InstaMelody Application API

Table of Contents

1. Research 5

1.1 Database Schema 5

1.1.1 Schema Name 5

1.1.2 Schema Tables 5

1.1.3 Schema Diagram 7

1.1.4 Data Access 8

1.2 System Analysis 9

1.2.1 Scope Definition 9

1.2.2 Problem Analysis 9

1.2.3 Requirements Analysis 9

1.3 Application Structure 10

1.3.1 Application Components 10

1.3.2 Logical Design 11

1.3.3 API Functions 12

1.3.4 Application Logging 14

2. Planning 15

2.1 Goals 15

2.2 Ground Rules and Assumptions 15

2.3 Environment and Software Specification 16

2.4 Deliverables 16

2.5 Available Resources 16

3. Development 17

3.1 User API 18

3.1.1 CreateUser 18

3.1.2 GetUser 20

3.1.3 UpdateUser 22

3.1.4 UpdateUserProfileImage 22

3.1.5 DeleteUserProfile 23

3.1.6 RequestFriend 23

3.1.7 ApproveFriendRequest 24

3.1.8 DenyFriendRequest 24

3.1.9 DeleteFriend 25

3.1.10 GetUserFriends 25

3.1.11 GetPendingUserFriends 25

3.2 Authentication API 27

3.2.1 Authenticate (login) 27

3.2.2 ValidateSession 27

3.2.3 EndSession (logout) 28

3.2.4 UpdateUserPassword 28

3.2.5 ResetUserPassword 28

3.3 Category API 30

3.3.1 GetCategories 30

3.3.2 GetChildCategories 30

3.4 Messages API 31

3.4.1 CreateChat 31

3.4.2 GetChat 32

3.4.3 AddUserToChat 32

3.4.4 SendChatMessage 33

3.4.5 GetChatMessage 33

3.4.6 RemoveUserFromChat 33

3.5 Melodies API 35

3.5.1 GetMelodies 35

3.5.2 GetUserMelodies 36

3.5.3 AddUserMelody 36

3.5.4 DeleteUserMelody 37

3.5.5 GetLoop 38

3.5.6 AddUserLoop 38

3.5.7 AttachToUserLoop 40

3.5.8 DeleteUserLoop 40

3.6 Stations API 42

3.6.1 CreateStation 42

3.6.2 UpdateStation 43

3.6.3 DeleteStation 43

3.6.4 DeleteStationCategories 44

3.6.5 GetStations 45

3.6.6 GetAllStations 46

3.6.7 GetStationFollowers 46

3.6.8 FollowStation 46

3.6.9 UnfollowStation 47

3.6.10 CreateStationPost 47

3.6.11 CreateStationMessage 48

3.6.12 CreateStationPostReply 48

3.6.13 GetStationPosts 49

3.6.14 GetStationMessages 49

3.6.15 LikeStationPost 50

3.6.16 UnlikeStationPost 50

3.6.17 DeleteStationMessage 51

3.7 Uploads API 52

3.7.1 UploadFile 52

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Date** | **Name** | **Description** |
| 0.1 | 17 June 2015 | Jeff Dennis | Draft created |
| 0.2 | 19 June 2015 | Jeff Dennis | API Data Flows and Examples added |
| 0.3 | 6 July 2015 | Jeff Dennis | Updated 1.1.2 Schema Tables; Removed Graphic for 1.1.3 Schema Diagram; Updated sub-sections of 1.3.3 API Functions; Added many sub-sections to 3 Development |
| 0.4 | 28 July 2015 | Jeff Dennis | Updated 1.1.2 Schema Tables; Updated sub-sections of 1.3.3 API Functions; Added sub-sections to 3.5 Melodies API; Added section 3.6 Stations API; Moved Uploads API from section 3.6 to 3.7; Removed Example subsections for every API call in section 3 to separate documents; Uploaded new Graphic for 1.1.3 Schema Diagram |
| 0.5 | 31 July 2015 | Jeff Dennis | Fixed minor illustration error in Graphic within 1.1.3 Scheme Diagram; Added Info to Sections: 3.5.5 GetLoop, 3.5.6 AddUserLoop, 3.5.7 AttachToUserLoop, 3.5.8 DeleteUserLoop, 3.6.1 CreateStation, 3.6.2 UpdateStation, 3.6.3 DeleteStation, 3.6.4 DeleteStationCategories, 3.6.5 GetStations, 3.6.6 GetAllStations, 3.6.7 GetStationFollowers |
| 0.6 | 3 August 2015 | Jeff Dennis | Added Info to Sections: 3.6.8 FollowStation, 3.6.9 UnfollowStation, 3.6.10 CreateStationPost, 3.6.11 CreateStationMessage, 3.6.12 CreateStationPostReply, 3.6.13 GetStationPosts, 3.6.14 GetStationMessages, 3.6.15 LikeStationPost, 3.6.16 UnlikeStationPost, 3.6.17 DeleteStationMessage |
| 0.7 | 6 August 2015 | Jeff Dennis | Updated Section 3.4.2 GetChat |

# 1. Research

## 1.1 Database Schema

### 1.1.1 Schema Name

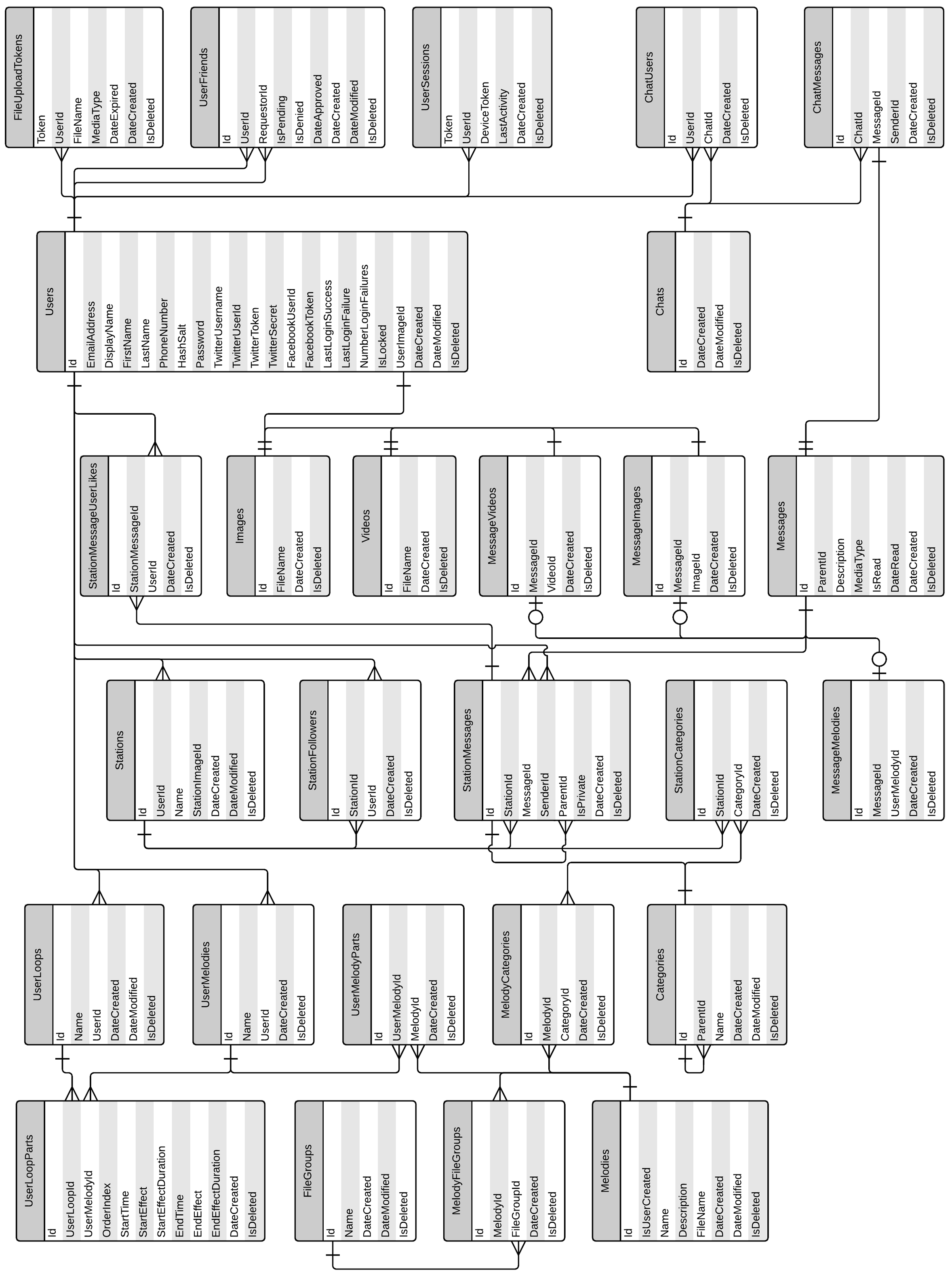
The schema name “dbo” will be used for database components that are part of this API. This schema will have interactions with other application schemas, as needed.

