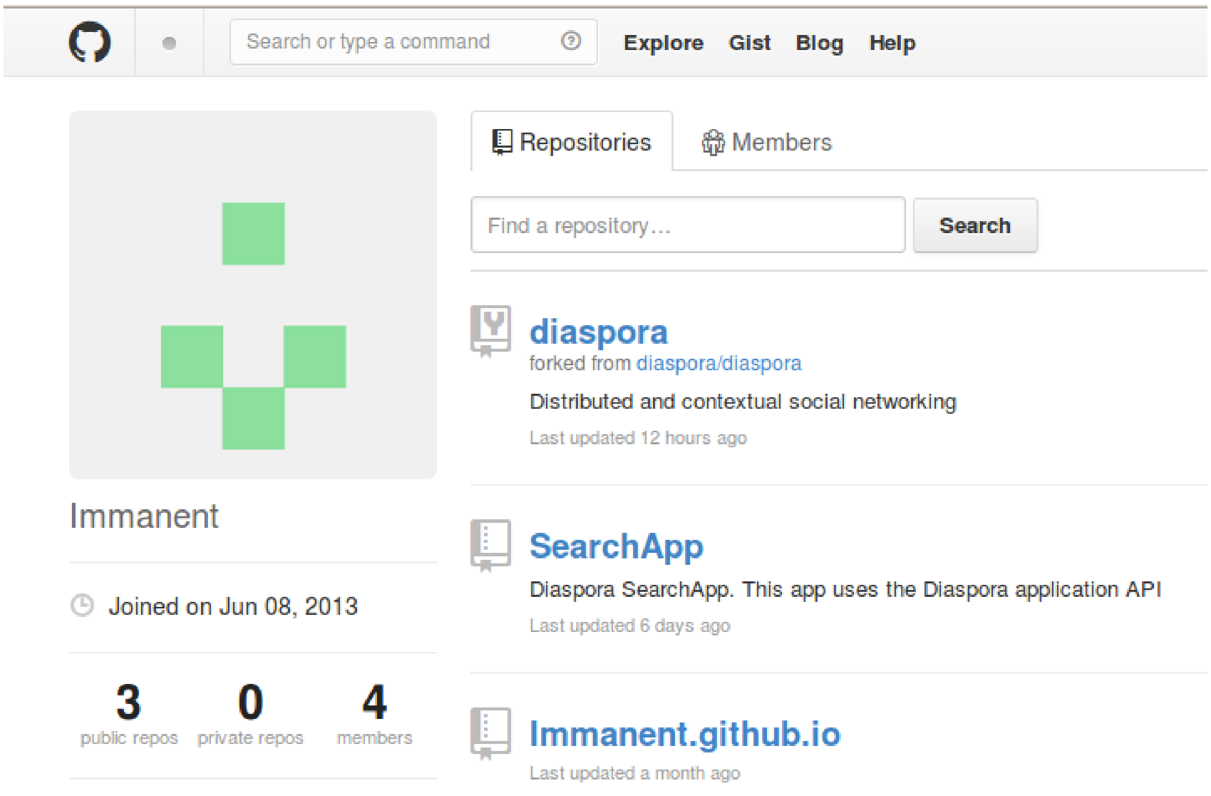
This section contains the overall project process and version control system we used. In later sections discuss the coding standards and best practices and testing we used the API implementation process.

4.4.1 Project Process

In the initial stage of our project we created the project proposal and identified the purpose of the project. We defined the final project scope after analyzing the relevant literature to our project and created the requirement specification for two main deliverables. Then in project implementation we planned each two weeks by assigning task to every member then at the end of the two weeks measured the completion of the tasks and incomplete task were forward to next two weeks.

4.4.2 Version Control

The project is version controlled using Git, and hosted at GitHub [30]. The primary reason for selecting Git and GitHub is that Diaspora project uses them. By using them it is easier to work with them and integrate the API code to the main project.



Immanent

Joined on Jun 08, 2013

3 public repos
0 private repos
4 members

diaspora
forked from diaspora/diaspora
Distributed and contextual social networking
Last updated 12 hours ago

SearchApp
Diaspora SearchApp. This app uses the Diaspora application API
Last updated 6 days ago

Immanent.github.io
Last updated a month ago

Figure 17 - Immanent GitHub repository

By using the distributed nature of Git, each member of the group keeps his own repository of the project. Our work is done on a clone of the diaspora project. Each member creates feature branches as necessary and brings in final work to the main branch named 'diaspora_api'. We used the 'organization' feature in Github to identify the collective effort. Our work is submitted to the reviewing of Diaspora community under this organization named 'immanent'.

According to the request from the diaspora developers our work is continuously reviewed by the diaspora community.

4.4.3 Coding Standards and Best Practices Guidelines

Model View Controller (MVC) software architecture is used when implementing the authentication model and the API. API is developed as a REST API, so all the responses are in the form of JSON objects**.** Views for edit manifest file and download manifest are implemented using Haml. JSON web tokens (JWT) and RS256 algorithm is used for encoding and decoding purposes.

4.4.4 Website

Figure 18 shows our project website [31]. It contains the details about our project motivation, API design and DAuth authentication model.

Figure 18 - Project website

4.4.5 Testing

We implemented API methods using Test Driven Development (TDD). In test driven development, we first write a failing test and execute it using the Terminal. That failing test results are represented by the color red. We then implemented code to get the test to pass and its results represented by the color green. Then we refactored the code by changing its form without changing its function.

To test API methods we should have some preconditions in our test database. In rails there is a default feature called Fixtures to keep objects which contain relevant preconditions that then get loaded into the test database when we run our test suite. Because it has some drawbacks in maintaining and very quickly get out of control we used Factory Girl [32] gem to create our preconditions.

Factories allow us to quickly build the data we need for each test. Building of relevant data for each test case can be done before each method. It becomes easy to manage and keep track of test data. We tested all the JSON responses of each method including error codes. All the models are tested for correct validations.

5. Diaspora Search Application Design and Implementation

This chapter discusses about Diaspora search application design considerations which we thought at the design phase. In this chapter we discuss Diaspora search application of our project in terms of its architecture and design. We discuss about the high level architecture using an architecture diagram and in detail with the use of 4+1 architectural view model with various different diagrams such as class diagrams, use case diagrams, class diagrams, etc.

5.1 Diaspora Search Application High Level Architecture

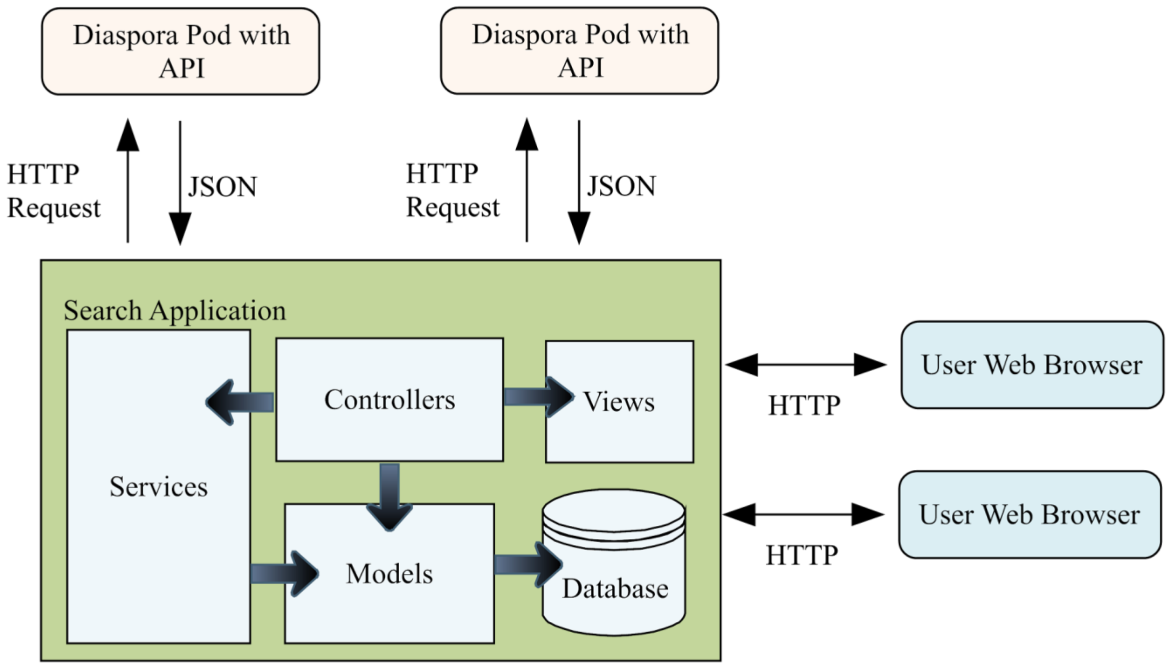


Figure 19 - High-level architecture of the search app.

In Diaspora search application high level architecture we can consider 3 main physical component. They are Diaspora pod with API, Diaspora search application and user web browser.

- Diaspora pod with API
  Diaspora search application communicates with the Diaspora pods in Diaspora social network. These Diaspora pods should have the API with we developed. Search application send http request to Diaspora pod requesting the services and Diaspora pod respond with JSON.

- User web browser

User access the search app functionalities through a web browser and it access server functionalities by HTTP request. Web browser renders the HTTP response and display pages.

- Search Application

  The web server with search application is the important component in Diaspora search application. It provides the functionality of discovering users in Diaspora social network. Like mentioned above it communicate with Diaspora pods and user web browser to provide overall functionality.

