SOFTWARE DESIGN DOCUMENT

for

μ2 - A Social Network for Musicians and Bands

Prepared by
Suryansh D Kumar 13IT148
Chinmay Deshpande 13IT210
PBT Manikanta 13IT226

Department of Information Technology
NITK Surathkal

Contents

1 Introduction

The purpose of this software design document is to provide a low-level description of the $\mu 2$ system, providing insight into the structure and design of each component. Topics covered include the following:

- Class Hierarchies and interaction diagrams
- Data-flow and design
- Processing narratives
- Algorithmic models
- Design constraints and restrictions
- User Interface Design
- Test cases and expected results

1.1 Goals and objectives

The purpose of mu-2 is to become a complete social platform for all musicians and bands alike. Although music is an integral part of the Indian community, we still rely on word-of-mouth publicity for musicians who are looking to find a job. We plan to develop an integrated platform where everyone can post job requirements, band vacancies, etc. and help the community find each other.

Accordingly, the final product must be quick, efficient and extremely easy to use with a minimal interface. It must offer all the useful features without cluttering the application with unnecessary buttons and bars.

1.2 Statement of Scope

This system is made of two primary components: a client side application that the users will be using on their mobile phones and a server-side application(which acts like an API endpoint) which updates and synchronizes data across devices. As seen below, we can break down the features into core, additional and dream. Core features are to be developed before writing the test cases and hence comprise of the core of the application. Additional feature development is to be done after completion of core testing and before the deadline. Dream features are to be developed on completion of integration of additional and core features. The organization is as follows:

- Core
  - Users(whether they be musicians, bands or others) need to have a profile page. The entire idea of the application hinges on users checking other profiles and communicating to each other via the contact provided.
  - 'The Wall' is a vital concept since we need to have all the posts listed on a single page. Addition of posts should be made available to everyone. All rights must be given to the post content creator to mark his request as satisfied/unsatisfied on completion.
  - Filter search is also another important feature we plan to integrate since there will be posts from all over the area and hence there should be a mechanism to filter out only the important posts to a specific user.
  - Account management since it is not ideal for the user to input his information each time he install the application.
- Additional
  - Updation/Deletion of posts.
  - Live updates on addition of posts by someone else whose requirements matches the one specified by the user.
  - Account Management/Login can be done through OAuth.
- Dream
  - Fantastic User interface.
  - In-App chat functionality.
  - Extending the reach of the application to the entire world.

1.3 Software Context

We will be looking to place the software in the product-line context. We will be offering the application on Google Play free of cost. Since development and maintenance are virtually nonexistent, funding should not be an issue. Currently, there does not exist any business model for the same and hence we are first looking to experiment with the product. If we get a good response, we can then add advertisements and collaborate with other companies on 'The Wall'.

One of the major issues involved in the same is that there already exist similar applications with quite a lot of popularity. Our aim is to make ours even better and contribute to the music community. Future development plans will be based on the features (if any) that do not make it in the initial release of the application. If all of these features are included, there are several experimental features that will potentially be incorporated.

1.4 Major Constraints

Since the application needs to be light-weight we need to add very minimal HTML/CSS. Also we need to make the UI responsive so that it is intuitive to use the application. These are kind-of opposing requirements which we will have to satisfy. Also, since the development team has absolutely no experience with the Android platform, a significant amount of the time before the deadline has to be spent understanding the environment. This might lead to fewer features in the initial release.

Also, we are faced with a major chicken-and-egg problem here. It is important that we publicize this application enough so that we get a lot of registrations in the initial phase and have word-of-mouth publicity. If the users do not register in large numbers, we are afraid that it may lead to the failure of the idea.

2 Data Design

2.1 Internal Software Data Structure

μ2's internal architecture is divided into two parts: server-side and client side.

On the client side there is no data dependence and we will not be storing any data on the application cache. The entire data is accessed via the Model View Controller system. Data will be requested from the server at application initialization and refreshed based on the necessary user actions.

The data structure on the server will essentially mirror the structure of the local Android Client in terms of member fields of classes. Server is implemented as an API endpoint using RubyonRails. Permanent information of the user is stored on the server using the SQLite3 Database.

The server and the Android client exchange the data using the JSON format. JSON is a lightweight object description language that is similar to XML. JSON is being used due to its versatility and because of the fact that the implementation is available both for Ruby and Java.

2.2 Global Data Structure

The global data structure of this application is best characterized by the database. The database structure shows the data involved in the application in its purest sense. The local Android client of μ2 will never access this database directly; it will instead issue requests to the server.