### 1.1.2 Schema Tables

The database will consist of the following tables:

|  |  |  |
| --- | --- | --- |
| **Table Name** | **Description** | **Primary Key** |
| Users | Holds data to identify and authenticate users. | UNIQUEIDENTIFIER |
| Melodies | Holds data to identify melodies. | Id; autoincrement |
| Categories | Holds data to identify categories. | Id; autoincrement |
| Messages | Holds data to identify messages. | UNIQUEIDENTIFIER |
| UserSessions | Holds data used to verify that a user is currently logged in. | UNIQUEIDENTIFIER |
| UserFriends | Holds data to maintain relationships between Users. | Id; autoincrement |
| Images | Holds data to identify user-uploaded images | Id; autoincrement |
| Videos | Holds data to identify user-uploaded videos | Id; autoincrement |
| FileGroups | Holds data to group Melody records into groups | Id; autoincrement |
| FileUploadTokens | Holds data used to validate and accept inbound file transfers to the server | UNIQUEIDENTIFIER |
| Chats | Holds data to establish a many-to-many relationship between Users and ChatMessages | UNIQUEIDENTIFIER |
| ChatMessages | Holds data to establish a many-to-many relationship between Messages and Chats | Id; autoincrement |
| ChatUsers | Holds data to establish a many-to-many relationship between Users and Chats | Id; autoincrement |
| MelodyFileGroups | Holds data to establish a many-to-many relationship between Melodies and FileGroups | Id; autoincrement |
| MelodyCategories | Holds data to establish a many-to-many relationship between Melodies and Categories | Id; autoincrement |
| UserMelodies | Holds data to establish a many-to-many relationship between Melodies and Users | UNIQUEIDENTIFIER |
| UserMelodyParts | Holds data to establish a many-to-many relationship between UserMelodies and Melodies | Id; autoincrement |
| UserLoops | Holds data to identify User-owned Loops | UNIQUEIDENTIFIER |
| UserLoopParts | Holds data to identify a many-to-many relationship between UserLoops and UserMelodies | Id; autoincrement |
| MessageImages | Holds data to associate Images to Messages | Id; autoincrement |
| MessageVideos | Holds data to associate Videos to Messages | Id; autoincrement |
| MessageMelodies | Holds data to associate UserMelodies to Messages | Id; autoincrement |
| Stations | Holds data to identify a User-owned Station | Id; autoincrement |
| StationCategories | Holds data to establish a many-to-many relationship between Stations and Categories | Id; autoincrement |
| StationFollowers | Holds data to establish a many-to-many relationship between Stations and Users | Id; autoincrement |
| StationMessages | Holds data to establish a many-to-many relationship between Stations and Messages | Id; autoincrement |
| StationMessageUserLikes | Holds data to identify Users who have liked posted StationMessages | Id; autoincrement |

### 1.1.3 Schema Diagram



### 1.1.4 Data Access

For initial development and testing purposes, all database access will be executed through MS-SQL using ADO.NET. Future development may include converting some database access calls to TSQL Stored Procedures (SProcs). Testing should identify which portions might benefit from this conversion.

## 1.2 System Analysis

### 1.2.1 Scope Definition

The API will provide a centralized point of access for mobile and web applications on the InstaMelody platform.

### 1.2.2 Problem Analysis

The goal is to have a single interface that can provide a way to store and retrieve data from the InstaMelody mobile and web applications.

### 1.2.3 Requirements Analysis

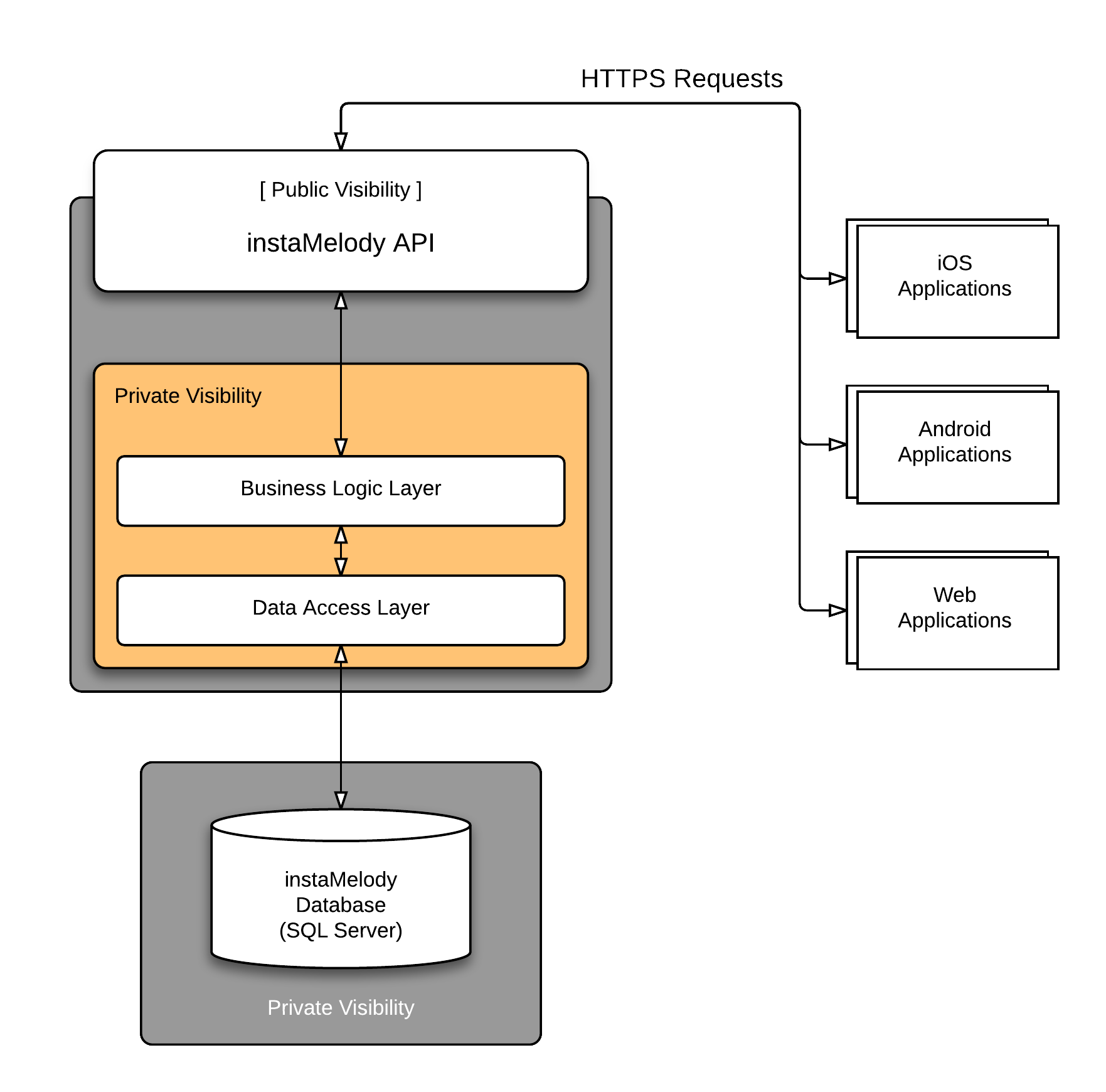
* Functional design documentation
* Technical design documentation
* Establish standard authentication models and behaviors
* Allow users to create new user accounts
* Allow users to create melodies and send them to other users in the form of messages
* Allow users to create stations and post melodies to these stations
* Allow users to follow stations that have been created by other users
* Error message should be displayed in a user-friendly context
* The API will handle all interaction with the database