  In the design of the search application we used MVC software architectural pattern which helps separate the representation, Logic and user interactions. Hence the search application consists of models which consist of application data and logic functions; views which can be output of representation of data; controllers which mediate input, converting it to commands for the model or view; services which provide additional functionalities to controllers and models and database which stores data.

In the next section of this chapter we discuss the architecture of search application using 4+1 architectural view model.

5.2 4+1 Model – Diaspora Search Application

In this chapter we use 4+1 architectural view model describing the architecture of Diaspora search application. The views are used to describe the Diaspora search application from the viewpoint of end users and developers and system administrators. The four views of the model are logical, development, process and physical view. In addition we use ‘use cases’ to illustrate scenarios in search application.

5.2.1 Logical View

The logical view is concerned with the functionality that Diaspora search application provides to end users. UML diagrams are used to represent the logical view including Class diagram and Sequence diagram.

Class Diagram

The class diagram of Diaspora search application is shown in the figure.

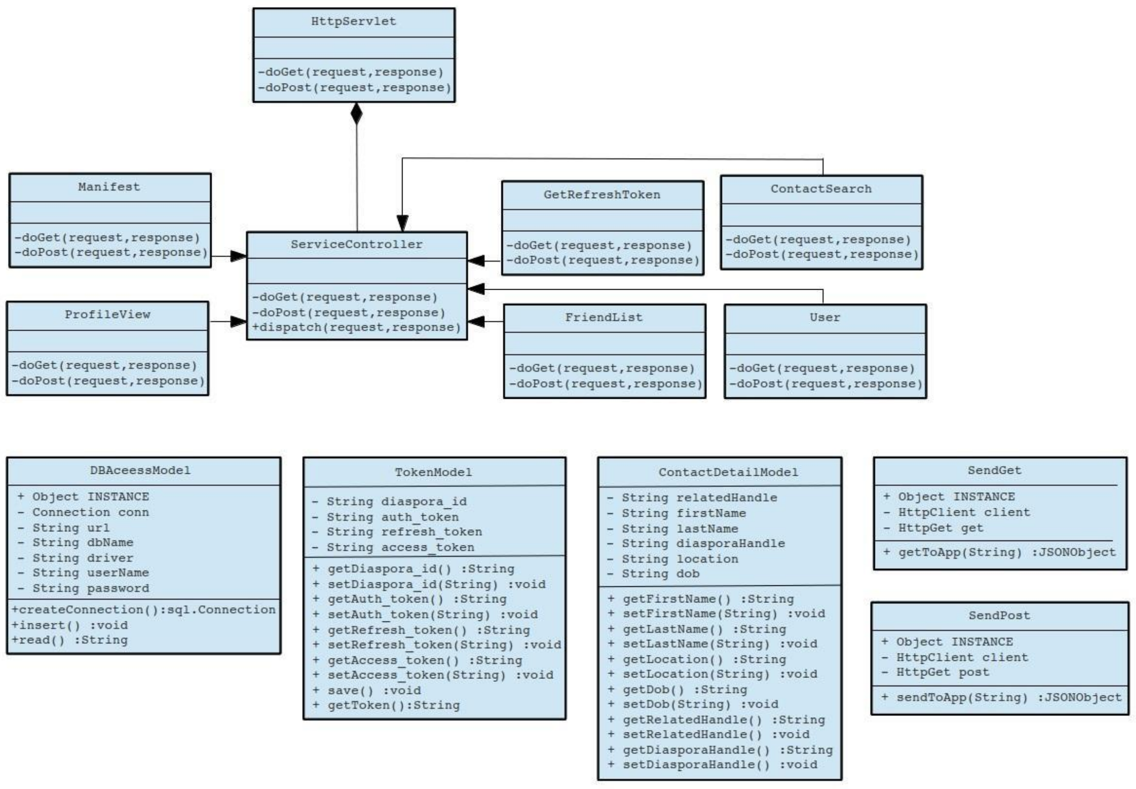


Figure 20 - Class diagram - Search app.

Sequence Diagram

This sequence diagram shows the scenario of, Diaspora user search friends using Diaspora search application. In the sequence diagram it shows how the scenario happens from application home page to search result page.

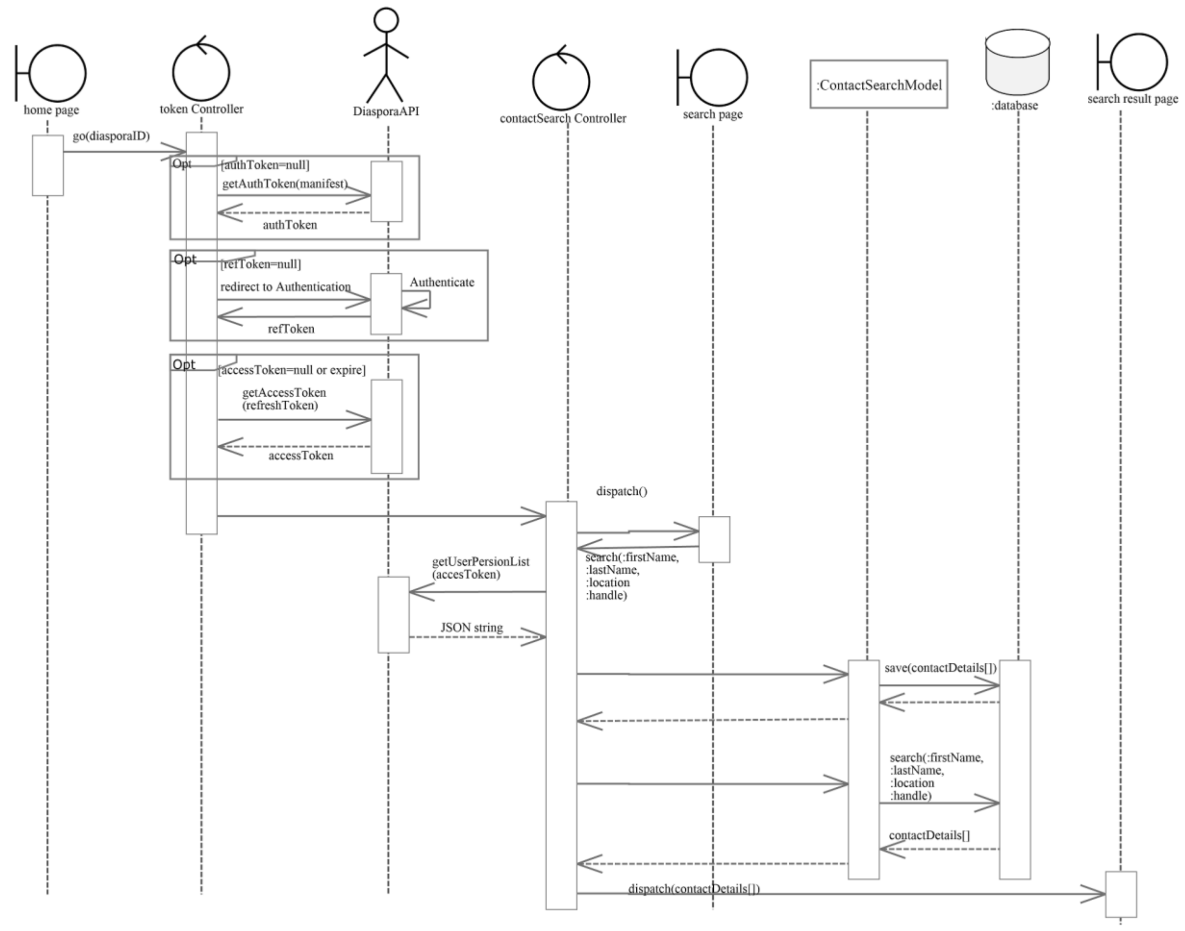


Figure 21 - Sequence diagram - Contact search

5.2.2 Implementation View

The implementation view illustrates Diaspora search application from a programmer's perspective. We use the UML Component diagram to describe Diaspora search application components.

