# Mixus

## Proximity-based music platform


- Written by:
  - Daniel Mittelman
  - Tomer Brosh
- Supervisor: Kfir Lev-Ari

- Submission date: 03/08/2015

# Table of Contents

# 1. Introduction

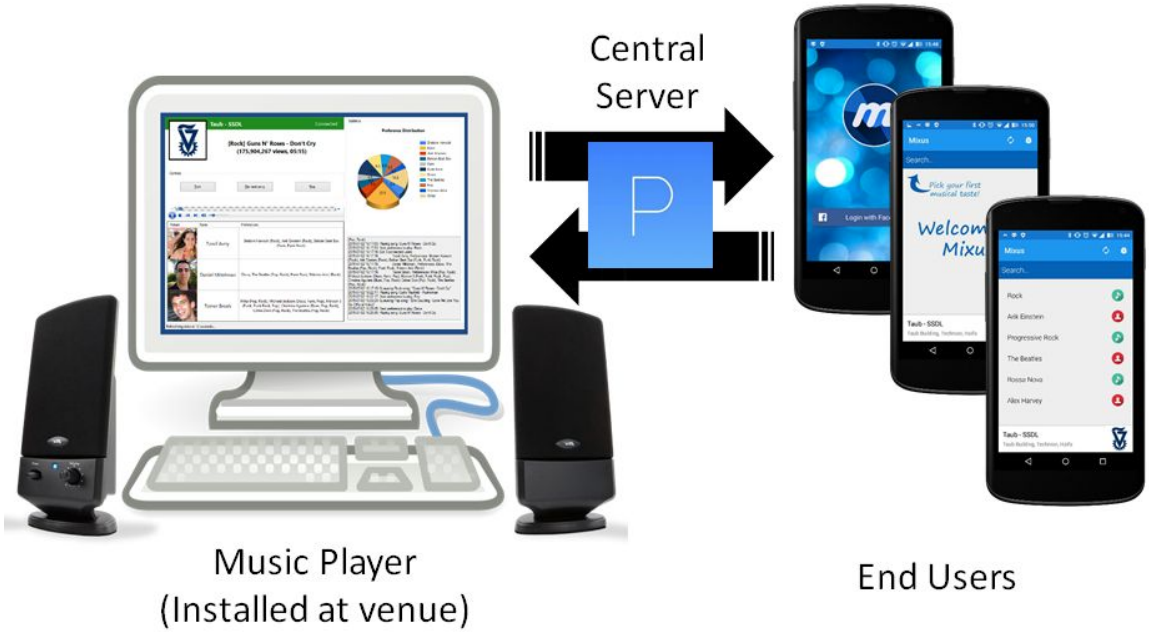
Today, we're all surrounded with music. We have virtually unrestricted access to every song, every artist and every album ever recorded. We all have different music tastes and preferences, and we've invented ways to hear the music we like, wherever we are and whenever we want. Our project's goal is to bring this level of personalization to places where it doesn't yet exist - bars, restaurants, coffee shops and other public places, who usually play music throughout the day, yet do not consider the fact that different people have different tastes, and would associate music that they like with an overall good experience.

***Mixus*** is a proximity-based music platform that enables users to directly influence the music that's being played at a public location, by generating a unique playlist based on the musical preferences of all the users within that location.

Users interact with the system using our mobile application (Android-only at this point), and businesses can utilize the service by installing our music player on a regular Windows PC (many of which already have one in their business) and registering it as a *deployment*. Then, once a user enters a deployment, the mobile app communicates the presence of the user to the music player, and that, in turn, recalculates the playlist to suit the new user as well.

Chapters 2-4 of this report offer a high-level overview of the solution, along with some of the basic questions we had faced. Chapters 5-7 offer an in-depth review of our work process, and dive into implementation details of the different components. We highly recommend that you read chapter 5 before reading chapters 6-7, as it provides background to those chapters and introduces technical terms that will appear later on.

## 2. Solution Architecture and Design Guidelines



## 2.1. Solution Components

Mixus comprises three major components:

**Mobile application**

Allows end-users to interact with the system: Communicate their musical preferences, report whether they're located in the vicinity of a Mixus music player and get real-time information about the venue and the playlist.

**Music player**

Installed by a business owner on a Windows computer at the business venue. The player has several responsibilities:

- Manage the list of users currently located within the business
- Obtain the preferences of each user, and calculate the preference distribution of all users
- Find, download and enqueue the next song based on the users' preferences, ensuring that the next song is always available and ready
- Broadcast the playlist to the users, including the name of the current and next song, and the current song's progress

**Central server**

The mobile app and the music player do not communicate directly. Instead, all communication is done via a central server, in charge of passing information from the player to the app (and vice versa), handling user authentication and user data storage, and other automated tasks that require a server.

## 2.2. Design Guidelines

We've designed Mixus with several important principles in mind:

- ➔ **Minimal interaction**: Do not disturb the user while he's engaged in social activities. Our goal is to provide an enhancing experience, not a disruptive one.
- ➔ **All users have an equal say**: Give all users an equal say, regardless of how many preferences one has.
- ➔ **Balanced transparency**: Share as much relevant information as possible with the user (and business owner) and display it wisely, without overcrowding the user's display.
- ➔ **Graceful handling of connection loss**: Mobile Internet and home routers occasionally fail, so know how to gracefully handle events of loss of Internet connectivity for short periods of time.

The solution components, and the steps we've taken to realize our design guidelines, will be covered thoroughly in the next chapters of this report.

# 3. Fundamental Dilemmas

Before writing any code, we had to discuss and resolve a few basic questions that would determine how we continue.

## 3.1. What is a Music Preference?

One of the most basic questions we had to answer was *"How do we classify user preferences?"*. To answer this question, we came up with two different approaches:

- ➔ **The narrow approach** - A preference can only be a musical genre. This would be easier to implement, however would not provide a fine enough granularity to precisely reflect users' musical tastes.
- ➔ **The wide approach** - A preference can be either a musical genre, an artist, a song or an album. This method would allow users to portray their musical preferences very accurately, however would be very complex to implement, especially in the timeframe of a project. Also, this level of freedom may not be required by most users.

Our solution was to find a middle ground, saying that a preference can either be a musical genre or an artist.

While implementation will be feasible within the given timeframe, in our opinion, this approach would also best mirror the human response to the question "*What music do you like?*". Another advantage is that we would be able to limit user input to known genre and artist names, allowing us to avoid nominal ambiguity (for example - some artists have their albums named after them).

## 3.2. How Do We Get the Music?

For our music source, we considered several options:

- Using a fixed set of local music files, just like most business owners do right now when playing music. This has the advantage of using music of which the business owner already approves. However, there are also serious disadvantages - we still need to get the local music files somehow, and it limits the variety of songs that can be played in a venue.

- Using an online commercial service designed to stream music, such as Pandora, Grooveshark and others. The advantage is that these services already contain a large variety of music and support streaming it to clients, but to our knowledge they do not offer services to applications like Mixus, which would offer the music to other business venues.