## 1.3 Application Structure

### 1.3.1 Application Components

* InstaMelody Database (SQL Server; as described in section 1.1)
* Server capable of hosting .NET API applications
* Server security certificate (for HTTPS communication)
* API Service (visible API, BLL, DAL)

### 1.3.2 Logical Design

****

### 1.3.3 API Functions

#### 1.3.3.1 User API Functions

* CreateUser
* GetUser
* UpdateUser
* UpdateUserProfileImage
* DeleteUserProfile
* RequestFriend
* ApproveFriendRequest
* DenyFriendRequest
* DeleteFriend
* GetUserFriends
* GetPendingUserFriends

#### 1.3.3.2 Authentication API Functions

* Authenticate (login)
* ValidateSession
* EndSession (logout)
* UpdateUserPassword
* ResetUserPassword

#### 1.3.3.3 Category API Functions

* GetCategories
* GetChildCategories

#### 1.3.3.4 Messages API Functions

* CreateChat
* GetChat
* AddUserToChat
* SendChatMessage
* GetChatMessage
* RemoveUserFromChat

#### 1.3.3.5 Melodies API Functions

* GetMelodies
* GetUserMelodies
* AddUserMelody
* DeleteUserMelody
* GetLoop
* AddUserLoop
* AttachToUserLoop
* DeleteUserLoop

#### 1.3.3.6 Stations API Functions

* CreateStation
* UpdateStation
* DeleteStation
* DeleteStationCategories
* GetStation
* GetAllStations
* GetStationFollowers
* FollowStation
* UnfollowStation
* CreateStationPost
* CreateStationMessage
* CreateStationPostReply
* GetStationPosts
* GetStationMessages
* LikeStationPost
* UnlikeStationPost
* DeleteStationMessage

#### 1.3.3.7 Uploads API Functions

* UploadFile

### 1.3.4 Application Logging

The API application will log information using the static InstaMelodyLogger object exposed by the InstaMelody.Infrastructure project. Since this is a static object that is external to the specific structure of the project, it will be available to all component projects that have a reference to the Infrastructure project.

#### 1.3.4.1 External Application Logging

Logging functionality will not be exposed at the API level. The included logging capabilities are only for use by the API itself. It is assumed that any client applications will handle their own logging needs.

#### 1.3.4.2 Logging Levels

The NLog library currently supports seven “levels” of logging messages:

|  |  |
| --- | --- |
| Off | Turns logging activity off |
| Fatal | Severe errors that cause unplanned application termination. |
| Error | Runtime errors or unexpected conditions that do not cause termination. |
| Warn | Runtime situations that are undesirable or unexpected, but not necessarily “errors.” |
| Info | Important runtime events (startup/shutdown), or other information useful for application monitoring. |
| Debug | Detailed information on the flow through the system. Generally disabled for production environments. |
| Trace | The most detailed information. Primarily useful in development or testing environments. |

## 2. Planning

## 2.1 Goals

* Provide user authentication functionality
* Persist data when submitted by the client application
* Retrieve data when requested by the client application

## 2.2 Ground Rules and Assumptions

* All business logic will be included in the BLL.
* All data access logic will be included in the DAL, and use ADO.NET.
* All public functionality will be available only through the API.
* API calls will be HTTP Get or Post requests.
* Data for API calls will be either query string, or JSON objects, as appropriate.
* Password rules will be read from a config file that is read on application start.
* User data will be stored in a database.
* All active users will have sufficient access to manage their own profile.
* Data should not be deleted by automated processes of the API, but marked as “deleted.”
* The API will have access to a mail server.
* The API will have an email account with rights to send emails.
* The API will use a “Salted Hash” algorithm for secure password storage.
* The API will use a “cryptographic hash function” (e.g. SHA256, SHA512).

## 2.3 Environment and Software Specification

* The environment consists of a Windows 2012 Server with an instance of Microsoft SQL Server 2012 running.
* The server must be publicly visible, to allow authentication and other functions on mobile devices.
* Only the API public interface will be exposed for public access.
* All database access will be executed within the API process.
* All business rules validation will be executed within the API process.
* Accessing the API will be accomplished through a domain: (TBD).

## 2.4 Deliverables

* Database with initial data.
* API to receive HTTP requests, process as required, return response.

For the definition of deliverables, the API will include all individual layers of the middle tier executable projects: API, BLL, and DAL.

* Unit tests as required for various project modules.

## 2.5 Available Resources

|  |  |
| --- | --- |
| **Name** | **Role** |
| Matt Pate | CEO, Founder, Project Manager, App Design |
| Ahmed Bakir | Front-end/mobile programming, testing |
| Jeff Dennis | API Programming, testing |

# 3. Development

The foundation of the system is a SQL Server database that will store all relevant data for users.

The Data Access Layer (DAL) will reside just above the actual database. It will handle all of the details involved with direct database communication. The DAL will consist of three tiers, as illustrated above, and will be implemented to leverage ADO.NET to access the database.

The Business Logic Layer (BLL) will be on top of the DAL. This portion will contain all data validation and error handling for this API. This separation allows the BLL to portion to specialize in verifying the validity of the object models of data, without any regard for the data source or destination.

The topmost layer is the Application Programming Interface (API). This layer will be visible and accessible to other applications. The functions involved at this layer will be to interact with other applications, using HTTP Requests, and transforming internal object models to and from JSON (JavaScript Object Notation) objects.

All of these modules will be written in C#, and will be organized into the following projects:

InstaMelody.Infrastructure: Various custom classes, enumerations, exceptions and constants that are intended to be used across multiple applications.

InstaMelody.Model: Custom classes for use across the various components within the InstaMelody API project.

InstaMelody.API: The group of components needed to present a public interface for use by other InstaMelody applications.

InstaMelody.Business: The modules that perform the details of validating the actual data objects (as declared in InstaMelody.Model), and handling any errors that might occur for incoming requests or database access.

InstaMelody.Data: All of the code modules that either interact directly with the database, or support this access.

Any client applications will access the functionality of this API through the API function calls, as described in section 1.3.3. All of these calls are charted in detail in the next section.

## 3.1 User API

The function calls exposed by the User API will handle the needs of customers accessing the InstaMelody API. With the exception of the CreateUser function (see Section 3.1.1), all user API calls must be accompanied by a valid Session token that corresponds to the requesting user in order for any function calls to be processed.