2.3 Temporary Data Structure

Temporary data structures, as they relate to μ2, refer to the data objects that are created on the local Android client, and also to the JSON objects that interchanged between the server and the client. The data objects created on the local device will only exist for the duration of time that the application is running, and will subsequently be destroyed. The JSON objects will only exist for the duration of the transaction between the client and server. The server will destroy the objects after sending them, and the client will destroy the JSON objects once they have been parsed.

2.4 Database Description

Figure 2.1: Relation Diagram

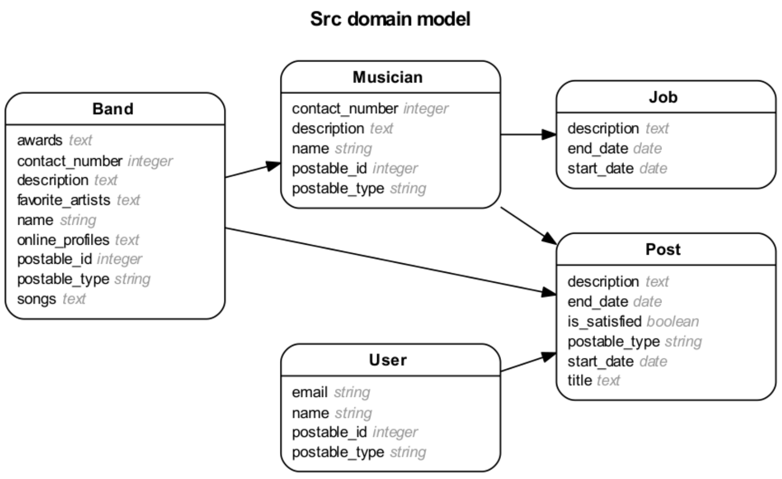
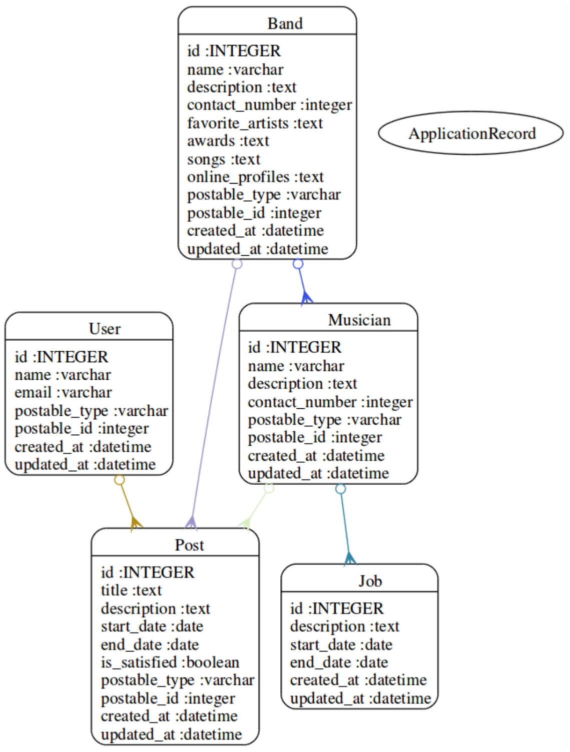