- Using YouTube. Some obvious advantages of YouTube are that it probably contains the largest variety of songs possible, it is constantly updated with new content, and that it's free. In addition, we can use the built-in "like count" and "view count" mechanisms to judge which music videos our users are more likely to enjoy. Even more importantly, YouTube integrates with a service called Freebase, which allows us to perform smart searches for videos related to musical entities (artists, genres, etc). Information about Freebase can be found in section 5.2. Freebase.

Our chosen solution was YouTube. Its listed advantages helped us greatly later in the implementation of Mixus.

## 3.3. How To Synchronize End-Users With the Music Player?

Another question we asked ourselves was how to pass information between the mobile app and the player - quickly, reliably, and with the possibility to store user-based information. We decided to prepare two models - a *Peer-to-Peer model* and a *centralized model*, and draft a pros and cons list to pick the most suitable model for our needs.

- The Peer-to-Peer model: According to the first model, clients communicate directly with other nearby clients and with the player. Communication is based on Bluetooth, allowing clients to quickly (and transparently) detect nearby Mixus-enabled devices and initiate a communication channel, until the devices are no longer close. We specifically looked at an open-source platform called *AllJoyn* which would provide us the necessary capabilities.

- The centralized model: All communication is handled by a central server, and clients and players connect directly to the server to retrieve information and send commands.

We then decided to put these models side by side and review their pros and cons:

| Centralized model (Server) | Peer-to-Peer model (AllJoyn) |
|---|---|
| A central server would allow us to have **real-time monitoring** of the system, log collection and analysis, and **production of extensive usage statistics**. Furthermore, it would allow users to know where Mixus-enabled venues are located (by making this information accessible via the mobile app or a designated website) | Independence from a central server increases the system's durability, since there's **no single point of failure** that can affect the entire system altogether |
| Registering the venue's wireless access point would allow the application, using a passive Wi-Fi scan, to detect proximity to a Mixus-enabled venue and associate the device with the venue. This would not require the user to connect to the venue's access point, an action which might pose a security risk to the device, and thereby we would be **preventing a potential security risk** | Using AllJoyn's Onboarding API would allow the music player to **automatically connect to mobile devices** that have entered its wireless network's range (by connecting these devices automatically to the Wi-Fi), so that no further action is required from the user for basic usage of the application, and in a way that does not require mobile data transfer |
| | **Availability** - Installation of the system in a new venue is quick and easy and requires no additional steps to the register the venue. Just download and install the player software |

After reviewing both models, we decided to go with the **centralized model** approach: Aside from giving us detailed statistics and information about the system as a whole, we decided that requiring users to connect to a foreign (and usually unencrypted) Wi-Fi

access point without their permission would be an unsuitable solution that would put users' private information at risk.

# 4. Solution Overview

The purpose of this section is to explain how the solution's components work together, and how they look and feel.

## 4.1. Music Player

We chose to implement the Mixus Player as a Windows application, because we believe most current music playing solutions in business venues already use a Windows computer connected to speakers, so the Mixus Player could be used as a drop-in replacement.

The player is written in C#, and uses the Windows Media Player interface to play music, which comes preinstalled in Windows computers, thus would not require additional music-playing software.

The player's responsibilities are the following:

- Continuously fetch the list of users who are connected to the business venue, along with each user's musical preferences
- Create a list of all the preferences of connected users, and give each preference a weight. For each preference, an algorithm decides its weight while taking into consideration the number of users that chose the preference.
- Fetch songs from YouTube according to the preferences list, so that preferences with higher weights will be used more often. The player keeps a short playlist (about 3 songs) at all times, and each time a song ends, it downloads another one to keep the playlist's length the same. The playlist is chosen to be long enough so that when a song finishes playing, the next song is already fetched, and short enough so that any change in the list of connected users or in their

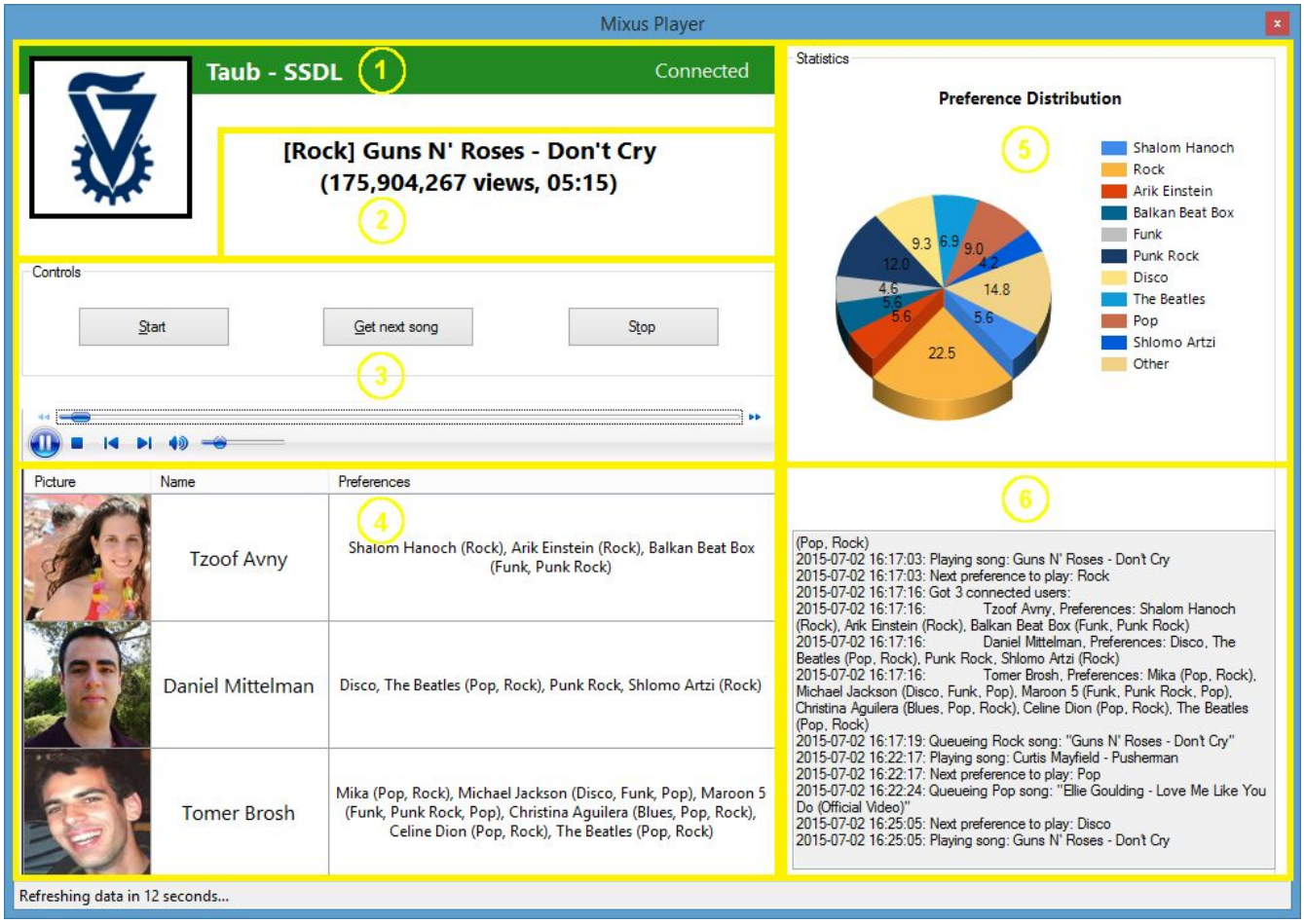
preferences can have an immediate effect.