Any errors are handled in the BLL, with error messages being translated to more friendly and informative messages, based on the use context of the call.

### 3.1.1 CreateUser

The CreateUser API function can be called without having a session token. The API will receive the user data in the body of an HTTP PUT or POST request, and use that data to create a *User* object.

This new *User* object will be sent to the BLL where it will be validated. The BLL will check the *User* object’s email address and display name values and make sure that these values are unique by confirming that no records matching the provided email address and display name exist in the database. If a matching email address or display name exists, an error will be returned back to the client and no further action will take place. Once the object has been validated a random password salt will be generated, and the salted password will be hashed and stored in the *User* object.

Once the validation and password hashing has completed the data will be sent to the database via the DAL using ADO.NET. After the data has been successfully saved in the database, a *User* object will be returned to the BLL, and then to the API, and finally be returned to the client in the form of an HTTP response message containing the stored user details. For security reasons, the password and password salt will not be returned to the client.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.



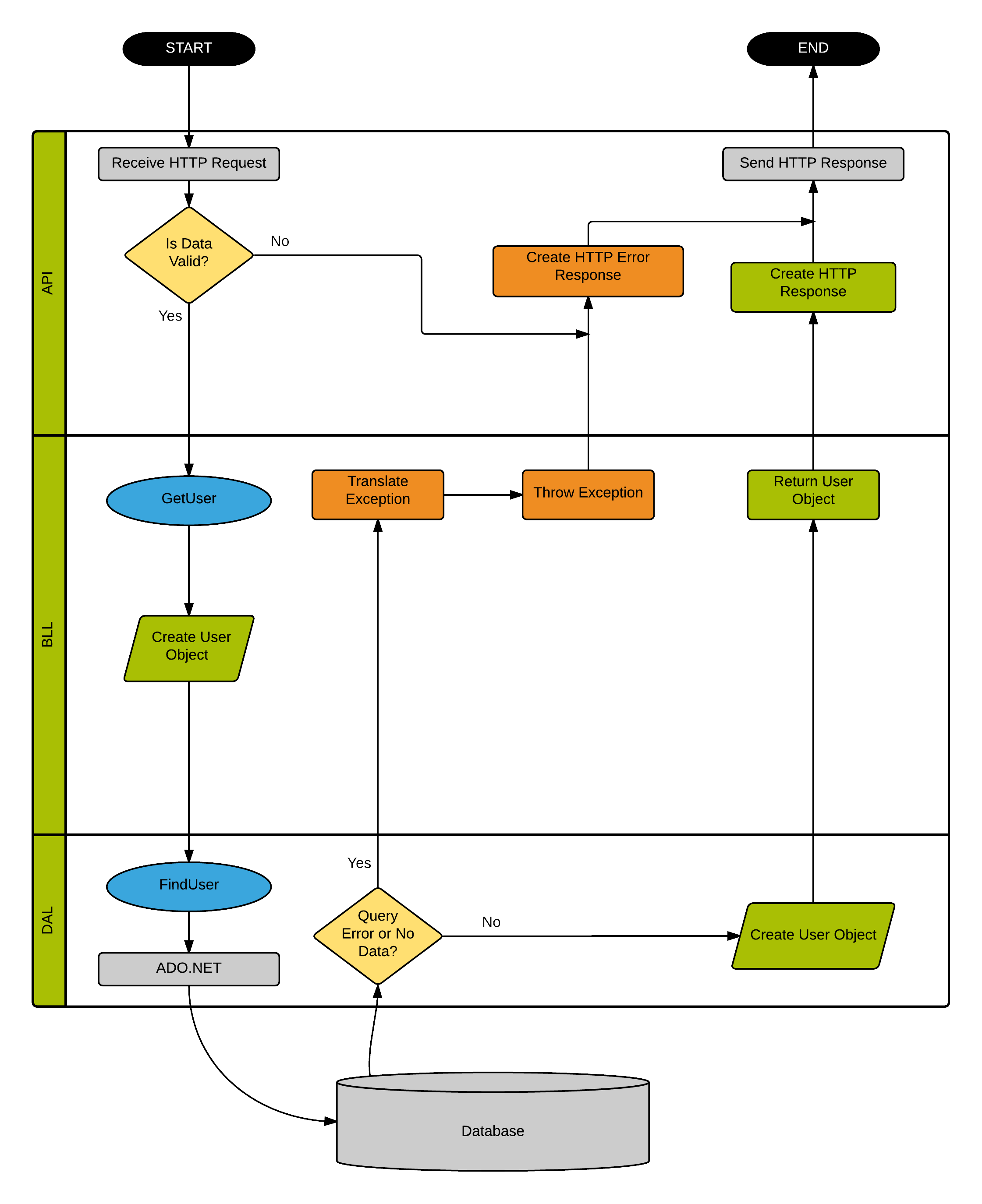
### 3.1.2 GetUser

The GetUser API function will be called in the form of an HTTP GET request, and will require a *token* URL parameter. If the *token* URL parameter is not provided, an HTTP 401 (Unauthorized) response will be returned to the client. To identify the user, an *id, emailAddress,* or *displayName* URL parameter will need to be included with the *token* parameter in the URL.

The provided information will then be passed to the BLL, which will send a call to the DAL to lookup the user.

Once the data has been sent to the DAL, it will be queried against the database via the DAL. If a result is found, a *User* object will be returned to the BLL, and then to the API, and finally be returned to the client in the form of an HTTP response message containing the stored user details. For security reasons, the password and password salt will not be returned to the client.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.



### 3.1.3 UpdateUser

The UpdateUser API function will receive the user data in the body of an HTTP POST request. This API will not let a client update the *Id*, *UserImage*, *HashSalt*, and *Password* values. Along with the User fields that will be updated a valid session *Token* must be provided to authenticate the User. The API layer will serialize the request into a new *User* object and send that object to the BLL.

This new *User* object will be sent to the BLL where it will be validated. The BLL will check the *User* object’s email address and display name values and make sure that these values are unique by confirming that no records matching the provided email address and display name exist in the database. If a matching email address or display name exists, an error will be returned back to the client and no further action will take place.

Once the validation has completed the data will be sent to the database via the DAL using ADO.NET. After the data has been successfully saved in the database, a *User* object will be returned to the BLL, and then to the API, and finally be returned to the client in the form of an HTTP response message containing the stored user details. For security reasons, the password and password salt will not be returned to the client.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

### 3.1.4 UpdateUserProfileImage

The UpdateUserProfileImage API function will receive the user data in the body of an HTTP POST request. This API will let a client update the *UserImage* only. Along with the User fields that will be updated a valid session *Token* must be provided to authenticate the User. The API layer will serialize the request into a new *User* object and send that object to the BLL.

This new *User* object will be sent to the BLL where it will be validated. The BLL will the User’s existing profile Image and delete it if an Image is found. The BLL will then check the image *FileName* to make sure it is unique. If a *FileName* is not provided, or if an image already exists with the same *FileName*, an error will be returned back to the client and no further action will take place.

Once the validation has completed the data will be sent to the database via the DAL using ADO.NET. After the data has been successfully saved in the database, a *User* object and a *FileUploadToken* object will be returned to the BLL, and then to the API, and finally be returned to the client in the form of an HTTP response message containing the stored user details. For security reasons, the password and password salt of the *User* will not be returned to the client.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

Once the UpdateUserProfileImage API call has been made, the client must make the UploadFile API call (see section 3.7.1 UploadFile API) within 10 minutes to upload the image to the database.