Figure 2.2: Database Schema



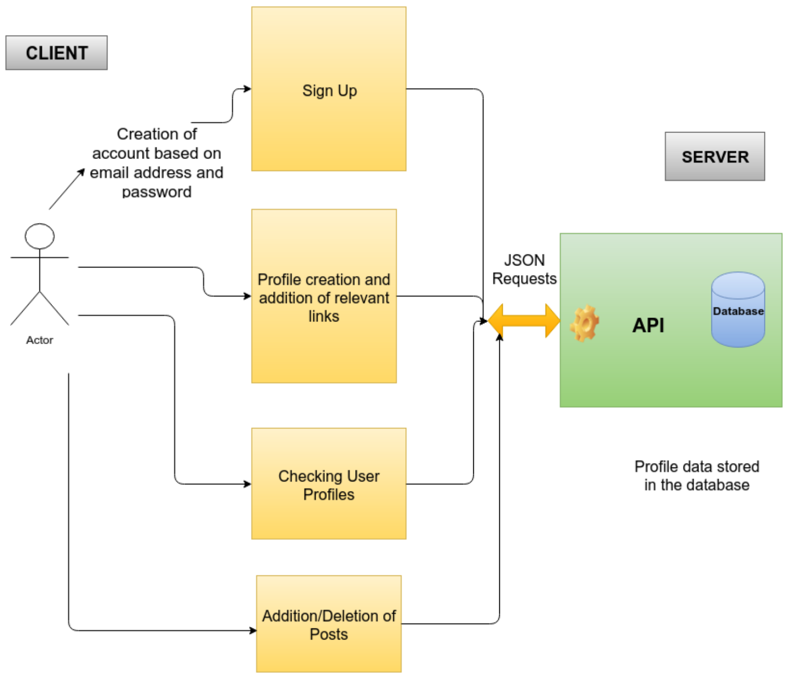
Consider the following schema:

```
create_table “bands”, force: :cascade do |t|
t.string “name”
t.text “description”
t.integer “contact_number”
t.text “favorite_artists”
t.text “awards”
t.text “songs”
t.text “online_profiles”
t.string “performable_type”
t.integer “performable_id”
t.string “locatable_type”
t.integer “locatable_id”
t.string “postable_type”
t.integer “postable_id”
t.datetime “created_at”, null: false
t.datetime “updated_at”, null: false
end

create_table “jobs”, force: :cascade do |t|
t.text “description”
t.date “start_date”
t.date “end_date”
t.datetime “created_at”, null: false
t.datetime “updated_at”, null: false
end

create_table “musicians”, force: :cascade do |t|
t.string “name”
t.text “description”
t.integer “contact_number”
t.string “performable_type”
t.integer “performable_id”
t.string “postable_type”
t.integer “postable_id”
t.string “locatable_type”
t.integer “locatable_id”
t.datetime “created_at”, null: false
t.datetime “updated_at”, null: false
end

create_table “posts”, force: :cascade do |t|
t.text “title”
t.text “description”
```

```
t.date “start_date”
t.date “end_date”
t.boolean “is_satisfied”
t.string “postable_type”
t.integer “postable_id”
t.datetime “created_at”, null: false
t.datetime “updated_at”, null: false
end

create_table “users”, force: :cascade do |t|
t.string “name”
t.string “email”
t.string “postable_type”
t.integer “postable_id”
t.datetime “created_at”, null: false
t.datetime “updated_at”, null: false
end
```

3 Architectural and component-level design

Figure 3.1: Use Case Diagram


CLIENT
Sign Up
Creation of account based on email address and password
Actor
Profile creation and addition of relevant links
Checking User Profiles
Addition/Deletion of Posts
JSON Requests
SERVER
API
Database
Profile data stored in the database

3.1 System Structure

A detailed description the system structure chosen for the application is presented. The $\mu 2$ system is broken up into two major components: a client side Android Application and a server side RubyonRails application which talks to the SQLite database.

The primary system architecture, as shown in the diagram, is a basic client server architecture which works on the request response mechanism.

The server component of $\mu 2$ is comprised of the Rails API interface, which manages incoming and outgoing messages and retrieves and stores data into the database. The interaction between the client and server takes place via JSON objects. Once a user performs an event which triggers a request to the server, a JSON object of the query is sent to the server. The server parses the JSON request and then uses its data to query the database. Once the back end database is queried, it returns its results as to the server. The server then sends the result of the query as a JSON object back to the client.

The client receives the server response as a JSON object. The application parses the JSON object, extract the results and displays it to the user of the application on the screen.