- Display information to the business owner:
  - The player's status (playing, stopped, paused, etc)
  - The current song's information (title, length, YouTube views count, etc), and the the next song's title
  - The users who are connected to this business venue
  - Aggregated information about the preferences currently affecting this player (the preferences of the connected users, along with each preference's importance)

An introduction to the Mixus Player's user interface:

When the player is run, it first needs to identify the venue in which it is located. This is currently done by recognizing the venue's wireless network (or one of them), and then querying the server for the venue associated with that network. The network can be identified automatically on computers with Wi-Fi access, or the wireless network's BSSID address can be typed manually (for more information about the way Mixus identifies wireless networks, see 5.3. Wireless Networks Identification).


After the network association process completes, the main player window is displayed. In this window, the business owner receives constant information about the player's status.




The main view consists of the following parts, marked in yellow in the above screenshot:

1. Connected Mixus deployment (the business venue associated with the player)
2. The current song title, length and view count, and the next song's title (when available)
3. Media player controls (play, pause, stop, etc)
4. Information about the currently connected users
5. A pie chart of musical preferences. The chart shows the weight that is given to each preference (you can learn more about how the weights are calculated in section 6.4. The music selection algorithm). The preference which the currently

playing song was selected by is emphasized by an "exploded" piece in the pie chart

6. Textual status messages, which may be used to debug the player or to better understand how it works

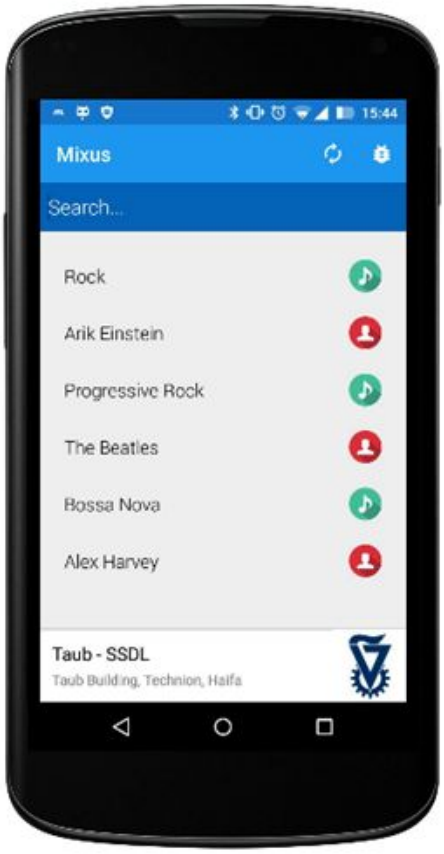
## 4.2. Mobile App

We chose to design the mobile application for Android devices, and write the code in Java using the native Android SDK.

Upon first access, the user is presented a login screen. We chose to use **Facebook Login**, since it is a familiar login system, with which most users would feel comfortable, and which doesn't require a username and password to be entered. In addition, it allows us to reuse the login token in following sessions, so the app won't have to ask users to login each time they open the application.

The main screen is designed to display several key pieces of information to the user:

- **Preferences screen** - The main screen area displays the list of chosen user preferences (i.e. genres and artists). A user can add a preference to the list by searching for it in the top search bar, and remove a preference by long-clicking on it and then tapping "*Unlike*"




- **Sliding panel** - The bottom portion of the screen is a sliding drawer (the user can slide it up or down with his finger). While collapsed, it displays the name and image of the venue in which the user is currently located (or "*Not connected*" if the user is not in a Mixus-enabled business).

  Once expanded, it displays additional key information:


  - **Playback information** - The name of the current song playing in the background, its progress and the name of the next song to be played
  - **Business information** - The user can tap on one of the business links at the right side of the drawer to launch the corresponding link (e.g. the business' Facebook page, Twitter hashtag, website, etc).

# 5. Technological Background

This section covers in detail the key technologies we've used in order to build the project. These will later be referenced in 6. Implementation Details.

## 5.1. Parse


Parse is a large-scale cloud service provider that allows application developers to quickly and easily integrate remote database storage, user authentication and user management capabilities. In case of mobile applications, it also provides a push messaging service to send real-time information to devices.

Parse also provides a feature called *Cloud Code*, enabling developers to deploy server-side functions and tasks to be triggered in response to events, or executed at a predetermined schedule. These functions are written in JavaScript, and can be extended using several third-party libraries (like *Underscore*, *SendGrid*, *Express*, *Twilio* and more).

Parse was acquired by Facebook in April 2013, and Parse's servers now operate from within Facebook's server farms - making it a reliable and fast service provider.

While providing all cloud services on their remote servers, Parse enables developers to integrate their services directly into their software using a set of SDKs designed for multiple platforms and programming languages. Naturally, we used the Parse SDK for C#.NET and for Android.