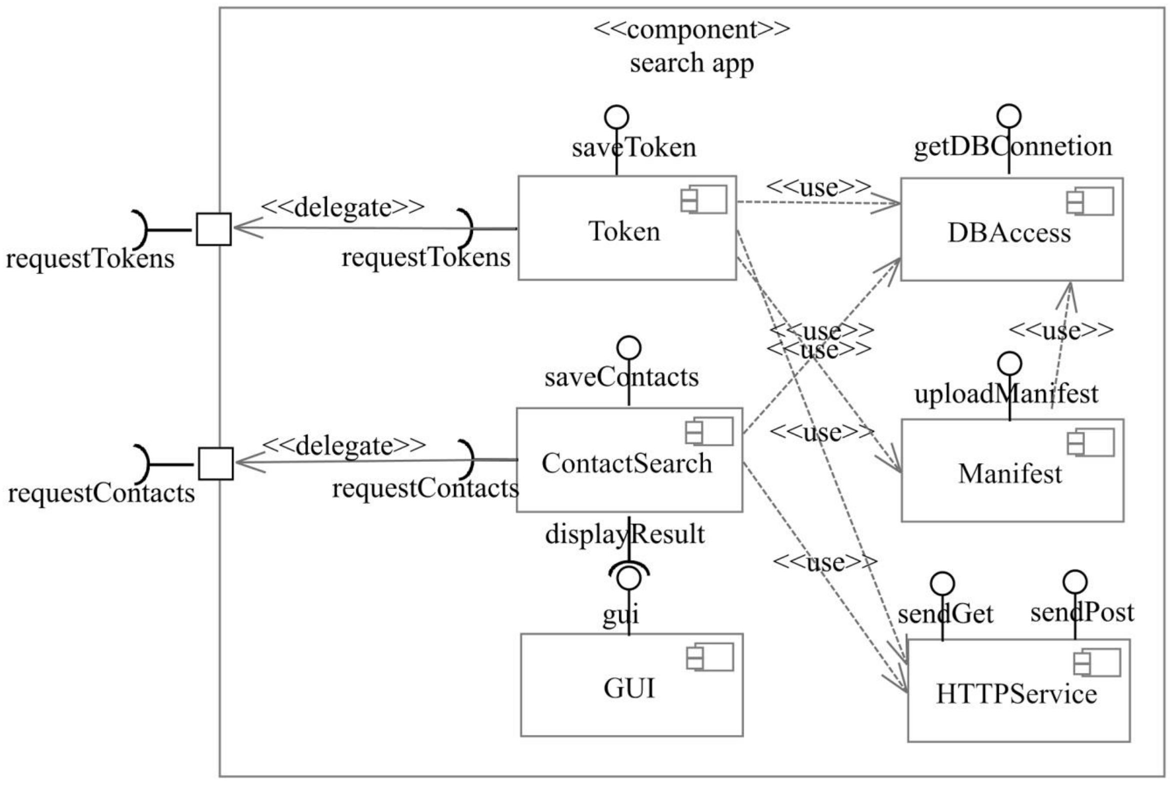


Figure 22 - Component diagram - Search app

In Diaspora search application component diagram search app is the main component and it contains Token component, contact search component, manifest component, DBAccess component, HTTPService component and GUI component.

- Search App Component

  Search application considered as the main component. It has two required interface to perform its functionalities. It requests and obtains the authentication token, refresh token and access token from the Diaspora pod. It also requests and obtains user friend list from the Diaspora pod.

- DBAccess Component

  This component is responsible for providing database connections for other components.

- HTTPService Component

  This component is responsible for providing sending GET request and sending POST request from the Diaspora search application.

- Token Component

This component is responsible for requesting and obtaining authentication, refresh and access token for users and saving them. Token component use DBAccess component, Manifest component and HTTPService component to provide those functionalities.

- Manifest Component

  Manifest component responsible for providing developer to upload the manifest file to the application. Manifest component use DBAccess component save the uploaded manifest.

- ContactSearch Component

  ContactSearch component is responsible for request friend list from the Diaspora pod and save new contacts in search application database and search the friend relevant to the user criteria and calling the necessary views display the search results.

- GUI Component

GUI component is responsible for displaying models.

5.2.3 Process View

The process view deals with the dynamic aspects of the system. We use the UML Activity diagram to describe the user search functionality of Diaspora search application.

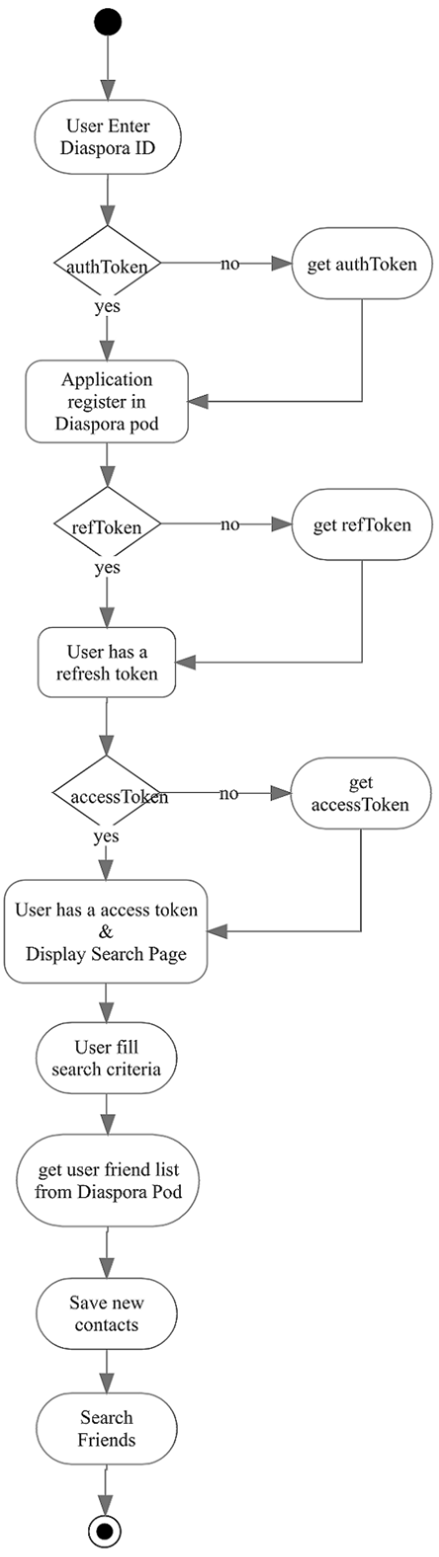


Figure 23 - Activity diagram - Search app

This activity diagram gives a detailed description of end to end of friend search functionality in Diaspora search application.

5.2.4 Deployment View

The deployment view depicts the system from a system engineer's point-of-view. It is concerned with the topology of the components on the physical layer, as well as the physical connections between these components. We use the UML deployment diagram to describe the deployment view of Diaspora search application.

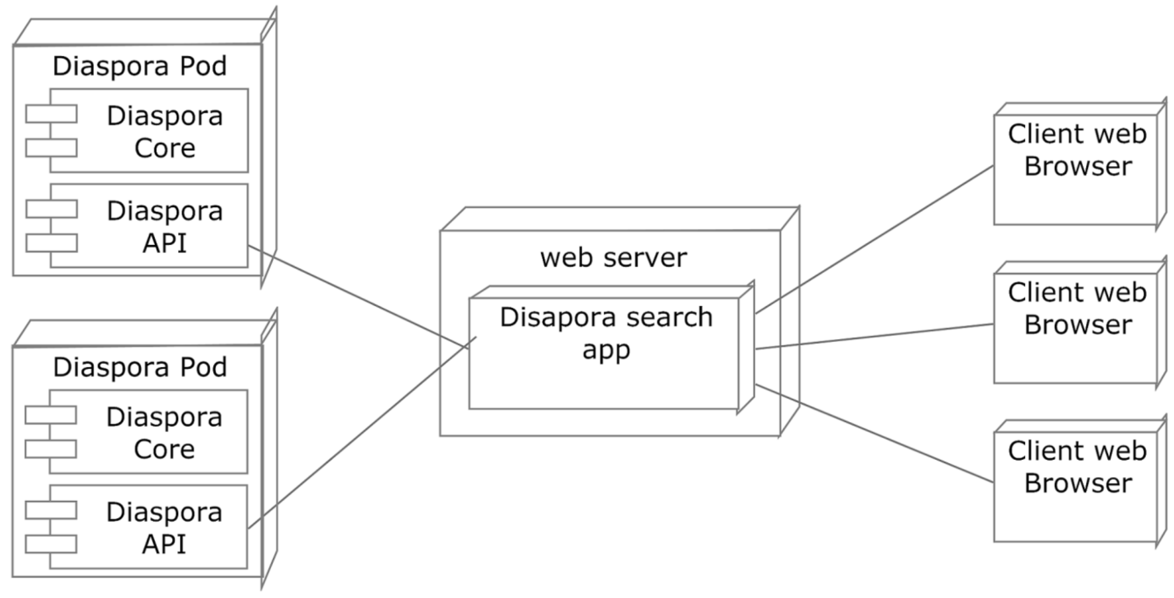


Figure 24 - Deployment diagram - Search app

The above deployment diagram depicts the deployment of Diaspora search application. Search application deployed in a separate web server and it communicate with pods in Diaspora social network through HTTP protocol. Users can access functionalities of the web application through their web browser.