3.1.1 Architecture Diagram

Figure 3.2: Architectural Diagram

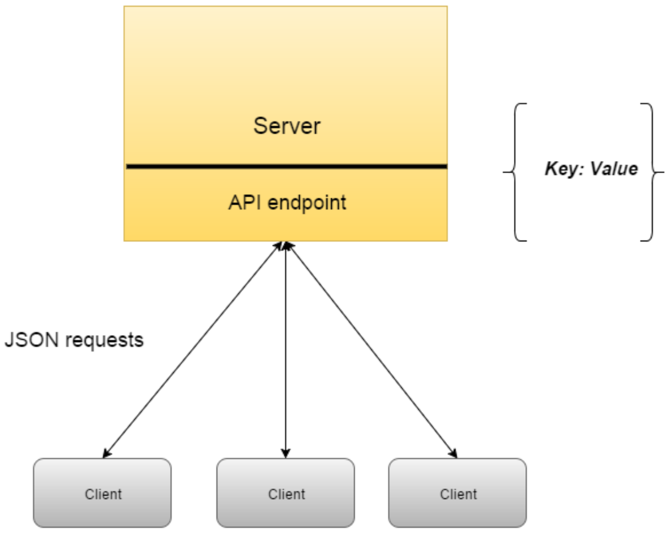
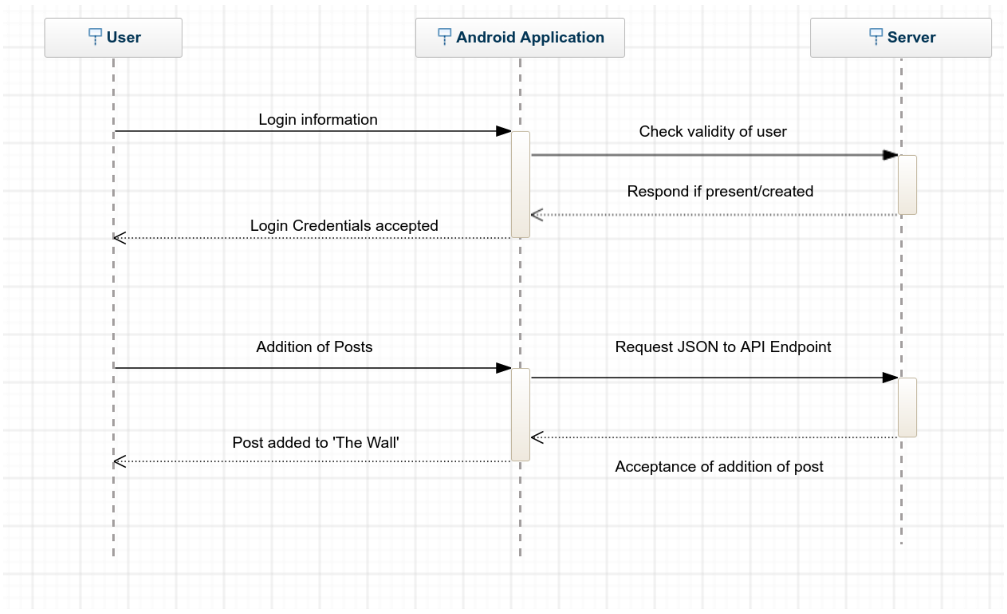


Figure 3.3: Sequence Diagram



User
Android Application
Server
Login information
Check validity of user
Respond if present/created
Login Credentials accepted
Addition of Posts
Request JSON to API Endpoint
Post added to 'The Wall'
Acceptance of addition of post

3.2 Description of components

The following is the system architecture, which explains the request response mechanism implemented by us, for this application.

3.2.1 User Class

The User class represents a general user(non-musician/non-band) who is looking to hire someone or post a job description. Such a user has a name and contact details only. The user is not prompted to input any other information apart from this.

PSPEC

We create the user by taking as input the following details:

- Name
- Email

We have tried to keep this class as simple as possible since there is not much to be done here. A user can add posts to the wall and he becomes the Author of that post.

Interface Description

new(Name: String, Email: String)
posts.add(PostObject)

Processing Detail

No complicated algorithms involved here. Profile creation and post creation are trivial operations to be performed.

Design Class Hierarchy

It has no parent/child class. It is a similar model as that of Musician/Band.

Restrictions

No particular restrictions.

Performance Issues

As mentioned before, there are not many performance issues involved in the same.

Design Constraints

No particular design constraints for this class.

Processing detail

- new()

  Creates a new User.

- posts.add(PostObject)

  If we provide the above method with a PostObject, a new post is created with the user as the author.

3.2.2 Post Class

The post class is meant to represent a post generated by a user and includes the Author of the post, a small description, Genre, Instrument, Location, Date Range and an is_-satisfied? flag. Each post is associated with a User, Musician or a Band and this is a polymorphic relationship in the database backend. An entity who can create a post can have multiple posts associated with it.

PSPEC

Whenever a user first creates a post a new PostObject is created which is responsible for storing information unique to that post. This includes:

- Author(User/Musician/Band)
- Description
- Genre
- Instrument
- Location
- Date_Range
- is_satisfied? (boolean)

Now the user who created the post can set/unset the is_satisfied? flag based on his query. After the Date Range has expired, the post is no longer shown on the interface.

Interface Description

new(Author: Object, Description: String, Genre: String, Instrument: String, Location: String, Date_Range: DateTime Object, is_satisfied?: Bool(optional))
update(Author: Object, Description: String, Genre: String, Instrument: String, Location: String, Date_Range: DateTime Object, is_satisfied?: Bool(optional))
delete()

Processing Detail

There are no algorithms associated with this class. The only methods are creators and destructors.