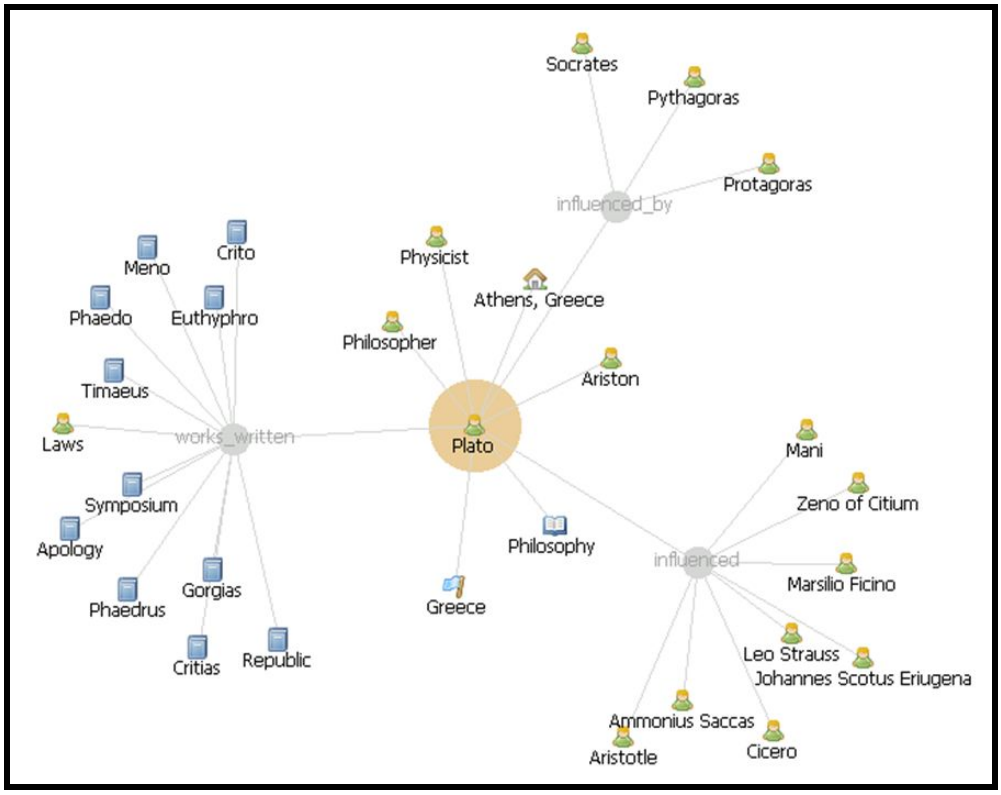
For more information about how we used Parse, refer to 6.1. Parse Integration

## 5.2. Freebase

From the Freebase article on Wikipedia:

> ***Freebase*** *is a large collaborative knowledge base consisting of data composed mainly by its community members. It is an online collection of structured data harvested from many sources, including individual, user-submitted wiki contributions. Freebase aims to create a global resource which allows people (and machines) to access common information more effectively. It was developed by the American software company Metaweb and has been running publicly since March 2007. Metaweb was acquired by Google in a private sale announced July 16, 2010. Google's Knowledge Graph is powered in part by Freebase.*
>
> *Freebase data is freely available for commercial and non-commercial use under a Creative Commons Attribution License, and an open API, RDF endpoint, and database dump are provided for programmers.*

Freebase is an open source service that contains a very large knowledge base, which is fundamentally a directed graph. The nodes in the graph represent entities, such as people, places, and concepts, and the edges in the graph represent relations between the entities.




For example, there is an entity that represents the philosopher Plato, which connects via a "profession" edge to a "Philosopher" entity, and via "influenced_by" edges to the entities for Socrates, Pythagoras and Protagoras.

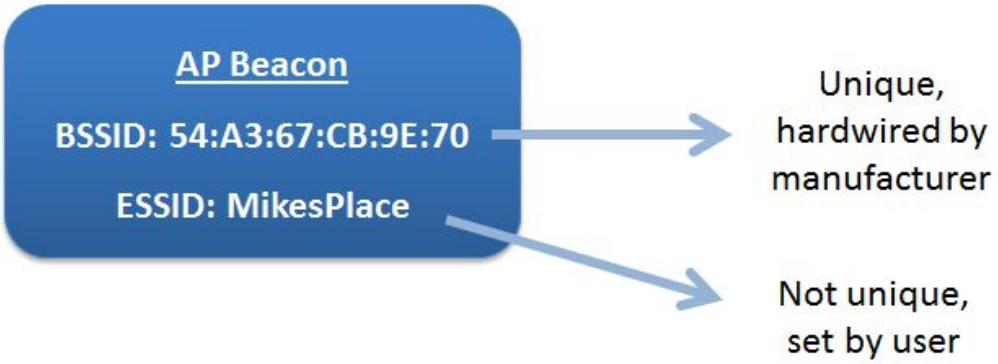
For our project, Freebase is interesting mainly because it contains entities for musical genres and artists, and relations between them (which artist is part of which genre).

## 5.3. Wireless Networks Identification

Wireless networks play a major role in how Mixus identifies whether a user is located in a Mixus-enabled business, and in which one.
Wireless networks are formed by wireless access points ("*Wi-Fi AP*"), more commonly known by the misnomer "*home routers*". In order to be identifiable, these APs broadcast *beacon frames* (usually several times a second) containing information about the access point.

We look at two key data fields in the beacon frames:

- **BSSID** (**B**asic **S**ervice **S**et **Id**entification): The BSSID is the access point's hardware address. It is **hardcoded** by the manufacturer, and it is **unique** to the access point, meaning there's no other AP in the world with the same BSSID.
- **ESSID** (**E**xtended **S**ervice **S**et **Id**entification): The ESSID is the access point's published name, which is easily distinguishable by humans. It is neither unique nor hardcoded (it can be modified by the AP's owner at any given time).




For the reasons above, Mixus uses the BSSID to identify an access point. It is worth noting that this method of identification allows the application to notify the system with its whereabouts (in terms of access points and not accurate location, like GPS) **without** having to connect to the venue's access point, but by merely doing a passive Wi-Fi scan.

## 5.4. NTP

From the NTP article on Wikipedia:

> ***Network Time Protocol** (**NTP**) is a networking protocol for clock synchronization between computer systems over packet-switched, variable-latency data networks.*
>
> *...*
>
> *NTP uses a hierarchical, semi-layered system of time sources. Each level of this hierarchy is termed a "stratum" and is assigned a number starting with zero at the top. A server synchronized to a stratum n server will be running at stratum n + 1. The number represents the distance from the reference clock and is used to prevent cyclical dependencies in the hierarchy.*

The NTP protocol is usually used in a client-server configuration. Essentially, the protocol is used to adjust the client's local clock according to the server's clock. The server's *stratum* number indicates how directly it is connected to a reference clock server: Reference clock servers are the "*stratum 0*" servers, and they keep time using a very high accuracy atomic clock. Several "stratum 1" servers are connected to each stratum 0 server, several stratum 2 are connected to each stratum 1 server, and so forth.

The clock synchronization algorithm itself is pretty simple: It consists of sending a request to an NTP server, and waiting for the response, which contains the server's time. The problem created by this approach is that the client still needs to compensate for the time it takes for the request packet to reach the server, the time it takes the server to calculate a response and the time it takes the response packet to reach the client.

The details of the clock synchronization protocol are better explained in this paragraph, taken from Wikipedia:

> *A typical NTP client will regularly poll three or more servers on diverse networks. To synchronize its clock with a remote server, the client must compute the round-trip delay time and the offset. The round-trip delay δ is computed as*
>
> $$\delta = (t_3 - t_0) - (t_2 - t_1)$$
>
> *where*
>
> *$t_0$ is the client's timestamp of the request packet transmission,*
>
> *$t_1$ is the server's timestamp of the request packet reception,*
>
> *$t_2$ is the server's timestamp of the response packet transmission and*
>
> *$t_3$ is the client's timestamp of the response packet reception.*
>
> *Therefore,*
>
> *$t_3 - t_0$ is the time elapsed on the client side between the emission of the request packet and the reception of the response packet and*
>
> *$t_2 - t_1$ is the time the server waited before sending the answer.*
>
> *The offset θ is given by*
>
> $$\theta = \frac{(t_1 - t_0) + (t_2 - t_3)}{2}$$

After calculating the offset (θ in the above explanation), the client needs to add it to the relevant server timestamp it has received, and in that way it compensates for the time lost over the network and over calculating the server's response.