5.2.5 User Cases View

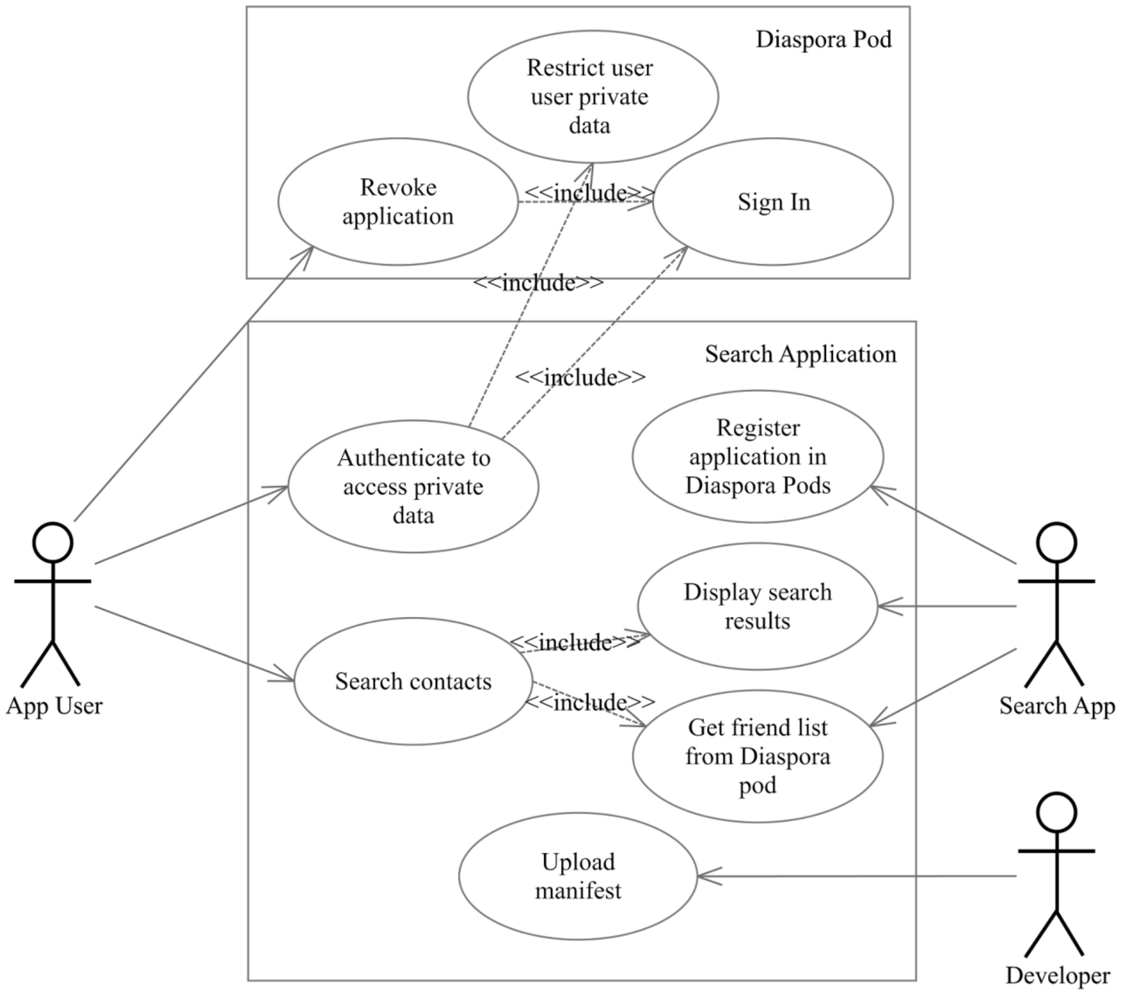


Figure 25 - Use case diagram - Search App

Use Case 1: User authenticate to access private data

Users should be able to authenticate their private data to the application. In authentication process they have to sign in to their Diaspora accounts and then they can restrict exposure of their private data.

Use Case 2: User search contacts

Users should be able to search new contacts using Diaspora search application. Search application get contact information from Diaspora pods and display the search results.

Use Case 3: Revoke application

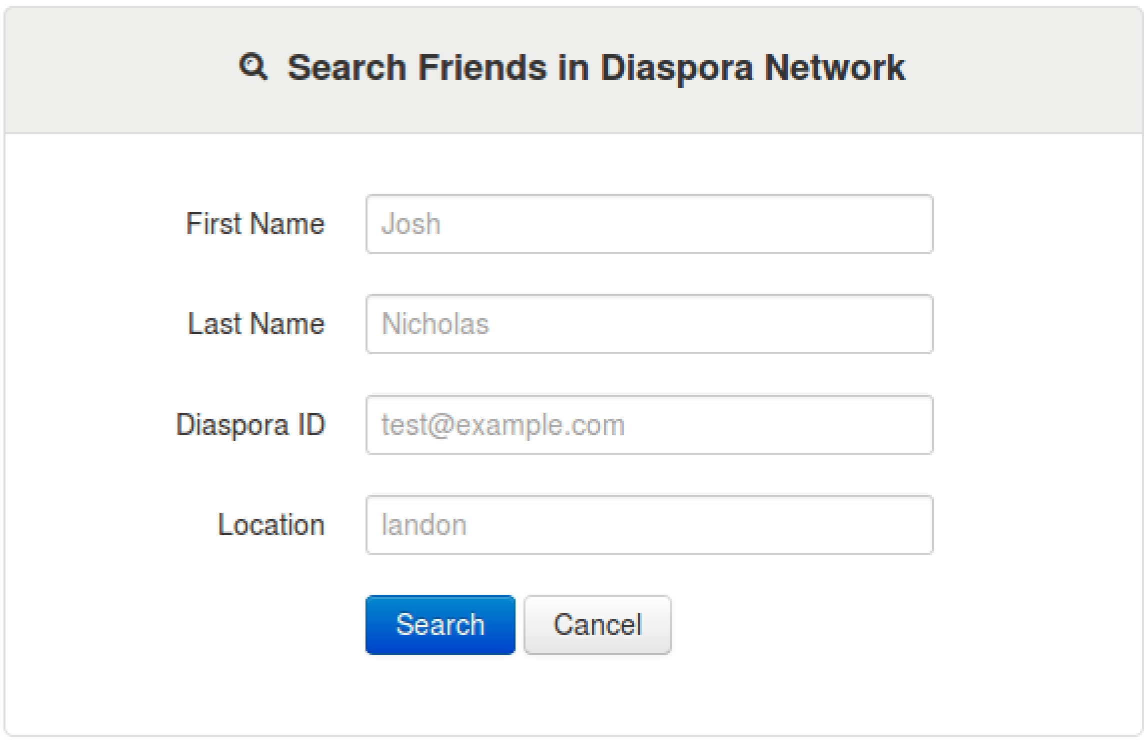
Users should be able to revoke the search application. Before revoking the application they have to sign in to their Diaspora accounts.

Use Case 4: Developer upload manifest file to the application

Developer should be able to upload the manifest file to the search application. This manifest file help when registering the application in new Diaspora pods.

5.3 Implementation Details

This section discuss the coding standards and best practices and testing we used in search implementation process.



Search Friends in Diaspora Network

First Name	Josh
Last Name	Nicholas
Diaspora ID	test@example.com
Location	landon

Figure 26 - Contact search page - Search app

5.3.1 Coding Standards and Best Practices Guidelines

Search application is developed following the MVC software architecture pattern which separates the representation of information from the user's interaction with it.The *model* consists of application data, business rules, logic, and functions. Java Server Pages (JSP) are used as *views* for the output representation of data. Multiple views of the same data are possible, such as a HTML and JSP. The *controller* mediates input, converting it to commands for the model or view. Java EE HTTP Servlets are used as controllers.

We use bootstrap front-end frame work [33] when implementing the user interfaces of search application.

5.3.2 Testing

JUnit [34] unit testing were used to test the facilities in the search application. When testing functions related to database operations DbUnit [35] is a JUnit extension targeted at database driven projects that puts database into a known state between test runs. All the model functionalities of the search app have tested.

6. Summary

Distributed social networks are one of the emerging areas in today's social network because it provides users more control on their private data. However, the popularity and usefulness of the these distributed social networks are limited due to lack of features compared to the centralized social networks such as API for third-party app developers, real name searching, online chat, etc. In our project we identified that absence of an API for third-party app development is one of the key limitations in DSNs. Hence we decided to implement an API for a selected DSN. We selected Diaspora as our DSN. Diaspora also lacks real name searching option. Users had to remember Diaspora handles of friends who they are going to searching for. Those lack features keep users away from Diaspora. Therefore, we decided to implement a searching application using API which we are going to implement at the end of our project time period. It is for search friends using their names and locations.

We decided to implement the API as a RESTful API which uses CRUD operations and represent responses as JSON objects. At the end our project time period we have implemented API for the Diaspora social network with the Authentication and Authorization model. When completing that objective we design an authentication and authorization mechanism called "DAuth", which can authenticate and authorize app in distributed environment. This is the first time such a distributed authentication and authorization mechanism is implemented on a DSN. It is similar to the OAuth protocol which is used in centralized social networks such as Facebook. We also implemented the RESTful API for the use of app developers with the API Specification document. It contains all the details about the API methods which can be used as a user manual in app development for the Diaspora social network.

We implemented information discovery application within Diaspora network as a web application using methods given in its RESTful API. This facilitates real-name-based searching within Diaspora network. Users who would like to find friends without bothering their Diaspora handles can register themselves by accessing that web application.

Users in DSNs demands different features. It may sometimes create conflicts among themselves. As example some people more concerns about their privacy and they do not like to be searchable within their social network. But some people are willing to be searchable. Therefore, using this API it is possible to provide such features in the form of an app, so that users who are willing can use it.

Likewise our implemented API gives the opportunity to create third-party apps for Diaspora distributed social network. It helps to fulfill users' demands for different feature in the form of third-party apps. Many developers will be more concern in developing apps using such a RESTful API because of its usability and understandability. Implemented API will be a great factor for raise the number of Diaspora users and Diaspora pods. With the development of the third-party apps for the Diaspora social network, its users will be satisfied with their demanded features.

At the beginning of our project we planned to implement real name searching facility as a inbuilt feature with the Diaspora software. After discussing with the Diaspora developer community they suggested to implement it as a separate third-party application due to the conflicts of its users about their privacy. As I mentioned earlier some people are very strict about their privacy and don't want to be searchable within the Diaspora network. Hence we have implemented that app with the use of RESTful API. Implementation of the API with the authentication and authorization model is worked according to our planned procedure.

With the development of this project we have experienced lot of knowledge in different areas. At the beginning we had to do a research about existing distributed social networks. We experienced specific features in different DSNs like Diaspora [1], Friendica [2], BuddyCloud [3], and Freenet [4]. After selecting Diaspora social network which we planned to make feature rich, we have to examine its code base as well as its documentations to familiar with it. Hence we gained a good understanding about its distributed nature. We have collaborated with the Diaspora developer community in designing API and its features. They suggest some modifications to our suggestions as well as recommend to use some technologies like JWT (JSON Web Tokens) in encode and decode purposes.