Design Class Hierarchy

It has no parent/child class. As mentioned before, each instance of a post has to be associated with an Author(User/Musician/Band).

Restrictions

No particular restrictions.

Performance Issues

Performance issues will arise only when we have a lot of posts to sort and search from. But the inbuilt methods available in Rails make sure that the most optimal algorithm is implemented.

Design Constraints

No particular design constraints for this class.

Processing detail

- new()

  Creates a new post. Author can be a UserObject, MusicianObject or BandObject.

- update()

  Based on the information provided in the input, the existing post is updated in the database.

- delete()

  An existing post is deleted.

3.2.3 Musician Class

This class can be thought of as the core business model of the application. This is the model we are considering the target audience to be. A musician is considered to be similar as a normal user but he has the option to add performances, previous jobs and skill details on his profile. He also has the privilege of creating posts similar to the User Class. A musician can join a Band as well. He can be a part of a band and have a current job. He can also post the location he is most active in and hence help gig search in a centralized manner. He can add certifications to his profile as well.

PSPEC

The following details are to be collected when we create the Musician class:

- Name
- Description
- Picture
- Contact
- Job
- Genre
- Instruments
- (past) Jobs
- Certifications (String)
- Awards(String)
- Songs(String)
- Online Profiles
- Favorite Artists(Bands/Musicians/String)
- Location

Interface Description

new(Name: String, Picture: ImageObject, Description: String, Contact Number: String, Current Job: JobObject, Genre: String Array, Instruments: String Array, Previous Jobs: JobObject Array, Certifications: String Array, Awards: String Array, Songs: String Array, Online Profile: StringArray, Favorite Artists: String Array/MusicianObject/BandObject, Location: String)
posts.add(PostObject)

Processing Detail

There are no algorithms associated with this class.

Design Class Hierarchy

It has no parent/child class. It is similar in functionality to a User/Band Class in certain aspects.

Restrictions

No particular restrictions.

Performance Issues

No real performance issues involved. Issues might arise due to no local storage of data. Each time the application is switched on, we pull data from the database using JSON queries. If the profile data is large, it might take some time before we can process the entire profile and view it on the screen.

Design Constraints

No particular design constraints for this class.

Processing detail

- new()

  Creates a new Musician object allowing him to login into the application.

- posts.add(PostObject)

  Adds a new post where the author is the Musician invoking the operation.

3.2.4 Band Class

The Band Class is also one of the core classes of the application. It can be thought of as an aggregation of Musicians. A Band class is considered as a separate entity when adding posts and other profile details. Although, it is made up of Musicians, they need not be registered on the application. We can have a standalone Band also. They also have the privilege to create posts and view profiles.

PSPEC

The following details are to be collected when we create the Band class:

- Name
- Description
- Picture
- Contact
- Genre
- Instruments
- Awards(String)

- Songs(String)
- Online Profiles
- Favorite Artists(Bands/Musicians/String)
- Location

Interface Description

new(Name: String, Picture: ImageObject, Description: String, Contact Number: String, Genre: String Array, Certifications: String Array, Awards: String Array, Songs: String Array, Online Profiles: StringArray, Favorite Artists: String Array/MusicianObject/BandObject, Location: String, Members: MusicianObject Array(Optional))
posts.add(PostObject)

Processing Detail

There are no algorithms associated with this class.

Design Class Hierarchy

It has no parent/child class. It is similar in functionality to a User/Musician Class in certain aspects.

Restrictions

No particular restrictions.

Performance Issues

No real performance issues involved. Issues might arise due to no local storage of data. Each time the application is switched on, we pull data from the database using JSON queries. If the profile data is large, it might take some time before we can process the entire profile and view it on the screen.

Design Constraints

No particular design constraints for this class.

Processing detail

- new()

  Creates a new Band object allowing it to login into the application.

- posts.add(PostObject)

  Adds a new post where the author is the Band invoking the operation.

3.2.5 Server Component

The server compoent of $\mu 2$ is responsible for storing and synchronizing data across devices. The server ensures that the information all handsets is consistent, and that the central database is kept up-to-date. Each time a request is to be made to the server, a JSON object containing the details of the request are sent to the server and the result is parsed later.

PSPEC

Data exchanged is in the JSON format. The data stored in the database is made available to the $\mu 2$ application via API endpoints.