ADD DATA FLOW DIAGRAM HERE

### 3.1.5 DeleteUserProfile

The DeleteUserProfile API function will receive the user data in the body of an HTTP POST request. To process this request, the API must be passed a valid session *Token* belonging to the User, along with the User’s *Id*, *DisplayName*, or *EmailAddress* to validate the User. The API layer will serialize the request into a new *User* object and send that object to the BLL.

This new *User* object will be sent to the BLL where it will be validated. The BLL will validate the session *Token* against the provided User data. If the *Token* doesn’t match the User found with the provided User data, an error will be returned back to the client and no further action will take place.

Once the validation has completed the data will be sent to the database via the DAL using ADO.NET. This API will not delete the User from the database, but rather set the *IsDeleted* flag on the User object in the database, which will prevent the DAL from retrieving this data via any other API calls. Once the User object has been deleted, a confirmation will be passed up to the BLL, then to the API, which will trigger the API to respond with an HTTP 202 (Accepted) response.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

### 3.1.6 RequestFriend

The RequestFriend API function will receive the user data in the body of an HTTP POST request. To process this request, the API must be passed a valid session *Token* belonging to the User, along with another User’s *Id*, *DisplayName*, or *EmailAddress* to validate the friend being requested. The API layer will serialize the request into a new *User* object and send that object to the BLL.

This new *User* object will be sent to the BLL where it will be validated. The BLL will get the requesting user using the provided *Token*, and lookup the User to be sent a friend request using the provided *User* object. If the session cannot be validated using the *Token*, or if a User cannot be found with the provided data, an error will be returned back to the client and no further action will take place.

Once the validation has completed the data will be sent to the database via the DAL using ADO.NET. Once the data has been saved in the database, a response will be sent back to the BLL, which will trigger an Apple push notification to be sent to the friend that is being requested. After the push notification is sent, the requested friend’s display name will be sent from the BLL to the API layer, which will be used to send a message back to the client.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

### 3.1.7 ApproveFriendRequest

The ApproveFriendRequest API function will receive the user data in the body of an HTTP POST request. To process this request, the API must be passed a valid session *Token* belonging to the User, along with the requesting User’s *Id*, *DisplayName*, or *EmailAddress* to validate the friend request. The API layer will serialize the request into a new *User* object and send that object to the BLL.

This new *User* object will be sent to the BLL where it will be validated. The BLL will find the user using the provided *Token*, and lookup the requesting User to be approved using the provided *User* object. If the session cannot be validated using the *Token*, or if a pending friend request cannot be found with the provided data, an error will be returned back to the client and no further action will take place.

Once the validation has completed the data will be sent to the database via the DAL using ADO.NET. Once the data has been saved in the database, a response will be sent back to the BLL, and finally to the API layer, which will be used to send a message back to the client informing the client of the approved friend request.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

### 3.1.8 DenyFriendRequest

The DenyFriendRequest API function will receive the user data in the body of an HTTP POST request. To process this request, the API must be passed a valid session *Token* belonging to the User, along with the requesting User’s *Id*, *DisplayName*, or *EmailAddress* to validate the friend request. The API layer will serialize the request into a new *User* object and send that object to the BLL.

This new *User* object will be sent to the BLL where it will be validated. The BLL will find the user using the provided *Token*, and lookup the requesting User to be approved using the provided *User* object. If the session cannot be validated using the *Token*, or if a pending friend request cannot be found with the provided data, an error will be returned back to the client and no further action will take place.

Once the validation has completed the data will be sent to the database via the DAL using ADO.NET. Once the data has been saved in the database, a response will be sent back to the BLL, and finally to the API layer, which will be used to send a message back to the client informing the client of the denied friend request.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

### 3.1.9 DeleteFriend

The DeleteFriend API function operates exactly the same as 3.1.7 ApproveFriendRequest and 3.1.8 DenyFriendRequest, except that a message will be returned to the client to notify the client that the requested friend has been deleted.

As with any type of API delete request, the data is never actually deleted from the database, but rather the *IsDeleted* flag will be set in the database, which will prevent the DAL from retrieving this data from being used by any other API calls.

ADD DATA FLOW DIAGRAM HERE

### 3.1.10 GetUserFriends

The GetUserFriends API function will be called in the form of an HTTP GET request, and will require a *Token* URL parameter. If the *token* URL parameter is not provided, an HTTP 401 (Unauthorized) response will be returned to the client.

The provided information will then be passed to the BLL, which will send a call to the DAL to lookup all existing friends (who are not pending or denied) for the user.

Once the data has been sent to the DAL, it will be queried against the database via the DAL. If a result set is found, an array of *User* objects will be returned to the BLL, and then to the API, and finally be returned to the client in the form of an HTTP response message containing the stored user details for each friend. For security reasons, the password, password salt, twitter token, twitter secret, and facebook token fields will not be returned to the client for any requested friend data.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

### 3.1.11 GetPendingUserFriends

The GetPendingUserFriends API call functions exactly the same as 3.1.10 GetUserFriends, except that an array of pending friend requests will be returned to the client.

ADD DATA FLOW DIAGRAM HERE

## 3.2 Authentication API

The function calls exposed by the Authentication API will handle all authentication requests.

In all calls to this API, the user name data is assumed to be a valid email address or a valid display name. Any errors are handled in the BLL, with error messages being translated to more friendly and informative messages, based on the use context of the call.

### 3.2.1 Authenticate (login)

The Authenticate API process will be one of the more complex processes in this API. The process starts with an HTTP POST request that will hold a *DisplayName* or *EmailAddress, Password,* and *Device Token*. The *Device Token* can only be provided during the Authenticate API and will be used to send push notifications out to mobile devices during certain events. Use of a security certificate to support HTTPS communication will be required for this part to be secure. Without using HTTPS at this point will result in the user authentication data being transmitted as clear text. This would constitute a significant security weakness.

When the API receives an authentication request it will simply create a *User* object to hold the *DisplayName* (or *EmailAddress), Password,* and *DeviceToken* data. Then the Authenticate function of the BLL will be called with this data.

The BLL will first perform basic validation using regular expressions string evaluation tools, as needed. Invalid data will result in an exception that will immediately return to the API. Valid data will be followed by a call to the DAL to get full user information for the user by supplying the user email address or display name. Then the input password will be “hashed” after appending the “salt value” from the user data that was returned from the DAL. At this point the hashed input password will be compared to the stored hashed password. Success or failure of this comparison determines overall success or failure of authentication. The user data will be updated to reflect success or failure, and then sent to the DAL for saving.

Then the results will be returned to the API: a valid session Token in the case of a success, or an exception in the case of a failure.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

### 3.2.2 ValidateSession

The ValidateSession call is intended to assist application-specific APIs, by giving them the current “logged in” status of a user. The information will be sent to the API as URL parameters in the form of an HTTP GET request. The required inputs are the *token* and either the *id* or *deviceToken* parameters. This data will be passed down through the BLL, where it will be validated for simple formatting. Valid data will be forwarded to the DAL where it will become the input criteria to find a matching record in the database UserSessions table.

If a matching record is found, then the call will return a valid Token. If no matching record is found, then it will return an error message. Any error conditions will be caught at the BLL, where they will be evaluated and then returned to the calling process.

ADD DATA FLOW DIAGRAM HERE

### 3.2.3 EndSession (logout)

This API function will simply close the current user session. The EndSession function is called via an HTTP POST request, with the body of the request containing the *Token* and either the *UserId* or *DeviceToken* properties. If the properties are valid, the API will respond with an HTTP 200 message informing the client that the user has been logged out.

The end result is that the next time that user generates any call to the API, there will be no valid session found, and the user will be required to log in again before any API activity can be executed.