## 5.5. Google Play Services

From Google Play:

> *Google Play services is used to update Google apps and apps from Google Play. This component provides core functionality like authentication to your Google services, synchronized contacts, access to all the latest user privacy settings, and higher quality, lower-powered location based services.*
> *Google Play services also enhances your app experience. It speeds up offline searches, provides more immersive maps, and improves gaming experiences...*

*Google Play Services* is a wide set of proprietary APIs made available by Google for Android developers. These include many APIs allowing access to Google services (like *Google Drive*, *Google Analytics*, *Gmail*, *Google Fit*...)

Mixus uses an API found in Google Play Services called the *Activity Recognition API*. This API utilizes the device's low-energy sensors (mainly the accelerometer) to determine the type of **physical** activity performed by the device's owner. The API can distinguish, with a high level of accuracy, between standing still, walking, running, driving a vehicle and other daily activities, by analyzing the device's acceleration and trying to match it to a known pattern. Information about the user's current activity is provided alongside a value of "precision", reflecting the API's level of certainty that this is actually the user's activity, allowing developers to filter out results under a certain threshold.

For more information about where this capability is used, refer to 6.2. Battery management.

## 5.6. Android Wear

From Wikipedia:

> ***Android Wear*** *is a version of Google's Android operating system designed for smartwatches and other wearables. By pairing with mobile phones running Android version 4.3+, Android Wear integrates Google Now technology and mobile notifications into a smartwatch design. It also adds the ability to download apps from the Google Play Store.*
>
> *The platform was announced on March 18, 2014, along with the release of a developer preview. At the same time, companies such as Motorola, Samsung, LG, HTC and Asus were announced as partners… On December 10, 2014, an update started to roll out which, added new features, including a watch face API and changed the software to be based on Android 5.0 "Lollipop". The latest software supports both Bluetooth and Wi-Fi connectivity, as well as a range of features and apps.*
>
> *Watch face styles include round, square and rectangular…*
>
> *Hardware manufacturing partners include ASUS, Broadcom, Fossil, HTC, Intel, LG, MediaTek, Imagination Technologies,Motorola, Qualcomm, and Samsung.*
>
> *In the first six months of availability, Canalys estimates that over 720,000 Android Wear smartwatches were shipped. As of May 15, 2015, Android Wear had between 1 and 5 million app installations.*

Mixus features Android Wear integration, from which users can benefit if they own an Android device paired with an Android Wear smartwatch. For more information about Mixus' watch integration, refer to 6.5. Smartwatch integration.

# 6. Implementation Details

Now that we have covered the necessary technical background, this section gives an in-depth explanation of the implementation of the features presented in 4. Solution overview.

## 6.1. Parse Integration

We used several services provided by Parse, mainly:

- **NoSQL database storage** - Allows the player and mobile application to store information about the user, whether he's in a Mixus-enabled venue (and which one) and his musical preferences. Also, it allows storing information about the venues themselves.
- **Push service** - Allows the player to deliver real-time playback information to relevant devices.
- **User management** - Allows the mobile application to authenticate logged-in users and give them access to their data, while keeping other users' information private. Also, it allows for easy integration with the Facebook Login system.
- **Cloud Code** - Allows the server to do regular maintenance and respond to events. Some examples:
    - Download the user's Facebook profile image upon registration
    - Clear stale users that did not report their presence in a venue for a long period of time, and it is safe to assume they've left the place
    - Send a push message to clients when a player reports an update in the playlist or of the current song

Usage of Parse in detail:

1. **Database storage** - Also called "*Core*", Parse uses a NoSQL approach based on *models*. A model is an entity template that has *columns* (i.e. fields) and allows to save information in many different forms - numbers, text, objects (JSON objects, a combination of lists and dictionaries), pointers and even files, as long as it conforms to the specific model under which it is stored.

   Parse provides four built-in models: *Installation*, *Role*, *Session* and *User*. In addition to the built-in ones, we created several custom models to store our own information:

   1. **Deployment** - Represents a venue. Contains the following columns:
      a. placeName (String): The name of the venue
      b. placeAddress (String): The venue's address
      c. currentMedia (Object): The current playlist and player state (playing, paused, etc)
      d. onlineData (Object): A dictionary containing links to the business' online presence sites (Facebook page, website…)
      e. placeAvatar (File): An image of the venue, displayed in both the application and the player. The image's size should be 280x280px.
   2. **WirelessNetwork** - Represents a wireless access point. Contains the following columns:
      a. bssid (String): The AP's address
      b. description (String): An optional description of the AP, for internal use only
      c. deployment (Pointer): A pointer to the deployment where the access point is located
   3. **PreferenceInfo** - Represents a generic preference (genre/artist). Contains the following columns:

a. type (String): One of the values {genre, artist}
b. displayName (String): The preference's name
c. freebaseMid (String): The corresponding ID in Freebase
d. linkedPrefs (Array): An array of pointers to other PreferenceInfo objects that are relevant or similar
e. popularity (Number): Unused, may be used in the future

4. **UserPreference** - Represents a preference of a specific user. Contains the following columns:
    a. info (Pointer): A pointer to the corresponding PreferenceInfo object, containing information about the preference itself
    b. strength (Number): A number in the range [0,100] describing the preference's strength. Currently only the value 100 is used.

After creating the custom models, we were able to add custom fields to the **User** built-in model:

1. name (String) - The user's real name
2. profileImage (File) - The user's profile image
3. associatedDeployment (Pointer) - A pointer to the deployment object representing the venue the user is currently in, or *undefined* if the user is not located at a Mixus-enabled business.
4. musicPreferences (Array) - An array of pointers to the UserPreference objects belonging to the user. This field is used to store the user's list of music preferences.

2. **Push service** - Parse's push service allows different applications to push short payloads to mobile devices running Android, iOS or Windows Phone. We used the push service to send real-time updates from the music player to devices about changes in the playlist. Push messages are sent each time an update is

available, including when the player downloads a new song, when a new song starts playing, and even when the business' owner changes the player's position in the current song.

Push messages are sent in Parse by "*channels*". A mobile device can subscribe or unsubscribe from a channel, and only devices that are subscribed to a specific channel will receive push messages sent to that channel. We used that feature to allocate a channel to each venue, and then allow each venue to push messages to its users.

Push messages are not sent directly from the music players. Instead, players update their Deployment object on Parse with the following information:

- Message - A JSON object describing the current playlist
- Channel - The venue's channel
- Timestamp - Refer to 6.3. Time synchronization

Push messages that are received by users' mobile devices are immediately processed, and the mobile app then displays information about the playlist to the users, including the current and next song, and the progress of the current song.