Interface Description

- GET /users/id: Will return user information with id=2
- GET /users/: Will return a list of all users in the database
- GET /musicians/: Will return a list of all musicians in the database.
- POST /users/ : Request is parsed and a new user is created
- POST /posts/ : Request is parsed and a new post is created

Processing Detail

There are no complex algorithms involved in the above methods for addition and retrieval of users.

Design Class Hierarchy

The classes on the server will mirror those of the local application in terms of data member fields. They will be independent classes with no external dependencies, except for communication layers for database queries and HTTP communication.

Restrictions

Bandwidth and the presence of an internet connection are required in order to use these methods. If there is not one available, the methods will fail and the user will be notified. A caching function may be implemented in the future in order to batch updates that may need to be sent from the client until an internet connection is available.

Performance Issues

Performance issues are not of concern in this section. The data being exchanged is only plaintext, so bandwidth will never be an issue. The exception to this assumption is the optional feature of addition of images, which would require potentially large files to be transferred to and from the server. This may create a bottleneck, or prove to be too taxing on network throughput. This feature, however, is not designated for inclusion in the first release of the application.

Design Constraints

The current design for the server component requires that each method be able to handle any object of any type. Thus, the serialization and deserialization functions will need to be incredibly robust and cannot rely on any assumptions about the data until it is decoded. Each method must contain logic to determine the type of object that it has deserialized and to call other server-side methods to handle these objects accordingly.

4 User interface design

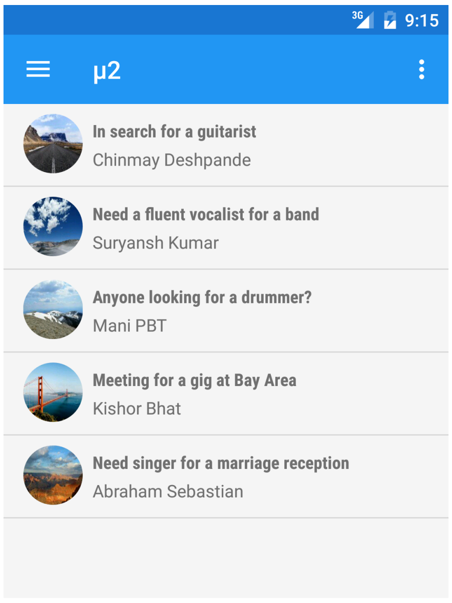
A description of the user interface design of the software is presented.
The underlying principle behind our interface design paradigm, is "Simplicity is the ultimate Sophistication". To enable all kinds of users to effectively utilise all features of our application, we plan to keep the user interface simple and intuitive.

4.1 Description of User interface

4.1.1 Objects and actions

All screen objects and actions are identified.
We have the following screens:

Figure 4.1: Wall



9:15

μ2

In search for a guitarist
Chinmay Deshpande

Need a fluent vocalist for a band
Suryansh Kumar

Anyone looking for a drummer?
Mani PBT

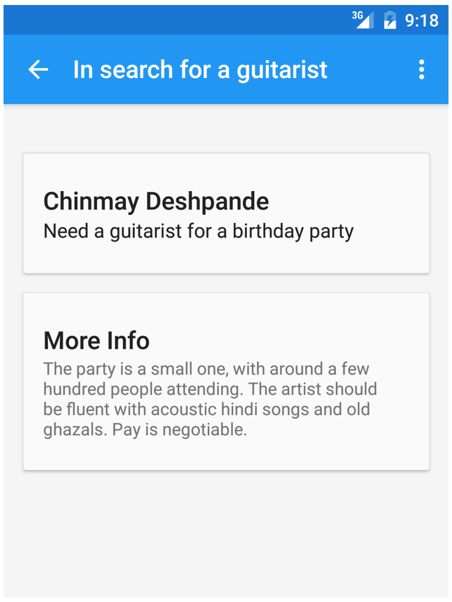
Meeting for a gig at Bay Area
Kishor Bhat

Need singer for a marriage reception
Abraham Sebastian

User registration and Welcome Screen

When the application is installed and run for the very first time, the user is presented an initial registration/welcome screen. This screen prompts the user to create an account on the server using the email address associated with his/her Google account. The user also enters a display name which will be the name that is shown as their handle. After user inputs all these details, he is prompted to go to the edit profile screen.