ADD DATA FLOW DIAGRAM HERE

### 3.2.4 UpdateUserPassword

The UpdateUserPassword API function is used to change a User’s password. This function is called via an HTTP POST request, with the body of the request containing the *Token* and a *UserPassword* object, which contains the *OldPassword* value and the *NewPassword* value. The API layer will serialize the request into a new object and send that object to the BLL.

The BLL will find the user using the provided *Token*, and lookup the requesting User’s password that is stored in the database. The BLL will then hash the *OldPassword* compare the hashed *OldPassword* with the encrypted password in the database. If these two values are not equal an error will be returned to the client. If the two values are equal, the *NewPassword* value will be hashed and sent to the DAL to be saved to the User’s record in the database.

Once the DAL has saved the new password the User’s active UserSessions will be inactivated, and a new session *Token* will be created and sent back to the API, where it will be returned to the client.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

### 3.2.5 ResetUserPassword

The ResetUserPassword API function is used to reset a User’s password who cannot successfully Authenticate, or whose account has been locked due to consecutive failed Authenticate attempts. This function is called via an HTTP POST request, with the body of the request containing either a *DisplayName* or *EmailAddress* value.

The BLL will find the user using the provided *DisplayName* or *EmailAddress*, and generate a temporary password for the User, which will be sent via email to the email address for that User. If the User’s account is locked, it will be unlocked at this point, and all active sessions for this user will be inactivated.

Once the email has been sent, the BLL will pass the following message up to the API, which will return that message to the client: *“An email has been sent to {emailAddress} with a temporary password.”*

ADD DATA FLOW DIAGRAM HERE

## 3.3 Category API

The function calls exposed by the Category API are used to retrieve pre-defined categories that will be used to categorize Melodies and Stations.

In all calls to this API, there is no authentication token required to retrieve data on Categories.

All errors are handled in the BLL, with error messages being translated to more friendly and informative messages, based on the use context of the call.

### 3.3.1 GetCategories

The GetCategories API function is used to retrieve a list of Categories. This API is called using an HTTP GET request with no parameters.

The BLL will find a list of all Categories that are not set as deleted. If there are no Categories found in the database the BLL will return an error message to the API, which will return that error message to the client. If any Categories are found, they will be sent from the BLL to the client via the API HTTP 200 response.

ADD DATA FLOW DIAGRAM HERE

### 3.3.2 GetChildCategories

The GetChildCategories API function is used get a list of Child categories for a given category. This API uses an HTTP GET request with an *id* parameter of the requested category to find any child categories.

The BLL will use the *id* parameter to find any Categories with a ParentId value that matches the given client *id* parameter input. If there are no Categories found in the database the BLL will return an error message to the API, which will return that error message to the client. If any Categories are found, they will be sent from the BLL to the client via the API HTTP 200 response.

ADD DATA FLOW DIAGRAM HERE

## 3.4 Messages API

The function calls exposed by the Messages API are used to send Messages to other Users in the form of Chats.

Any *Message* object that is sent to this API must contain a *Description*, *Image*, *Video*, or *UserMelody* value.

* **Description** – Text that will be sent to the user
* **Image** – an *Image* object containing the *FileName* of the image. This *FileName* will be added to the database as metadata, and a *FileUploadToken* object will be sent back to the client to allow the client to upload the file to the server using the UploadFile API call (see section 3.7.1 UploadFile API).
* **Video** – a *Video* object containing the *FileName* of the image. This *FileName* will be added to the database as metadata, and a *FileUploadToken* object will be sent back to the client to allow the client to upload the file to the server using the UploadFile API call.
* **UserMelody** – a *UserMelody* object containing the *Id* of an existing *UserMelody*, or the necessary data required to create a new *UserMelody* (see section 3.5.3 AddUserMelody API).

All functions exposed by this API will require a valid session token to authenticate the User session.

All errors are handled in the BLL, with error messages being translated to more friendly and informative messages, based on the use context of the call.

### 3.4.1 CreateChat

The CreateChat API function will receive *User* data, *Message* data, and a valid *Token* in the body of an HTTP POST request. To process this request, the API must be passed a valid session *Token.* This API can accept a single User object, containing the *Id*, *DisplayName*, or *EmailAddress* fields, or an array of User objects containing any of the mentioned fields. The BLL will use this data to lookup the user(s) and add them to the chat. Finally, the API requires a *Message* object with valid data to start a Chat (see section 3.4 Messages API for more required *Message* object data).

This data will be sent to the BLL where it will be validated. The BLL will get the requesting user using the provided *Token*, and lookup all requested User(s) with the provided API data. Currently, only InstaMelody Users who are Friends can start Chats with each other or add Friends to existing Chats. This rule applies to multiple Users as well (e.g.: User A and User B are friends. User B and User C are friends. User A and User C are not friends. User B can add User C to a Chat that has been created between User A and User B, but User A cannot add User C to the chat).

After all users have been validated each user will be added to a new Chat Session, then a new Chat Message will be created using the Message data sent to the API and added to the Chat Session. Once this Chat Message has been created a push notification will be sent to each User in the Chat Session, provided this user has a valid Session in the database that contains a *DeviceToken*.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

### 3.4.2 GetChat

The GetChat API function will be called in the form of an HTTP GET request, and will require the *token,* and optionally *id* URL parameters. If the token URL parameter is not provided, an HTTP 401 (Unauthorized) response will be returned to the client.

The provided information will then be passed to the BLL, where the BLL will first try to search for a chat by Id. The BLL will send a call to the DAL to find the Chat. If a Chat with the provided Id is not found an error will be sent from the BLL to the client via an API HTTP 400 response message. If the *id* URL parameter is not provided, the BLL will get all chats for a User by the provided *token* value.

If the Chat is found, the BLL will request all chat Messages and Users via separate calls to the DAL. Once the DAL has passed back this information, the BLL will format all of the data into the Chat object, and return this object to the API. The API will pass back an HTTP 200 message to the client containing the Chat data.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

### 3.4.3 AddUserToChat

The AddUserToChat API function will be called using an HTTP POST request. This API will accept a *Token*, *Chat* object containing the Chat *Id* field, and *User* object containing the *Id*, *DisplayName*, or *EmailAddress* fields. The API layer will serialize the request and send the data to the BLL.

The BLL will receive the provided data from the API and check the *Token* to verify that the request is valid. Once the *Token* is validated, the BLL will send the requested User’s data to the DAL and verify that the requested User and the requesting User are friends. Finally, the BLL send the Chat *Id* to the DAL to retrieve the Chat data. If all criteria passes, the requested User will be added to the Chat and a push notification will be sent to all Users in the Chat that have a Session with a *DeviceToken*, informing each user in the chat of the new user.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

### 3.4.4 SendChatMessage

The SendChatMessage API function will receive *Chat* data, *Message* data, and a valid *Token* in the body of an HTTP POST request. To process this request, the API must be passed a valid session *Token.* The API also requires a *Message* object with valid data to start a Chat (see section 3.4 Messages API for more required *Message* object data).

This data will be sent to the BLL where it will be validated. The BLL will get the requesting User data using the provided *Token*, and will send a request to the DAL to retrieve the Chat data. After the Chat data has been found and the requesting User has been validated as a member of the requested Chat, a new Chat Message will be created using the Message data sent to the API and added to the Chat Session. Once this Chat Message has been created a push notification will be sent to each User in the Chat Session, provided this user has a valid Session in the database that contains a *DeviceToken*.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

### 3.4.5 GetChatMessage

The GetChatMessage API function will be called in the form of an HTTP GET request, and will require the *token, chatId,* and *messageId* URL parameters. If any of these URL parameters are not provided, an HTTP 401 (Unauthorized) response will be returned to the client.

The provided information will then be passed to the BLL, which will send a call to the DAL to find the Chat. If a Chat with the provided Id is not found an error will be sent from the BLL to the client via an API HTTP 400 response message.