3. **User management** - We also used Parse for user management and authentication. This is done via the Facebook Login integration (since Parse was bought by Facebook, it only makes sense). We added the Facebook Login feature by including the Facebook SDK in the project, and integrated that with Parse, so that User objects are created and managed almost automatically.

4. **Cloud Code** - We used Parse's Cloud Code feature to upload JS functions that are either triggered by events (updating a record in the database) or according

to a schedule. These functions allowed us to add some more advanced capabilities to the server:

a. disassociate_users_on_timeout - A scheduled job executed every 30 minutes. The function inspects all the users in the system and cleans out users that are said to be located in a venue, but did not report their presence for a long time. User disassociation from a venue happens when a device reports it's no longer in range of the venue's wireless access point, and this function allows us to ensure that the user will be disassociated if his device is no longer able to report his location (for example, if the device's battery ran out).
b. fetch_image_on_fb_registration - A function that's triggered when a new user was created. It checks if the registration process was done via Facebook (currently it should always run), and then queries Facebook's Graph API and retrieves the user's email address and profile image, and stores it in the User object.
c. get_freebase_linked_prefs - A function that's triggered when a new preference is added to the system. The function contacts Freebase and tries to find related genres or artists, and link them to the newly added preference.
d. push_notification_on_media_change - A function that's triggered when a music player has notified the server that its playlist has changed. The function then sends a push notification to all the users in the venue's channel with the playlist information.

## 6.2. Freebase Integration

We use Freebase in several ways:

- In the mobile app, when a user enters a search term, that search term is sent to Freebase and the results are displayed back to the user, allowing us to provide an autocomplete feature for artists and genres.
- Once a user clicks on a musical preference (an artist or genre), the mobile app queries our database (on Parse) and checks if the preference was already selected before. If it wasn't, the preference is inserted into the database, along with information from Freebase - the Freebase ID of the preference, and other preferences which are linked to it, such as related genres. The preference entry in our database is then associated with the requesting user.
- After the Mixus Player picks the next music preference to play, it sends a request to the YouTube API and uses the preference's Freebase ID as the search query.

## 6.3. Location Awareness and Battery Management

As mentioned several times in this report, Mixus is a proximity-based music platform. This means that, in order to influence the playlist of a Mixus music player, the user needs to be close to the player. As explained in 5.3. Wireless Networks Identification, the mobile app uses the BSSID of the wireless access point to identify whether it's in a Mixus-enabled venue or not. We'll now cover the process in depth.

The mobile app is equipped with an autonomous network scanner that was written specifically for this project. The scanner is run within a background service of the app, and is designed to load automatically and always run, even if the user didn't open the app. This allows the app to frequently scan for Wi-Fi networks and know if the user has entered a Mixus venue. Here's a scan cycle in detail (including some Android

terminology):

1. **Scan** - The scanner requests the OS to start a Wi-Fi scan. This call is asynchronous, so the scanner registers a *BroadcastReceiver* to intercept broadcast messages notifying that the scan has completed.
2. **Process scan results** - Once the scan completes, the app receives the full list of BSSIDs detected in the device's vicinity, and their signal strength. Duplicate results are eliminated, and the result list is sorted by strength.
3. **Send results to Parse** - The app performs a query of the WirelessNetwork model on Parse, requesting all objects whose BSSID is one of the found networks.
4. **Determine deployment** - Once the result set returns from Parse, the app iterates over that list and determines the detected access point whose signal was the strongest at the time of the passive Wi-Fi scan. This network is then selected, and its associated deployment is determined to be the venue in which the user is located.

This process allows both the application to retrieve information about the venue it's in, or if the device is not close to a Mixus-enabled venue, and also allows the server to register the user at the venue. Running the scan process regularly allows the application to quickly associate the user with a venue as he enters one, and disassociate the user when he's left the venue.

The scan process, while essential to the application's functionality, poses a major problem: Running Wi-Fi scans frequently on the user's device consumes battery power. A lot of battery power. And since our goal is to provide a good service without hurting battery life, we had to come up with a way to solve this problem.

Our solution: **Activity-based scan frequency change.** In a nutshell - use our

knowledge of what the user is physically doing to regulate the frequency of our passive Wi-Fi scans.

As mentioned earlier in 5.5. Google Play Services, we use a Google proprietary API called *Activity Recognition API*. This API allows us to know, with a high degree of certainty, what the user is "doing" - walking, running, driving... So, instead of running scans in a constant frequency (we started at every 3 minutes), the app can increase its scan frequency when there's a good chance that the user will enter a Mixus-enabled venue in the near future, and reduce the frequency when that probability is low.

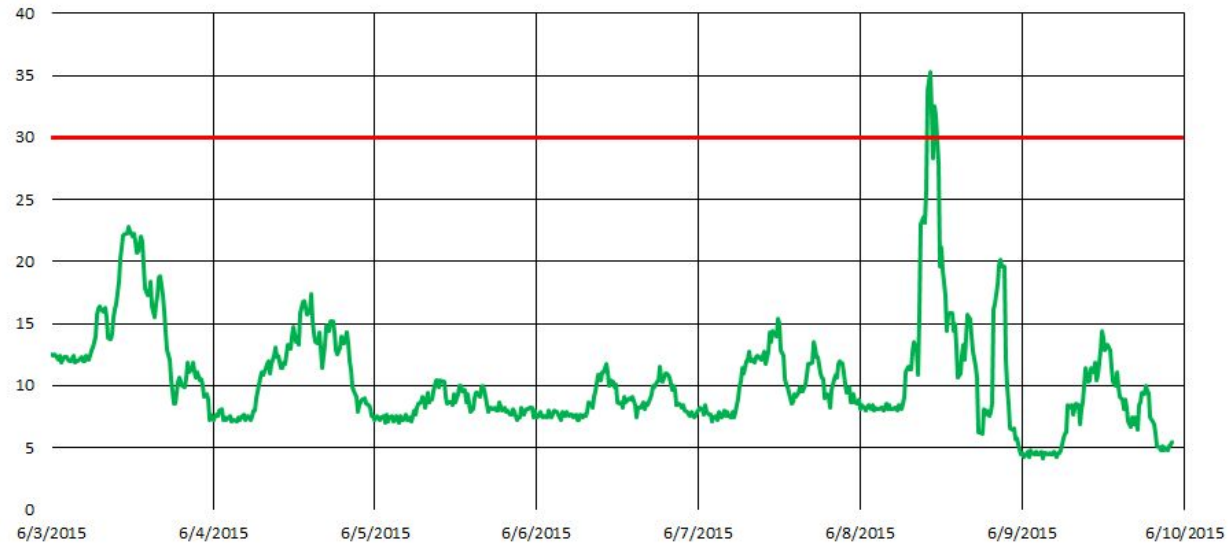
For the activity-based battery management algorithm we set 3 scan frequencies: **High** (every 2 mins), **Medium** (every 10 mins) and **Low** (every 30 minutes). We then built a decision tree, which determines if the scan frequency needs to be increased, decreased or left as is.