Figure 4.2: Post Description

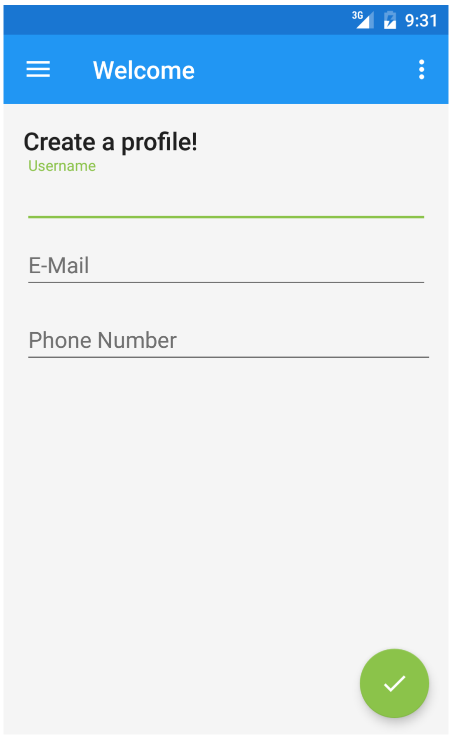

9:18

In search for a guitarist

Chinmay Deshpande
Need a guitarist for a birthday party

More Info
The party is a small one, with around a few hundred people attending. The artist should be fluent with acoustic hindi songs and old ghazals. Pay is negotiable.

Figure 4.3: Creation of Profile


9:31

Welcome

Create a profile!
Username

E-Mail

Phone Number

Edit Profile Screen

Based on the type of user, he will be prompted to input the various details that are required for all users of the application. Completing this process will modify and store an account for the user on the server, enabling all of the application's synchronization capabilities. After this is completed, the user is taken to his profile screen, to visualise how his profile looks to the outside world.

View Own Profile Screen

Here the user views his own profile, which consists of all details he had filled in during registration, his display picture and gets a view of how his profile looks to the outside world From here, he can either view the wall, search for something or view the his own posts,by touching the respective buttons/icons.

The Wall

The wall is an aggregation of all posts which have been published by users of the application, stating their requirements ot advertising themselves. It is shown as a list of posts, with their title, and clicking on the post takes them to the screen showing detailed description. Users can also find specific posts relevant to them on the wall, using tags. From the wall, they can also create their own post and publish it for all users to view.

Search Bar

The search bar is available on all every profile screen and the wall too. Here, a user can search for a musician, or a band based on whatever his requirements are. The result of his search queries is displayed as a list.

View Post Details

Once the user clicks on the post from the wall, he is sent to another screen which shows the entire description of the post, its tags and the specific requirements it asks for. From here the user can click on the profile of the post's creator and contact him.

View Other's Profile

Here, the user can view the profile of other musicians/bands and in case his interests match, he can contact them. This screen shows all the details which the musician/band had entered when registering and later updated, which includes their achievements, awards, links to online audio/video recordings etc, which they want to showcase to the world.

4.1.2 Interface Design Rules

Conventions and standards used for designing/implementing the user interface are stated. (You could list the eight golden rules mentioned in lecture.)

We will be taking into consideration Shneiderman's Eight Golden Rules of Design, when designing this application:

- Strive for consistency
- Enable frequent users to use shortcut
- Offer informative feedback
- Design dialog to yield closure
- Offer simple error handling
- Permit easy reversal of actions
- Support internal locus of control
- Reduce short-term memory load

5 Restrictions, Limitations and Constraints

As time is a limiting factor, the optional features previously mentioned in the SRS document are not discussed at all in this document. This is due to the fact that these features will likely not be implemented within the allotted time for this projects completion. However, as a result of the highly modular design and organization of data – as well as unlimited expansion potential on the server side – implementing these optional features at a later date would be arguably easier than incorporating them into the first design.

Another limitation of the software is the lack of a web interface. While not included in the optional features (as it may be considered a product of its own), a web interface to the $\mu 2$ system would allow users with or without the Android application to use a web interface with all of the capabilities of the $\mu 2$ application. Once the server for the client application has been developed, it would be possible to implement this interface with relative ease.

A constraint that is frequently mentioned in this document as well as the SRS is the requirement for the user to have internet access on their Android client. This is essential, as all data mutating actions make a call to the server in order to complete that action. A potential solution is an offline queue that stores actions to be sent to the server once an active internet connection is established. If any conflicting information has been uploaded by other users during the first users offline time, all of the first user's queued actions are discarded and the user is notified of this (and presented with the most recently changed data).