If the Chat is found, the BLL will find the requested Chat Message via a call to the DAL. Once the DAL has passed back this information, the BLL will format all of the data into a ChatMessage object, and return this object to the API. The API will pass back an HTTP 200 message to the client containing the ChatMessage data.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

### 3.4.6 RemoveUserFromChat

The RemoveUserFromChat API will allow a User to remove themself from a Chat. This function will be called in the form of an HTTP POST request, and will require the *Token* and a *Chat* object containing the *Id* of the chat. The API layer will serialize the request and send the data to the BLL.

This data will be sent to the BLL where it will be validated. The BLL will get the requesting User data using the provided *Token*, and will send a request to the DAL to retrieve the Chat data. Once the User has been validated as a member of the found Chat, a request will be sent to the DAL to remove the requesting User as a ChatUser. Once the DAL has completed this request, a push notification will be sent to all remaining users in the Chat, provided that each user has a valid Session with a *DeviceToken*.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

## 3.5 Melodies API

The function calls exposed by the Melodies API serve multiple purposes.

One purpose of this API is to retrieve pre-defined Melody files (referred to in this document as Base Melodies). These Base Melodies are pre-loaded in the database and are comprised of many audio files that can be “stacked” on top of one another to make up an audio melody or beat.

A second purpose of this API is to save and retrieve user-created audio content, built atop of the Base Melody audio files. These user-created audio files, when played back over the Base Melody sound files, create a User Melody that can be messaged to other users (see section 3.4 Messages API for more), or can be turned into a “Melody Loop”, which consists of several User Melodies chained together.

All functions exposed by this API will require a valid session token to authenticate the User session.

All errors are handled in the BLL, with error messages being translated to more friendly and informative messages, based on the use context of the call.

### 3.5.1 GetMelodies

The GetMelodies API is a flexible API that allows the client to get an array of Melody files using a few different approaches. This API function will be called in the form of an HTTP GET request, and will require the *token* URL parameter. If the *token* URL parameters are not provided, an HTTP 401 (Unauthorized) response will be returned to the client. The optional URL parameters are:

* ***groupId*** or ***groupName*** – allows the client to request an array of Melodies based on their Group.
* ***categoryId***  - allows the client to request an array of Melodies based on the Category they belong to.
* ***id*** or ***fileName*** – allows the client to request a specific Melody based on the Id or Filename of the Melody.
* ***{no other parameters}*** – the API will retrieve a list of all Melody files, sorted by Group.

The API will first attempt to search for the *groupId* or *groupName* URL parameters. If either or both of these URL parameters are found, a request will be sent to the BLL to attempt to get all Melodies with the matching values. If the *groupId* and *groupName* URL parameters are not provided, the API will then attempt to find the *categoryId* URL parameter. If the categoryId URL parameter exists, a request will be sent to the BLL to attempt to get all Melodies grouped by the requested Category. If the above parameters are not defined, the API will then attempt to find the *id* or *fileName* URL parameters. If either or both of these URL parameters are found, a request will be sent to the BLL to attempt to get a Melody that matches these values. Finally, if no other URL parameters are provided except the required *token* parameter, the API will send a request to the BLL to get all Melodies, sorted by their Group.

In any scenario, if no Melody or Melodies are found the BLL will return an error to the API, which will be passed back to the client in the form of an HTTP response message. If result(s) are found, the BLL will return the results to the API, which will pass those results back to the client in the form of an HTTP 200 response.

ADD DATA FLOW DIAGRAM HERE

### 3.5.2 GetUserMelodies

The GetUserMelodies API allows the client to get an array of Melody files that belong to the requesting user, or a specific melody if the *id* of the Melody is known. This API function will be called in the form of an HTTP GET request, and will require the *token* URL parameter. If the *token* URL parameters are not provided, an HTTP 401 (Unauthorized) response will be returned to the client.

If the *id* URL parameter is provided the API will send a request to the BLL to get the specified User-created Melody by it’s *id*. If no *id* URL parameter is provided, the API will send a request to the BLL to get all User-created Melodies belonging to the requesting User.

In either scenario, the data will be sent to the BLL where it will be validated. The BLL will get the requesting User data using the provided *Token*, and will send a request to the DAL to retrieve the Melody data. Once the Melody data has been found, the data will be returned to the API layer will it will be provided back to the client in the form of an HTTP 200 response.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

### 3.5.3 AddUserMelody

The AddUserMelody API allows the client to submit a new User-created melody recording. This API will use an HTTP PUT or POST request, and will require a *Token* value, along with a *UserMelody* object. The *UserMelody* object has the following properties:

* ***Name*** – this property will assign a name to the User-created Melody. If no name is given, it will default to the *Name* of the User-uploaded Melody part.
* ***Parts*** – this property represents an array of Melodies. To prevent redundancy, if a melody file already exists on the server the client can specify the *Id* of that Melody. If the Melody is a new Melody, the *FileName* is a required property. All properties for each Melody are as follows:
  + ***Name*** – this property will assign a name to the new Melody. If no name is given, it will default to the *FileName*.
  + ***FileName*** – this is a required property, and must be a unique file name. When the metadata for this request is saved in the database, this file name will be used to validate the upload of the file to the database, along with the *FileUploadToken*.
  + ***Description*** – this is an optional property to give the description to the uploaded Melody.

Once the API sends the request to the BLL, the BLL will validate the data and check to make sure that the Name of the UserMelody object is unique for this user. The BLL will also check each Part within the UserMelody object and search for a Part that contains a FileName property. At least one and only one Part within the UserMelody object can contain a FileName. If a Part is not found with a FileName, the BLL will return an error to the API, which will be passed to the client.

If the UserMelody object is valid, the UserMelody will be sent to the DAL to be saved in the database. Next, the BLL will create a FileUploadToken that will be sent back to the API along with the created UserMelody object. The API will return all of this data to the client in the form of an HTTP 200 response.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

### 3.5.4 DeleteUserMelody

The DeleteUserMelody API allows the client to delete User-created melody recording. For security purposes, only the requesting User can delete a Melody in which they have created. This API will use an HTTP POST request, and will accept a *Token* property along with a *UserMelody* object. The *UserMelody* object will only need to contain the *Id* property. The API layer will serialize the request into a new *UserMelody* object and send that object to the BLL.

This new *UserMelody* object will be sent to the BLL where it will be validated. The BLL will find the User using the provided *Token*, and lookup the requesting UserMelody to be deleted using the provided *UserMelody* object. If the session cannot be validated using the *Token*, or if the *UserMelody* cannot be found by its *Id*, or if the requesting User did not create the found UserMelody, an error will be returned from the BLL to the API, which will be sent back to the client in the form of an HTTP 400 response.

If all data is validated in the BLL, the BLL will send a delete request to the DAL to delete the UserMelody, along with the Melody file that was created by the User. As with any type of API delete request, the data is never actually deleted from the database, but rather the *IsDeleted* flag will be set in the database, which will prevent the DAL from retrieving this data from being used by any other API calls.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

### 3.5.5 GetLoop

The GetLoop API will allow the client to get any User-created Melody Loops. Melody Loops are multiple UserMelody records, created by 1 or many Users, chained together and played repeatedly in succession to form a loop. This API will use an HTTP GET request, and will require a *token* URL parameter. Optionally, this API will also accept the following parameters:

* ***id*** – allows the client to obtain a Loop by its Id.
* ***userId***and ***name*** – allows the client to obtain a Loop by the Name of the loop combined with the User Id of the creator of the Loop.