Finally, we successfully completed all the project objectives and implemented RESTful API for the use of third-party app developers in Diaspora distributed social network. Also we implemented information discovery app which demonstrates the use of API and covers a main feature "Real name searching" which is not available in Diaspora. Code bases related to the API [30] and information discovery app [36] are publicly available at Github. Furthermore, our project accepted for a poster and vocal presentation at mini-ERU symposium of university of Moratuwa. We participated for NBSQA and SLIC software competitions.

Future Work

We are currently getting feedback about our work from the Diaspora community [37]. We have modified the code according to those so that they adhere to the community guidelines and conventions. Some functions of the API are still under discussion on the community and we will work with them on those.

A native mobile app for Diaspora for Android and iOS is a feature highly demanded among Diaspora users. With the use of the API implementing such an app respecting user privacy is possible. We are hoping to implement an Android app to fulfill this demand.

Providing developers SDKs will speed up the development and increase the ease of development. We are hoping to start with an SDK for Java developers. We will use the experience we gathered in the development of the Search App. We are hoping to follow up with an SDK for android as well.

References

[1] "Diaspora," [Online]. Available: http://wiki.diaspora-project.org/. [Accessed 25 March 2013].

[2] "Friendica," [Online]. Available: http://friendica.com/. [Accessed 15 March 2013].

[3] "BuddyCloud," [Online]. Available: http://buddycloud.com/. [Accessed 13 March 2013].

[4] "Freenet," [Online]. Available: https://freenetproject.org/whatis.html. [Accessed 04 April 2013].

[5] "OAuth," OAuth community, [Online]. Available: http://oauth.net/. [Accessed 29 April 2013].

[6] "Distributed social networks," Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Distributed_social_network. [Accessed 15 March 2013].

[7] "BuddyCloud wiki," [Online]. Available: https://buddycloud.org/wiki/Main_Page. [Accessed 13 March 2013].

[8] "Freenet documentation," [Online]. Available: https://freenetproject.org/documentation.html. [Accessed 21 April 2013].

[9] "Friendica - Github," [Online]. Available: https://github.com/friendica/friendica. [Accessed 15 March 2013].

[10] "Friendica - Developer mail group," [Online]. Available: friendica@librelist.com. [Accessed 15 March 2013].

[11] "Friendica - Issue tracker," [Online]. Available: http://bugs.friendica.com/my_view_page.php. [Accessed 15 March 2013].

[12] "Diaspora - Software architecture," [Online]. Available: https://docs.google.com/presentation/d/1Okk-QOELHk4FXAL9t0msgfa8Hqclu7RgRg6rkHjaZME/edit#slide=id.g7efc55b2_2_18. [Accessed 24 April 2013].

[13] "Diaspora - Security architecture," [Online]. Available: https://github.com/diaspora/diaspora/wiki/Security-Architecture-Proposal. [Accessed 08 May 2013].

[14] "Facebook API," [Online]. Available:

http://developers.facebook.com/docs/reference/apis/. [Accessed 1 May 2013].

[15] "Twitter developers," [Online]. Available: Available: https://dev.twitter.com/. [Accessed 1 May 2013].

[16] "Google+ API," [Online]. Available: https://developers.google.com/+/api/. [Accessed 1 May 2013].

[17] "Google App Engine - Developer guide," [Online]. Available: https://developers.google.com/appengine/docs/. [Accessed 09 May 2013].

[18] "Building a Platform API on Rails," [Online]. Available: http://blog.thecodepath.com/2011/06/27/building-a-platform-api-on-rails/. [Accessed 1 May 2013].

[19] S. Ventures, "How to turn your rails site into an OAuth Provider," [Online]. Available: http://stakeventures.com/articles/2007/11/26/how-to-turn-your-rails-site-into-an-oauth-provider. [Accessed 28 April 2013].

[20] "BitTorrent," Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/BitTorrent. [Accessed 17 April 2013].

[21] "Gnutella," Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Gnutella. [Accessed 17 April 2013].

[22] "Kazaa," Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Kazaa. [Accessed 17 April 2013].

[23] S. Mei, "How Diaspora Connects Users," [Online]. Available: http://www.sarahmei.com/blog/2011/09/17/how-diaspora-connects-users/. [Accessed 15 March 2013].

[24] G. Fox, "Peer-to-peer networks," *Comput. Sci. Eng,* vol. III, pp. 75-77, 2001.

[25] C. Wang, L. Xiao, Y. Liu and P. Zheng, "Distributed caching and adaptive search in multilayer P2P networks," in *24th International Conference on Distributed Computing Systems*, 2004.

[26] A. Iamnitchi, I. Foster and D. Nurmi, “A Peer-to-Peer Approach to Resource Discovery in Grid Environments,” in *IEEE High Performance Distributed Computing*, 2002.

[27] H. Unger, T. Böhme, M. Wulff, G. Babin, P. Kropf, F. Informatik, U. Rostock, T. Ilmenau, É. Hautes and É. Commerciales, "An Optimized Search Mechanism for Large Distributed Systems," University of Rostock, Germany, 1999.

[28] Y. Upadrashta, J. Vassileva and W. Grassmann, "Social Networks in Peer-to-Peer Systems," in *38th Annual Hawaii International Conf. on System Sciences*, Washington, 2005.

[29] A. Hailes and Abdul-Rahman, "Using Recommendations for Managing Trust in Distributed Systems," in *IEEE Malaysia International Conf. on Communication*, Kuala Lumpur, 1997.

[30] Immanent, "Diaspora API - Github repository," 08 June 2013. [Online]. Available: https://github.com/Immanent/diaspora.

[31] Immanent, "Project web site," 21 July 2013. [Online]. Available: http://immanent.github.io/.

[32] "Factory Girl - Rails testing," [Online]. Available: http://www.hiringthing.com/2012/08/17/rails-testing-factory-girl.html#sthash.YDBNF8m0.dpbs. [Accessed 12 Augest 2013].

[33] "Bootstrap," Bootstrap, [Online]. Available: http://getbootstrap.com/. [Accessed 19 July 2013].

[34] "JUnit - Java unit testing," [Online]. Available: http://junit.org/. [Accessed 21 Augest 2013].

[35] "DBUnit - Java database testing," [Online]. Available: http://dbunit.sourceforge.net/. [Accessed 21 Augest 2013].

[36] Immanent, "Search App - Github repository," 19 July 2013. [Online]. Available: https://github.com/Immanent/SearchApp.

[37] Immanent, "Diaspora API - Pull request," 19 10 2013. [Online]. Available: https://github.com/diaspora/diaspora/pull/4554.

Appendix I

Software Requirement Specification

Overall Description

Product perspective

The API for Diaspora should provide simple interfaces to retrieve protected data and authentication for the apps. The search app should be capable of listing friends according to the searching texts. It should get relevant friends from different pods using an efficient searching mechanism.

Product functions

According to the user input the search app should responds with relevant instruction messages. App should provide help menu for the convenience of the user.

User characteristics

API developers will be provided required documentations in order to understand the API functionalities. So the app developers should have an experience in using an API and such documentations.

Specific Requirements

API for third-party app developers

The API should be able to authenticate and authorize third-party apps

When provided with the Diaspora handle with the user the API should redirect user to the login page of Diaspora. After login is successful the user is asked to allow the app to access her personal data. User should not be entering her password to the app itself. If the user has a valid session in Diaspora already the step with the login page is omitted.

Functionalities of the API should be available to both standalone applications and web applications

The API should be able to handle requests from both web apps and standalone (Desktop, mobile, etc.) apps.

Users must be able to allow or disallow apps

Before authorizing an app, the user must be informed. The app is authorized only if the user explicitly allows it.

Users need to be displayed the set of actions the app is allowed to do

The access scopes (which defines the actions the app can perform on behalf of the user, and information the app will be provided) the app requested must be displayed to the users. The API should not allow the app any actions or data that are not displayed to and allowed by the user.

Users should be able to pick what scopes they desire and allow only them

There will be mandatory access scopes that the app needs to its operation. The user will be able to see those but won't be able to disallow. Other optional scopes can be allowed or disallowed at users will. For example, the sample search application must have the access to user’s contacts list, but can optionally be allowed to post on behalf of the user.

API must be allowed to provide authorized apps functionalities to perform following actions on behalf of the user

- Post status messages with photos, mentions and location information
- Comment on posts and like posts
- Send friend requests
- Send messages

User must be able to limit above actions of the app to a certain set of aspects or made them public

Developer must be able to upload the manifest file of her app to his diaspora pod and download a signed document which will be provided to any pod the app will access

The manifest file contains data about the app. This file needs to be verified by the pod to which the app requests controlled access. So that there will be a developer profile associated with every app. This component should be implemented with the API although it is not a part of it.