Some examples (the full decision tree can be found in the app's source code):

- If the user is **walking**, set the frequency to **High.**
- If the user is **bicycling**, **running** or **driving**, set the frequency to **Low.**
- If the user is **not moving**, and is in a Mixus venue, set the frequency to **Medium.** If he's not moving, and not in a Mixus venue, set the frequency to **Low.**

From the final presentation:

After implementing this algorithm, we experienced a subjective improvement in battery life. However, we wanted to quantify that improvement, so we installed the application on several devices and collected scan logs. We then used these logs to compare the number of scans that actually took place to those that would have taken place on the devices had we conducted scans in a constant frequency, and created a graph that shows the number of scans that have taken place in 10 minute increments:




(**Red** - constant frequency, every 3 minutes; **Green** - actual number of scans)

As we anticipated, the analysis proved that there was a significant improvement in battery life due to the reduced number of background Wi-Fi scans. More accurately, we found that, under normal use, we were able to attain a **65% reduction** in average battery consumption. This means that the Mixus app now barely enters Android's "Battery" screen, as it accounts for less than 1% of the total battery consumption.

## 6.4. Time Synchronization

As mentioned earlier, the Mixus Player updates the clients that are connected to it (via Parse push notifications) with information about the current playlist, including the current song's title, length, view count and the time offset since the beginning of the song. This is needed so that the mobile app can display the progress of the current song (the user can view the progress bar by sliding the drawer upwards. See screenshot in section 4.2. Mobile app).

Once we implemented this feature and tested it, we encountered a problem - Push notifications often do not reach the clients immediately, but instead take up to a minute. For most use cases this is OK, but it's not enough for updating clients with the song's progress, since even a few seconds of delay can cause the song's progress to be displayed incorrectly (a song is usually 3-4 minutes long).

Our solution was to have the player send the time when it started playing the current song (and not the offset since the beginning of the song). This works as long as the player's clock and the clients' clocks are synchronized, but normally there may be a gap of seconds or even minutes between the different clocks. In order to synchronize the clocks we used the NTP protocol (described in 5.4. NTP).

The final process is as follows:

- When the player or a client starts, one of the first things they do is get the time from one of 3 NTP servers we selected (IIX, NIST, NTP Project). The time difference between the player or client's local clock and the NTP server's clock is then saved, and will be referred to as the "delta".

- Every time the Mixus Player sends a message with the current playlist to its clients, the message contains the time when the currently playing song has begun playing, in the NTP server's time. This timestamp is calculated by taking the time by the server's local clock and adding the saved "delta" to it.

- Once a client receives the message, it first translates the timestamp to its local time by subtracting the "delta" from it. Note that the client and the server each have a different "delta" (that each of them calculated when they started).

A note about the NTP servers we use: We start by accessing the NTP servers of the Israeli Internet Exchange (IIX). If these servers are for some reason not accessible, we fall back to the NTP servers of the United States' National Institute of Standards and Technology (NIST). Finally, if none of the earlier servers could be reached, we fall back to NTP servers of the NTP Project.
An NTP server may not be accessible if it's down, or if there's geo-based filtering of requests (for example, only clients in Israel can query the IIX time server, and sometimes even legitimate devices in Israel are filtered out by mistake).

## 6.5. The Music Selection Algorithm

One of the Mixus Player's responsibilities is using the preferences of connected users to decide upon the next song to be fetched into the playlist. We designed this algorithm with our "all users have an equal say" principle in mind.

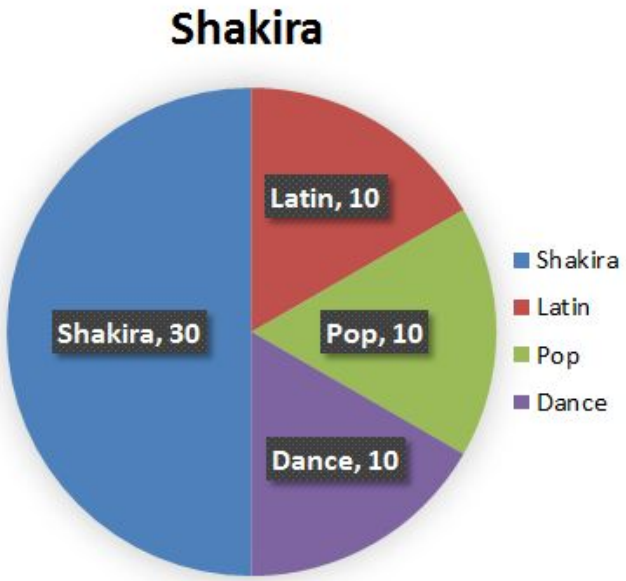
To understand the algorithm, we must first remember how a user's preference is represented in Mixus: It is a pair of **(preference information, strength)**.
The strength part is simply a number between 1 and 100 that represents how much a user "likes" that preference. Note that even though the player supports the strength parameter, the mobile app currently does not, so for now each preference is automatically awarded a strength of 100.

The preference information part contains fields like the preference's type (whether it is an artist or a genre) and name (for example, Shakira; for the full list of fields refer to the PreferenceInfo object in 6.1. Parse Integration), but the algorithm mainly uses 2 fields: The preference's Freebase ID, and the "linked preferences" fields, which contains a list of other preferences that are linked to our preference.

The idea behind linked preferences is to deduce more preferences that a user may like from a certain preference that we know for sure that the user likes. In the current implementation, the linked preferences are only genres. When we add a preference to our Parse database, we also fetch a list of related genres from Freebase and set the linked preferences to that list.