As the application frequently queries a server over the internet, care must be taken to ensure that large amounts of traffic are not being sent or received by our application, as this may dissuade users with costly data plans from using our application. Currently, we do not anticipate this being a problem, due our use of encoded JSON messages for passing data between the client and the server. However, if during testing we find our usage is too high, we will begin to investigate ways to decrease this usage. Possible ideas include requesting specific fields that have changed, rather than entire objects, and also compressing the JSON objects that are sent between client and server.

Another constraint imposed on the user of this software is that they must have an email address. This is used a unique identifier for each user and also provides an avenue for a user to re-register their account in the event that they switch phones (as the ID of the phone will provide authentication for a user).

6 Testing

Test strategy and preliminary test case specification are presented in this section.

Each feature/functionality will be tested individually at first, using the principles of Unit Testing.
Unit Testing focuses testing on the function or software module. It concentrates on the internal processing logic and data structures.

After performing unit testing we will move on to integration testing. This will focus on inputs and outputs and how well the components fit and work together.

Then we perform validation testing. The requirements are validated against the constructed software. This provides final assurance that the software meets all functional, behavioral, and performance requirements.

Now, at the end, we finally perform system testing, where the software and the entire system as a whole is tested. This verifies that all system elements (software, hardware, people, databases) mesh properly and that overall system function and performance is achieved.

6.1 Classes of tests

We perform the following types of testing:

6.1.1 White Box Testing

While each class is being implemented, the developer implementing that class will test to make sure each function is working.Knowing the internal workings of a class, we test that all internal operations are performed according to specifications and all internal components have been exercised. The developer is fully responsible for debugging his/her own code because the overhead of sharing code between developers has been deemed too costly. However, all code will be accessible through the provided version control system, so this rule may be violated if needed.
White box testing Involves tests that concentrate on close examination of procedural detail and all the logical paths through the software are tested.

6.1.2 Black Box Testing

This will be done after all the components are assembled, and will consist of running through all possible situations that may occur in the use of $\mu 2$. Black box testing involves knowing the specified function that a product has been designed to perform, test to see if that function is fully operational and error free. It includes tests that are conducted at the software interface. This is not concerned with internal logical structure of the software.

6.2 Expected software responses

Account Creation

- Description: correct data input
  Input: Valid email, that is not already on the server
  Output: Account is created on server, user taken to edit profile.

- Description: incorrect data input
  Input: invalid email
  Output: Account is not created, user is asked for different email

- Description: incorrect data input
  Input: email already exists on server and is an online account
  Output: Account is not created, user is asked for different email and notified email already exist

Connecting to Server

- Description: connection is established.
  Input: Device tries to access the server, and succeeds
  Output: Device pushes and pulls information as normal

- Description: connection cannot be established
  Input: Device tries to access server and fails
  Output: User is alerted that they are offline, no data is transferred to server. All changes are stored locally and temporarily.

Creating a profile

- Description: correct data input
  Input: Valid details, all input fields validated
  Output: Account is created on server, user taken to view profile.

- Description: incorrect data input
  Input: invalid input in textbox and other input fields
  Output: Profile is not updated, user is asked for valid input

Creation of a post

- Description: correct data input
  Input: Valid description, correctly filled in all details and tags
  Output: Post is created and published on the Wall, user is taken to Wall

- Description: incorrect data input
  Input: Empty fields, no tags listed
  Output:Post is not created, user is asked to re enter details of post

- Description: incorrect data input
  Input: A very similar post already exists on the Wall by the same user
  Output: Post is not created, user is asked re enter details of post

6.3 Performance bounds

Due to fact that the application is very demanding with respect to resources, execution time for all local actions should be negligible. This includes screen navigation, group management, bill/transaction creation, etc. Also, since data is exchanged with the server in small plaintext messages, interactions between the client and server should also take very little time.

In relation to the server component, it must uphold acceptable performance ability when negotiating the passing of information between server and the client. For the scope of this project, a Rails application implemented on a standard virtual dedicated server will suffice. As this server is only performing database calls, it is not processing-intensive. The only expected issue involving the server (in terms of performance) involves the amount of groups and devices performing an interaction. This would be easily throttled by upgrading the bandwidth of the server. However, this will not be an issue for this project. If the application is launched on the Android market, the server system will be revamped entirely.

6.4 Identification of critical components

Those components that are critical and demand particular attention during testing are identified.

Here, we identify those modules and classes which are most frequently called and used during the usage of the application. Then we perform extensive testing, checking all boundary conditions and corner cases repeatedly to ensure that these critical components are foolproof. It is highly important that these critical components remain free from errors, as this ensures that the basic structure and important functionalities of the application remain stable and the app does not crash.