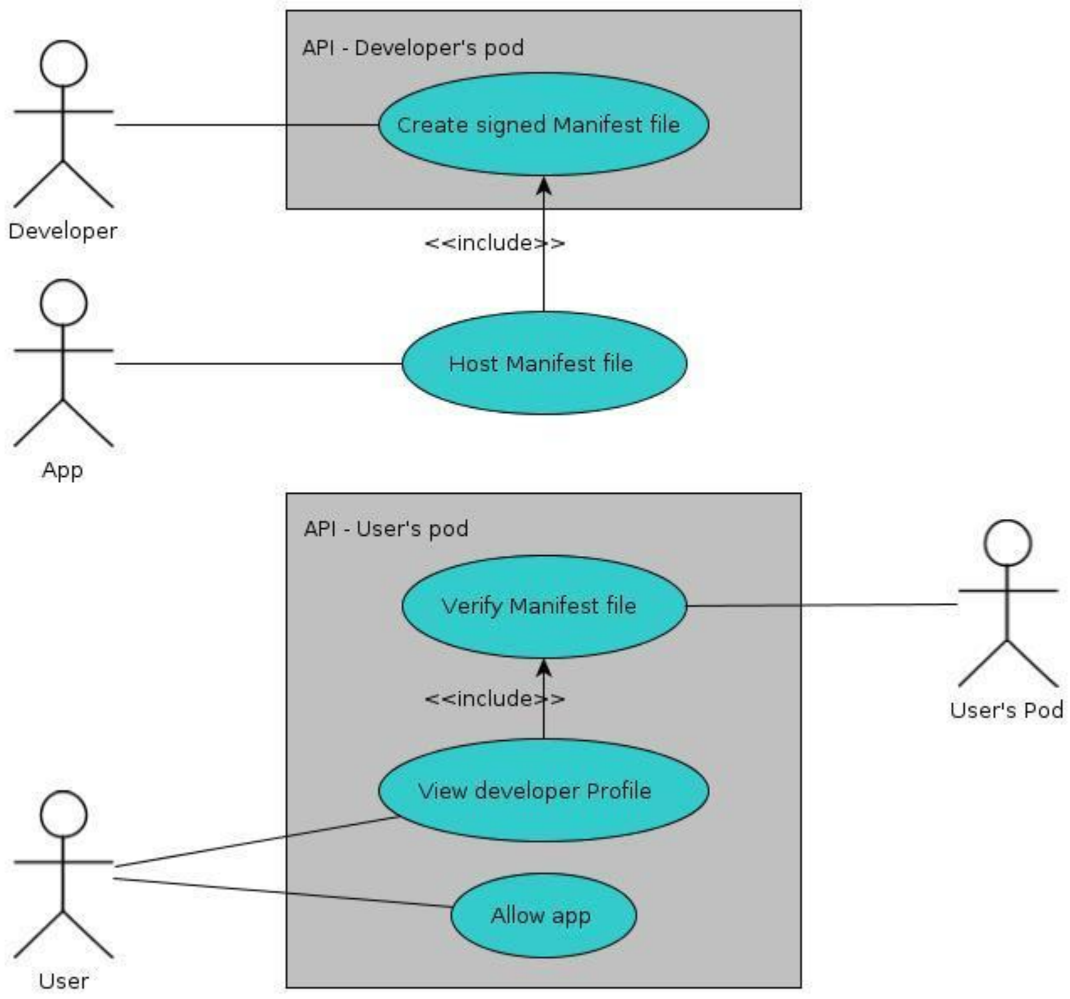


Figure I1: User trusting app based on the Developer profile

Users should be able to review allowed apps

The user should be able to review the granted scopes and the associated developer profile via interface. From the developer profile user should be able to view other apps by that developer and other users' rates of the developer. The purpose is giving the user an idea about the trustworthiness of the app. User must be able to revoke a given token to an app making it disallowed.

User Discovery App

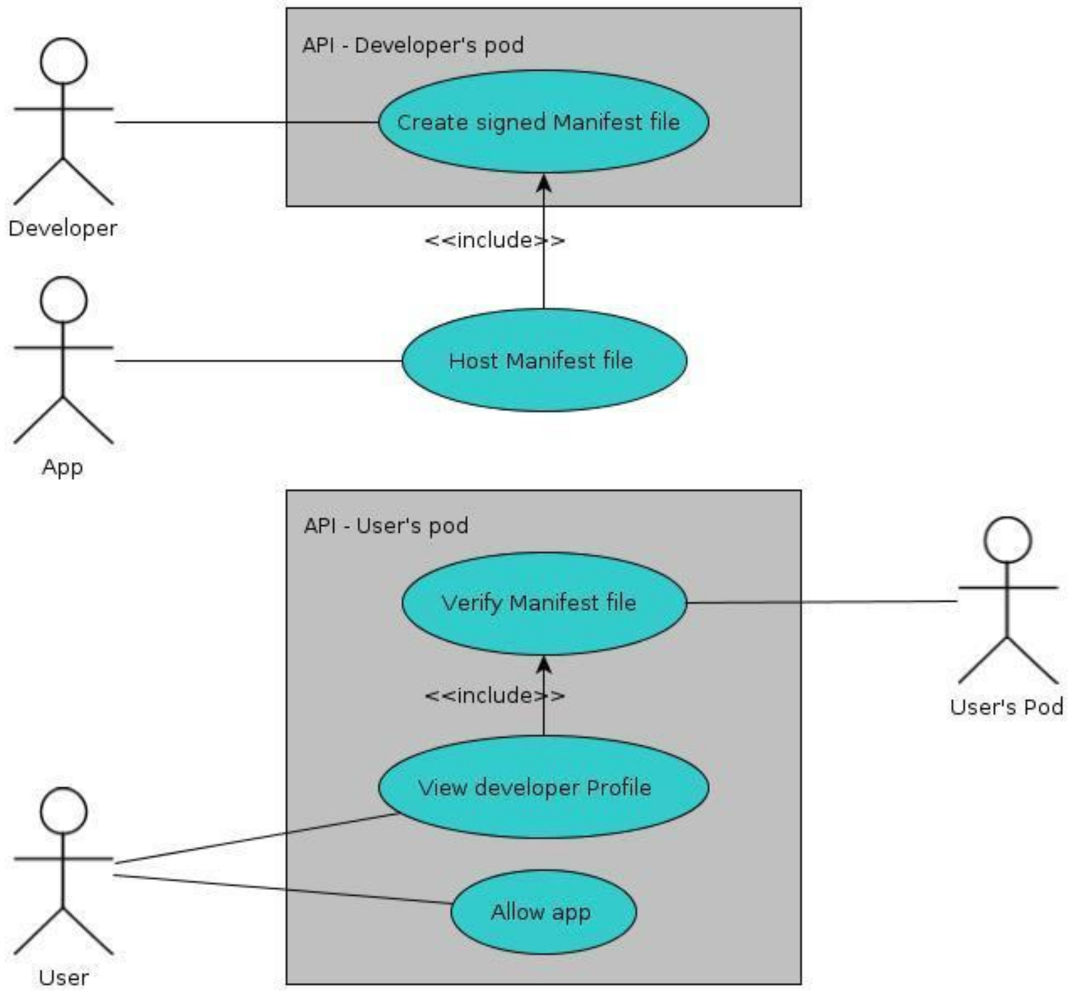


Figure I2: Use Case - User discovery app

User discovery application will be a web application

Search application will be a web application. App get contact information by calling Diaspora API through HTTP request and receives contacts in JSON response. It contains a central database in the app server to store to the users information, pods information.

Users should able to register and restrict the exposure of their private data

When user authenticating the search app it should get the minimum user information (e.g., name) and gives users the option to restrict his private information (friend list, location, etc.) if he/she wants. It should not expose users private information without his permission.

Users should be able to search friends by their real names

Currently Diaspora finds friends using their Diaspora ID. So users need to remember their friend's pod ID to search their friends. Instead of searching friends by their Diaspora ID's, users should be able to find friends using real names without remembering relevant pod ID's.

Users should be able to get a list of relevant friends

The search app should be able to list friends according to the searched name. It should list friends who contains full or partial of searched name in Diaspora network.

User should able limit his search level

Application should give users the option of limiting the search depth (friend of friend...). Default it will list up to 5 levels. User can limit the search depth and get more relevant results.

User should able to revoke the search app

User should be able to revoke the search app whenever he wants. He has the option of removing the exposed information to application. If user did not remove his private data it will help other application users get more accurate results.

Usability

Provide API manual for developers

A user who is familiar with Diaspora must be able to learn how to use the API to develop a simple app (which is authenticated by the user and capable of retrieving user's information and posting on behalf of his/her) by reading the manual.

Familiar UI for Search app

Search application UI will adhere the diaspora UI style hence user get a friendly environment in search application.

Search app should responds within few seconds

User may be able to get relevant friend list in maximum time period of 10 seconds because the search app should be efficient than the manual searching mechanism. Result friend list will be updated every time search depth increases hence user does not want to wait until whole search process complete to view the search result.

Reliability

Availability

The API should be available whenever the pod is up. But it is possible that when a major version is released and the pod is updating the API to be unavailable for at most 1 hour. Search application server is up 24/7.

Accuracy

Output of the search app should be 100% accurate. The search app should answer user inputs accurately. API should expose relevant data according to the users preferences.

Mean Time to Repair

After a major release, the pod’s mandatory updating session takes at most one hour. Other failures should be repaired within a mean time of one hour.

Performance

Search app performance

Diaspora search app is highly user interactive. So its response time should be in optimal range So the app should respond less than three second for each and every request. Its performance varies depending on the network traffic.

Access the database

API is responsible for exposing relevant data from the users. So to make it more efficient to third-party apps to get those exposed data, at the back-end the API should be accessed data within a few milliseconds.

Access to the other pods

Search app responsible for listing requested friend list from existing pods. So when searching friends from many Diaspora pods the app should get data within few seconds.

Supportability

User interface supportability

The search app shall allow users to easily find out where the Graphical interfaces (buttons, text fields and labels) with suitable contrast background and suitable font size. Follow the current diaspora UI design style.

Supportability for app developers

API will be designed to maximize and enhance the third-party apps developed for the Diaspora by providing simple interfaces to retrieve protected data and authentication for the apps. API manual documentation is provided describing each functionality with examples.