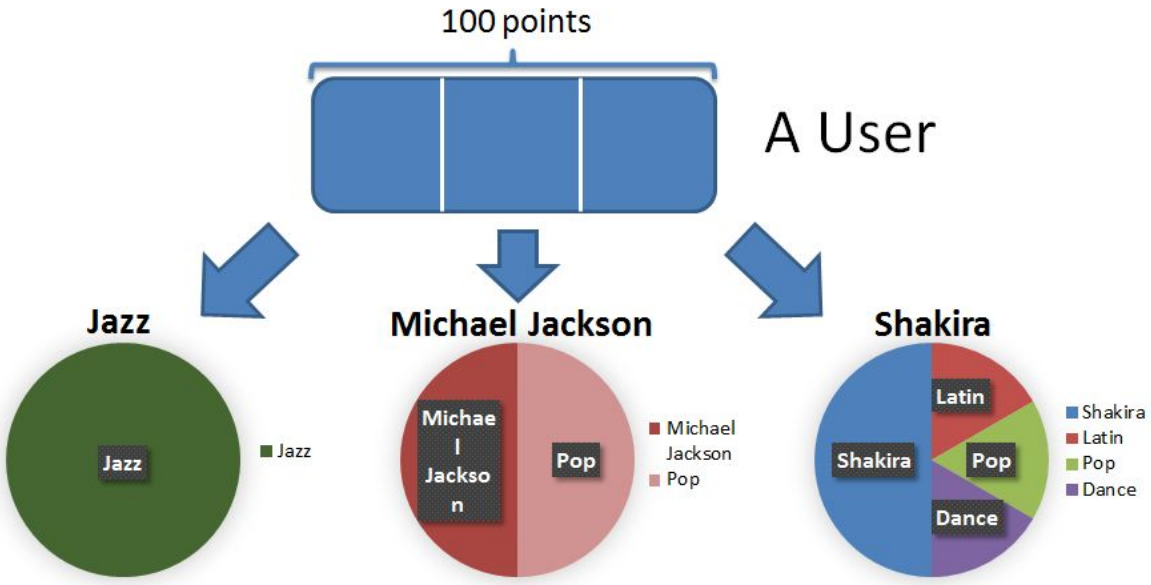
The algorithm distributes points to each such pair of (preference information, strength). The pairs' points are then split: Half of the points are awarded to the preference object itself, while the other half are split equally between all of its linked preferences. For

example, let's assume that a user has chosen to like the artist "Shakira", with a strength of 60. Shakira's linked preferences are the genres Latin, Pop and Dance. So the algorithm awards this specific pair of preferences 60 points, 30 of which are given to the "Shakira" preference, and 10 are given to each of the linked genres.



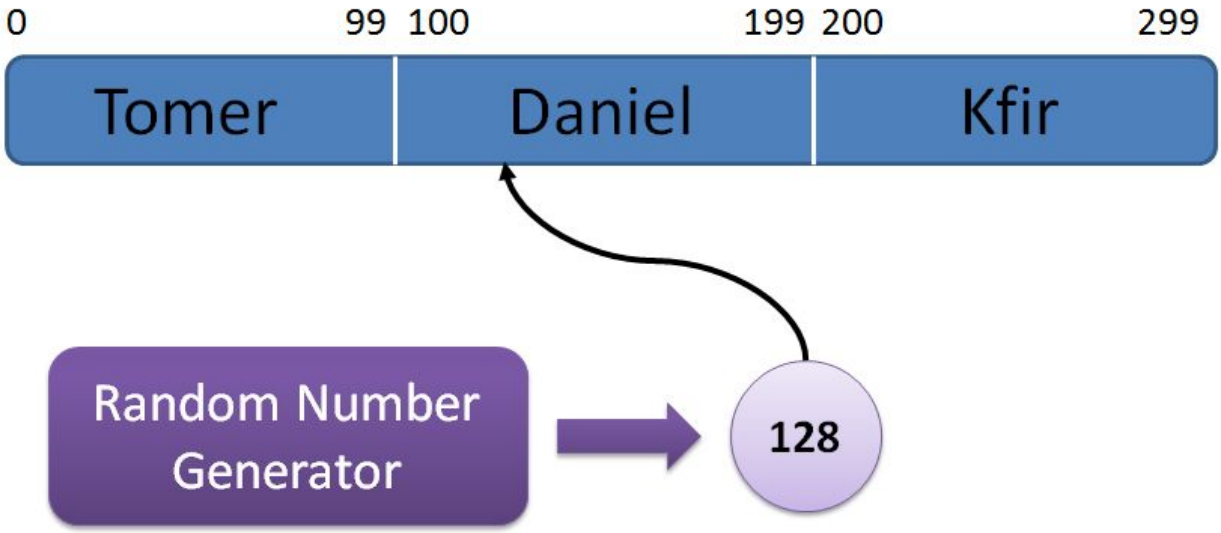

When the player goes over the user's preferences, it distributes points to each preference as explained above. Note that if it were to leave the point distribution in this manner, a state of inequality between users would form - those with more preferences would receive more points to their preferences than users with less preferences.

To correct this situation, we normalize the amount of points each user gets to 100. For example, if a user has 3 preference pairs: (Jazz, 100), (Michael Jackson, 80), (Shakira, 60), then the algorithm would first distribute 100 points to Jazz, 80 to Michael Jackson, and 60 to Shakira, then split each of the preferences points with their linked preferences, and finally normalize, so that each point that was supposed to be given to a preference or linked preference becomes $\frac{100}{100+80+60} = \frac{10}{24} = 0.41$ points.




One thing to note about the point distribution is that one preference's points can come from several users that like it. For example, Shakira may get X points from one user, and Y points from a second user, so it will have an overall of X+Y points.

Once the player finishes going over all of the users that are connected to the player's business venue, and over all of their preferences, it has a long list of preferences, and each preference is matched with its final number of points. Now, when the player needs to pick the a preference for the next song to fetch from YouTube, it generates a random (floating point) number between 0 and the sum of all distributed preference




points. The player keeps all the preferences in a certain order, so the generated number acts as a pointer to one of the preferences, and this preference is chosen for the next song to be fetched.

## 6.6. Smartwatch Integration


Users that install Mixus on their Android phone, who also own an Android smartwatch, are able to receive concise information about their current location and the song being played in the venue in real-time, just like they would receive in the mobile application itself. We used the *WearableExtender* class in the Android SDK to transfer this information to wearable devices. When relevant users enter a Mixus-enabled venue, they'll be able to display the following screens on their wearable device:

1. The name of the **current song**
2. The name of the **venue**
3. An action button allowing the user to **"leave" the place**. This means that despite the fact that the user's mobile phone is in range of a registered wireless AP, the system will ignore the user until he's entered the range of another AP
4. An action button allowing the user to **open the Mixus app** on his phone.

## 7. Quality Assurance

In order to properly develop and test the different parts of the Mixus system, we've taken several steps:

1. **Beta testing** - During the project, we conducted a 7-week long beta period, with the help of 8 beta users who installed the Mixus mobile app on their mobile phones. During this period, we released 17 different beta versions of the mobile application, some of which were designed to test new features on multiple devices, and some where urgent bugfixes for bugs that have been discovered by our beta users. One of the things we've learned during the project is that Android applications tend to behave differently on different devices, so it is **imperative to test the application on real devices** for a long period of time, and testing only on emulators is not sufficient.

2. **Automatic log collection** - To ensure our beta testing is optimal, we used an automatic log collection service called *Splunk*. By integrating their SDK to our mobile application, we were able to receive detailed information about both handled and unhandled exceptions that have taken place on users' devices. We were able to use this service to display both aggregated and segmented log data, and show logs for individual crashes and exceptions. This helped us a lot to learn about how our mobile application behaves on different devices, and correct our code accordingly.

3. **Real-device testing service** - In addition to running the mobile application on 8 real devices of beta users, we used an online service called *TestObject* which allows mobile developers to run and test their application on real devices over the Internet. The service offers over 100 different mobile phone models, so we were able to conduct further testing over multiple device types.