Information discovery supportability

Today Diaspora users find their friends using their friends Diaspora ID. So users have to remember those IDs to search their friends. But users can easily find their friends using their real names using search app.

Follow Diaspora developer standards

When implementing the API and search app we will follow the current diaspora coding standards, naming conventions etc, which will assist Diaspora community to easily understand our implementation.

Design Constraints

Ruby programming language and other standards of Diaspora

As Diaspora is written in Ruby the same language will be used in the API implementation as well. The purpose of this constraint is to make the existing Diaspora community to be able to contribute to the API codebase easily.

Diaspora developers follow a specific workflow with their version controlling. The Git workflow documents this on their community run wiki. Following this workflow is strict for the development of the API because unless it will be difficult to merge it to the existing system.

Test Driven development or Behavior driven development is encouraged in the Diaspora developer community. It is important to follow this wherever possible. Specific technologies used in tests of different components and important of test coverage is described in the Test Workflow document.

Different opinion in Diaspora community

Diaspora is an community project hence we cannot make our own design decision in this project. Community has different people with different opinions hence their feedback may conflict regarding our designs.

Privacy

The API will expose users private data to the other parties (Apps). So it should happen according to the users preferences to secure their privacy.Some users who are using Diaspora don’t like to show up their names in search results of other users. So there should be an option for them to hide from such incidents.

Online User Documentation and Help System Requirements

A thorough documentation explaining the system is required as this software will be maintained by a community of developers. By reading the documentation new members of the community should be able to understand and be able to contribute as soon as possible.

A manual explaining each functionality of the API providing examples is required.

Interfaces

User Interfaces

Search application provides a user interfaces for the app users to discover friends in diaspora network. When user access the app first time, there is an authentication UI to authenticate app and restrict the exposure of his private data. Main UI takes the input from the user, display output result and error and instruction messages.

Software Interfaces

Database access for Person, User, Contact, Post components are done by the Rails Models. This class layer which is used by the current Diaspora system will be reused by the API as an interface to the pod Database.

Communications Interfaces

The API provides an interface to the third-party app developers to use functionalities in Diaspora. App will send http requests and take JSON responses from the API.

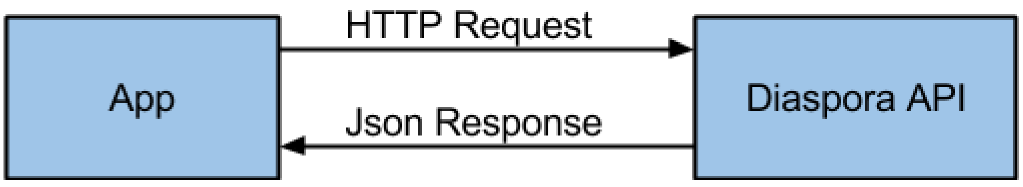


Figure I3: Communication between App and API

Appendix II

Specification of Application Programming Interface of Diaspora

Get Auth Token

Authenticate the third-party application and obtain the auth_token

Request

Method	**URL**
POST	http://<pod_name>/authorize/verify

Type	**Params**	**Values**
POST	diaspora_id	string
POST	signed_manifest	string

- **diaspora_id -** diaspora_id must be the user's diaspora id.
- **signed_manifest -** signed_manifest must be content of the manifest file signed with developer's private key.

Response

Status	Response
Ok	{ "auth_token": <auth_token> } auth_Token(string)

	User will be redirect to the authentication page.
000	{"error":"Signed manifest content missing."}
001	{"error":"Manifest decoding fail."}
002	{"error":"Manifest verification fail."}
003	{"error":"Invalid API key."}

Table 1 - Get Auth Token

Get Refresh Token

Get the refresh token using auth_token.

Request

Method	**URL**
POST	http://<pod_name>/dauth/authorize/authorization_token?auth_token=<auth_token>

Type	**Params**	**Values**
HEAD	auth_token	string

auth_token

The auth_token that was given in response to http://<pod name>/authorize/verify

Response

Status	**Response**
Ok	Refresh token sent to the callback URL and redirect to the relevant URL mentioned in the manifest.

101	{"error":"Invalidauth_token"}
102	{"error":"Refresh token generation failed"}

Table 2 - Get Refresh Token

Get Access Token

Get the Access token using the Refresh token in order to use API functionalities.

Request

Method	URL
POST	http://<pod_name>/dauth/authorize/access_token?refresh_token=<refresh_token>

Type	Params	Values
HEAD	refresh_token	string

get_user_contact_list

Get the user's contact list using user's diaspora_handle and access_token.

Request

Method	URL
GET	http://<pod_name>/api/users/get_user_contact_list/<diaspora_handle>/<access_token>

Type	Params	Values
HEAD	diaspora_handle	string
HEAD	access_token	string

diaspora_handle

diaspora_handle must be the user's diaspora handle.

access_token

access_token must be the token given for an user for a particular application.

Response

Status	**Response**

Ok	{
	"user_contact_list": [
	{
	"first_name": <first_name>,
	"last_name": <last_name>,
	"diaspora_handle": <diaspora_handle>,
	"location": <location>,
	"birthday": <birthday>,
	"gender": <gender>,
	"bio": <bio>,
	"url": <contact_url>,
	"avatar": <image_url>
	},
	]
	}

bad_request	{
	"error": "400"
	}

Table 3 - Get Access Token

get_user_aspects_list

Get the user's aspect list using user's diaspora_handle and access_token.

Request

Method	URL
GET	http://<pod_name>/api/users/get_user_aspects_list/<diaspora_handle>/<access_token>

Type	Params	Values
HEAD	diaspora_handle	string
HEAD	access_token	string

diaspora_handle

diaspora_handle must be the user's diaspora handle.

access_token

access_token must be the token given for an user for a particular application.

Response

Status	Response
Ok	{ "users_aspects_list": [{ "aspect_name": <aspect_name>, "id": <aspect_id>, "user_id": <user_id> },]

	}

bad_request	{
	"error": "400"
	}

Table 4 - Get user aspect list

get_user_followed_tags_list

Get the user's followed tag list using user's diaspora_handle and access_token.

Request

Method	URL
GET	http://<pod_name>/api/users/get_user_followed_tags_list/<diaspora_handle>/<access_token>

Type	Params	Values
HEAD	diaspora_handle	string
HEAD	access_token	string

diaspora_handle

diaspora_handle must be the user's diaspora handle.

access_token

access_token must be the token given for an user for a particular application.

Response

Status	Response
	{ "users_followed_tag_list": [

Ok	{
	"id": <id>,
	"name": <name>
	},
	]
	}

bad_request	{
	"error": "400"
	}

Table 5 - Get followed tag list

get_user_details

Get the user details using user's diaspora_handle and access_token.

Request

Method	**URL**
GET	http://<pod_name>/api/users/get_user_details/<diaspora_handle>/<access_token>

Type	**Params**	**Values**
HEAD	diaspora_handle	string
HEAD	access_token	string

diaspora_handle

diaspora_handle must be the user's diaspora handle.

access_token

access_token must be the token given for an user for a particular application.

Response

Status	Response

Ok	{
	"user_details": {
	"first_name": <first_name>,
	"last_name": <last_name>,
	"diaspora_handle": <diaspora_handle>,
	"location": <location>,
	"birthday": <birthday>,
	"gender": <gender>,
	"bio": <bio>,
	"url": <contact_url>,
	"avatar": <image_url>
	}
	}

bad_request	{
	"error": "400"
	}

Table 6 - Get user details

get_user_contact_handle_list

Get the user contact handle list using user's diaspora_handle and access_token.

Request

Method	URL
GET	http://<pod_name>/api/users/get_user_contact_handle_list/<diaspora_handle>/<access_token>

Type	**Params**	**Values**
HEAD	diaspora_handle	string
HEAD	access_token	string

diaspora_handle

diaspora_handle must be the user's diaspora handle.

access_token

access_token must be the token given for an user for a particular application.

Response

Status	**Response**

Ok	{
	"user_contact_handle_list": [
	{
	"handle": <diaspora_handle>
	},
	]
	}

bad_request	{
	"error": "400"
	}

Table 7 - Get user contact handle list

get_app_scopes_of_given_user

Get the user scopes for a given application using user's diaspora_handle, app_id and access_token.

Request

Method	URL
GET	http://<pod_name>/api/users/get_app_scopes_of_given_user/app_id/<diaspora_handle>/<access_token>

Type	Params	Values
HEAD	diaspora_handle	string
HEAD	access_token	string
HEAD	app_id	string

diaspora_handle

diaspora_handle must be the user's diaspora handle.

access_token

access_token must be the token given for an user for a particular application.

app_id

app_id must be the ID which user has registered.

Response

Status	Response

Ok	{
	"user_app_scopes":{
	<profile_read>,

	<profile_write>, } }

bad_request	{
	"error": "400"
	}

Table 8 - Get user app scopes

edit_email

Edit user email address using user's diaspora_handle, valid email and access_token.

Request

Method	**URL**
POST	http://<pod_name>/api/users/edit_email/<access_token>

Type	**Params**	**Values**
post	diaspora_handle	string
post	email	string
HEAD	access_token	string

diaspora_handle

diaspora_handle must be the user's diaspora handle.

access_token

access_token must be the token given for an user for a particular application.

email

email must be a valid email.

Response

Status	Response
Ok	-
bad_request	{ "error": "400" }
unsupported_type	{ "error": "402" }

Table 9 – Edit user email

edit_last_name

Edit user last name using user's diaspora_handle, name and access_token.

Request

Method	URL
POST	http://<pod_name>/api/users/edit_last_name/<access_token>

Type	Params	Values
post	diaspora_handle	string
post	last_name	string
HEAD	access_token	string

diaspora_handle

diaspora_handle must be the user's diaspora handle.

access_token

access_token must be the token given for an user for a particular application.

last_name

last_name is substituted for user's last name.

Response

Status	Response
ok	-
bad_request	{ "error": "400" }
unsupported_type	{ "error": "402" }

Table 10 – Edit last name

edit_user_location

Edit user location using user's diaspora_handle, location and access_token.

Request

Method	URL
POST	http://<pod_name>/api/users/edit_user_location/<access_token>

Type	Params	Values
post	diaspora_handle	string
post	location	string
HEAD	access_token	string

diaspora_handle

diaspora_handle must be the user's diaspora handle.

access_token

access_token must be the token given for an user for a particular application.

location

location is substituted for user's location.

Response

Status	Response
ok	-
bad_request	{ "error": "400" }
unsupported_type	{ "error": "402" }

Table 11– Edit user location

get_given_user_status_list

Get the user's status list using user's diaspora_handle and access_token.

Request

Method	URL
GET	http://<pod_name>/api/statusMessages/get_given_user_status_list/<diaspora_handle>/<access_token>

Type	Params	Values
HEAD	diaspora_handle	string
HEAD	access_token	string

diaspora_handle

diaspora_handle must be the user's diaspora handle.

access_token

access_token must be the token given for an user for a particular application.

Response

Status	Response
Ok	{ "users_status_messages_list": [{ "author_id": <author_id>, "comments_count": <comments_count>, "diaspora_handle_of_creator": <diaspora_handle>, "status_id": <status_id>, "likes_count": <likes_count>, "text": <text> },] }
bad_request	{ "error": "400" }

Table 12– Get status list

get_comments_for_status_message

Get comment list for a given status message using user's diaspora_handle, status_id and access_token.

Request

Method	URL
GET	http://<pod_name>/api/statusMessages/get_comments_for_status_message/<id>/<diaspora_handle>/<access_token>

Type	Params	Values
HEAD	diaspora_handle	string
HEAD	access_token	string
HEAD	id	string

diaspora_handle

diaspora_handle must be the user's diaspora handle.

access_token

access_token must be the token given for an user for a particular application.

id

id must be a valid ID of a status message.

Response

Status	Response
Ok	{ "comment_list": [{

	"author_id": <author_id>, "commentable_id": <commentable_id>, "id": <id>, "likes_count": <likes_count>, "text": <text> },] }

bad_request	{
	"error": "400"
	}

unauthorized	{
	"error": "401"
	}

Table 13– Get comment list

get_likes_for_status_message

Get like count for a given status message using user's diaspora_handle, status_id and access_token.

Request

Method	URL
GET	http://<pod_name>/api/statusMessages/get_likes_for_status_message/<id>/<diaspora_handle>/<access_token>

Type	Params	Values
HEAD	diaspora_handle	string
HEAD	access_token	string
HEAD	id	string

diaspora_handle

diaspora_handle must be the user's diaspora handle.

access_token

access_token must be the token given for an user for a particular application.

id

id must be a valid ID of a status message.

Response

Status	**Response**
Ok	{ "likes_count": { "likes_count": <likes_count> } }
bad_request	{ "error": "400" }
unauthorized	{ "error": "401" }

Table 14–Get likesfor a statusmessage

get_number_of_comments_for_status_message

Get comment count for a given status message using user's diaspora_handle, status_id and access_token.

Request

Method	URL
GET	http://<pod_name>/api/statusMessages/get_number_of_comments_for_status_message/<id>/<diaspora_handle>/<access_token>

Type	Params	Values
HEAD	diaspora_handle	string
HEAD	access_token	string
HEAD	id	string

diaspora_handle

diaspora_handle must be the user's diaspora handle.

access_token

access_token must be the token given for an user for a particular application.

id

id must be a valid ID of a status message.

Response

Status	Response

Ok	{
	"comments_count": {
	"comments_count": <comments_count>

	} }

bad_request	{
	"error": "400"
	}

unauthorized	{
	"error": "401"
	}

Table 15 - Get numberofcommentsfor a statusmessage

create_status_message

Create a status message using user's diaspora_handle, text_message and access_token.

Request

Method	URL
POST	http://<pod_name>/api/statusMessages/create_status_message/<access_token >

Type	Params	Values
POST	diaspora_handle	string
HEAD	access_token	string
POST	text	string

diaspora_handle

diaspora_handle must be the user's diaspora handle.

access_token

access_token must be the token given for an user for a particular application.

text

text should contains message of the status message.

Response

Status	Response
ok	
bad_request	{ "error": "400" }

Table 16–Create statusmessage

delete_status_message

Delete a given status message using user's diaspora_handle, status_id and access_token.

Request

Method	URL
DELETE	http://<pod_name>/api/statusMessages/delete_status_message/<access_token>

Type	Params	Values
POST	diaspora_handle	string
HEAD	access_token	string
POST	id	string

diaspora_handle

diaspora_handle must be the user's diaspora handle.

access_token

access_token must be the token given for an user for a particular application.

id

id must be a valid ID of a status message.

Response

Status	**Response**
Ok	
bad_request	{ "error": "400" }
unauthorized	{ "error": "401" }

Table 17 - Delete status message

get_given_user_comment_list

Get given user's comments list using user's diaspora_handle and access_token.

Request

Method	**URL**
GET	http://<pod_name>/api/comments/get_given_user_comment_list/<diaspora_handle>/<access_token>

Type	**Params**	**Values**
HEAD	diaspora_handle	string
HEAD	access_token	string

diaspora_handle

diaspora_handle must be the user's diaspora handle.

access_token

access_token must be the token given for an user for a particular application.

Response

Status	**Response**

Ok	{
	"user_comment_list": [
	{
	"author_id": <author_id>,
	"commentable_id": <commentable_id>,
	"id": <id>,
	"likes_count": <likes_count>,
	"text": <text>
	},
	]
	}

bad_request	{
	"error": "400"
	}

Table 18 – Get given user comment list

get_likes_count

Get given comment's likes count using user's diaspora_handle, comment_id and access_token.

Request

Method	**URL**

GET	http://<pod_name>/api/comments/get_likes_count/<id>/<diaspora_handle>/<access_token>

Type	**Params**	**Values**
HEAD	diaspora_handle	string
HEAD	access_token	string
HEAD	id	string

diaspora_handle

diaspora_handle must be the user's diaspora handle.

access_token

access_token must be the token given for an user for a particular application.

id

id must be a valid ID of a comment.

Response

Status	**Response**

Ok	{
	 "likes_count": {
	 "likes_count": <likes_count>
	 }
	 }

bad_request	{
	 "error": "400"
	 }

bad_request	{ "error": "401" }

Table 19 – Get likes count

create_comment

Create a comment using user's diaspora_handle, text_message and access_token.

Request

Method	URL
POST	http://<pod_name>/api/comments/create_comment/<access_token>

Type	Params	Values
POST	diaspora_handle	string
POST	text	string
HEAD	access_token	string

diaspora_handle

diaspora_handle must be the user's diaspora handle.

access_token

access_token must be the token given for an user for a particular application.

text

text should contains message of the comment.

Response

Status	Response

Ok	-
bad_request	{ "error": "401" }

Table 20 – Create comment

delete_comment

Delete a given comment using user's diaspora_handle, comment_id and access_token.

Request

Method	URL
DELETE	http://<pod_name>/api/comments/delete_comment/<access_token>

Type	Params	Values
POST	diaspora_handle	string
HEAD	access_token	string
POST	id	string

diaspora_handle

diaspora_handle must be the user's diaspora handle.

access_token

access_token must be the token given for an user for a particular application.

id

id must be a valid ID of a comment.

Response

HTTP Status	**Response**
Ok	
bad_request	{ "error": "400" }
bad_request	{ "error": "401" }

Table 21 – Delete comment

Error Codes

Following table contains API error codes.

Error Code	Description
000	Signed manifest content missing
001	Manifest decoding fail
002	Manifest verification fail
100	Illegal authentication token
101	Error generating refresh token
200	Illegal Refresh Token
300	Illegal Access Token
301	Access token is invalid
310	No profile read permissions
311	No profile write permissions
312	No profile delete permissions

320	No comments read permissions
321	No comments write permissions
322	No comments delete permissions
330	No post read permissions
321	No post write permissions
322	No post delete permissions
400	Bad request
401	Accessing unauthorized contents
402	Accessing with an unsupported param values
403	Accessing with an unauthorized access token

Table 22- API error codes

Abbreviations

API	-	Application Programming Interface
BFS	-	Breadth First Search
CRUD	-	Create, Read, Update, Delete
CSNs	-	Centralized Social Networks
DSNs	-	Distributed Social Networks
FQL	-	Facebook Query Language
GAE	-	Google App Engine
GUI	-	Graphical User Interface
HAML	-	HTML Abstraction Markup Language
HTML	-	HyperText Markup Language
HTTP	-	HyperText Transfer Protocol
JSON	-	JavaScript Object Notation
JSP	-	Java Server Pages
JWT	-	JSON Web Token
MVC	-	Model View Controller
P2P	-	Peer to Peer
REST	-	Representational State Transfer
RSS	-	Rich Site Summary
SRS	-	Software Requirement Specification
TTD	-	Test Driven Development
TTL	-	Time to Live
UML	-	Unified Modeling Language