The API will first attempt to send a request to the BLL to get a specific Loop using the *id* URL parameter. If the *id* parameter is not provided, the API will then attempt to send a request to the BLL to obtain a specific loop using the *userId* and *name* URL parameters (both *userId* and *name* need to be provided). If no other URL parameters are supplied other than the *token* the API will send a request to the BLL to get all Loops that are created by the requesting User, using the *token* to identify the requestor.

Once the BLL has received any of the above calls from the API it will first attempt to lookup the requested Loop(s) via a call to the DAL. If Loop(s) are found, the BLL will then send a request to the DAL to obtain the Loop Parts. After each Loop Part is obtained for each Loop, the BLL will request the User Loop data that’s associated to each Loop Part via a call to the DAL. If all of these requests from the BLL to the DAL are successful, the BLL will sort all of this data and send it back to the API. The API will send the received data back to the client in the form of an HTTP 200 response.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

### 3.5.6 AddUserLoop

The AddUserLoop API will allow a client to create a new Loop. This API will user an HTTP POST request, and will allow the client to perform 2 operations: 1 to convert an existing User Melody into a Loop, or 2, to Create a new User Melody and convert the created User Melody into a Loop.

There are currently no restrictions on the API client’s ability to convert existing User Melodies into Loops other than a valid *Token*. A given User has the ability to convert any User Melody created by any User into a Loop. Furthermore, the User triggering the AddUserLoop API will be the owner of the Loop, even though the User that owns the Loop may not own the User Melody or Melodies that make up the Loop.

This API will require a *Token* property and a *Loop* object. A *Loop* object consists of a required *Name* property and an array of *Parts* objects. Each *Part* object contains the following properties:

* ***UserMelody*** (required) – this is a required object that will let the API know which UserMelody will be used for a given Loop Part. If the client wishes to use an existing UserMelody, they must provide only the *Id* property within the *UserMelody* object. If the client is creating a new UserMelody, please refer to the data structure outlined in section 3.5.3 AddUserMelody for details on a *UserMelody* object (Note: the AddUserMelody API call does **not** need to be called prior to calling the AddUserLoop API. Only refer to the data structure for the *UserMelody* object).
* ***StartTime*** (optional) – this property is represented as a time span formatted as a string in the following format: “{HH:MM:SS}”, so one second would be represented as “00:00:01”. If milliseconds are required, it can be represented as “00:00:01.0010000”. This property will hold the data to let the client know when this track needs to start playing. If no value is defined, it will default to “00:00:00” when saved in the database.
* ***StartEffect*** (optional) – this property is represented as a string and will hold data to let the client know what type of audio effect needs to occur for this Part. Valid values currently are “FadeIn”, “FadeOut”, and “CrossFade”. If no value is provided, it will default to 0, which will represent an undefined value.
* ***StartEffectDuration*** (optional) – this property is represented as a time span formatted as a string. It will hold data to let the client know how long the duration of the *StartEffect* will be for this Part.
* ***EndTime*** (optional) – this property is represented as a time span formatted as a string. This property will hold the data to let the client know when this track needs to stop playing. If no value is defined, it will default to “00:00:00” when saved in the database. If this value is defaulted client should then assume to let the entire audio file play before starting the next Part.
* ***EndEffect*** (optional) – this property is represented as a string and will hold data to let the client know what type of audio effect needs to occur for this Part. Valid values currently are “FadeIn”, “FadeOut”, and “CrossFade”. If no value is provided, it will default to 0, which will represent an undefined value.
* ***EndEffectDuration*** (optional) – this property is represented as a time span formatted as a string. It will hold data to let the client know how long the duration of the *EndEffect* will be for this Part.

Once the API has serialized the request it will pass the data to the BLL. The BLL will first check for a valid session using the *Token*. If a valid session is found, the BLL will check to make sure the *Name* property is set within the *Loop* object. Next, the BLL will send a request to the DAL to make sure that the requesting User has not already created a *Loop* with the same *Name*. If these conditions pass, the BLL will send a request to the DAL to create a new Loop in the database.

Once a Loop has been created in the database, the BLL will then begin to create or add each Part of the loop. It will check each Part and determine if the UserMelody within the Part has an *Id* property defined. If an *Id* is defined, a new Loop Part will be created with the found User Melody *Id*. If no Id is defined, the BLL will then attempt to create a new UserMelody using the same data flow outlined in section 3.5.3 AddUserMelody. The BLL will repeat this process for each defined Loop Part.

If the Loop Part creation fails at any point due to improperly formatted or missing data, the BLL will send requests to the DAL to delete every Loop Part that has been successfully created, and then delete the Loop record. It will then send an error message to the API, which will be passed to the client in the form of an HTTP 400 response.

If the Loop Part creation succeeds, all information will be serialized by the BLL and sent to the API, where the created data will be passed back to the client via an HTTP 200 response.

ADD DATA FLOW DIAGRAM HERE

### 3.5.7 AttachToUserLoop

The AttatchToUserLoop API will allow a client to attach a new or existing UserMelody to an existing Loop. This API uses an HTTP POST request and requires a valid *Token* property, a *Loop* object property, and a *LoopPart* object property. The *Loop* object property requires either an *Id* property, or the *UserId* and *Name* properties to find the existing Loop. For the required properties contained inside the *LoopPart* object, please refer to the properties outlined in section 3.5.6 CreateUserLoop.

Once the API has serialized the request it will pass the serialized data to the BLL where it will be validated and processed. The BLL will first check for a valid session using the *Token*. If a valid session is found, the BLL will send a request to the DAL to retrieve the existing loop by either the Loop *Id*, or the Loop *Name* and *UserId* depending on the properties supplied in the HTTP request. Once the Loop is found the BLL will process the LoopPart object. The BLL will first check the LoopPart and determine if the UserMelody within the LoopPart has an *Id* property defined. If an *Id* is defined, a new LoopPart will be created with the found User Melody *Id*. If no Id is defined, the BLL will then attempt to create a new UserMelody using the same data flow outlined in section 3.5.3 AddUserMelody.

If the Loop Part creation fails at any point due to improperly formatted or missing data, the BLL will stop processing the request and send an error message to the API, which will be passed to the client in the form of an HTTP 400 response.

If the Loop Part creation succeeds, all data for the Loop will be retrieved from the DAL and sent to the API, where the created data will be passed back to the client via an HTTP 200 response.

ADD DATA FLOW DIAGRAM HERE

### 3.5.8 DeleteUserLoop

The DeleteUserLoop API will allow a client to delete an existing Loop. This API uses an HTTP POST request and requires a valid *Token* property. Depending on the other properties supplied in the POST request, this API can perform 2 different functions – deleting a Loop, or deleting a Part from a Loop. To delete a Loop, a *Loop* object must be supplied in the HTTP POST request containing the *Id* of the Loop. To delete a Part from a Loop, a *UserMelody* object with the *Id* of the UserMelody must be supplied in addition to the *Loop* object containing the *Id* of the Loop. The API will serialize all of the data supplied in the HTTP POST request and send it to the BLL where it will be validated.

Once the BLL receives data from the API it will first validate the *Token* as a valid session and will retrieve the User associated with the session via the DAL. The BLL will then send a request to the DAL to retrieve the Loop record via the supplied *Id* property. If a *UserMelody* object was also supplied in the request, the BLL will then attempt to find the UserMelody by its supplied *Id* property.

If a UserMelody record is found, the BLL will verify that the requesting User owns either the UserMelody or the Loop, and will send a request to the DAL to delete the UserMelody Part from the Loop. If a UserMelody record is not found, the BLL will verify that the requesting User owns the Loop, and will send a request to the DAL to delete the Loop and all Parts associated with the Loop. As with any type of API delete request, the data is never actually deleted from the database, but rather the *IsDeleted* flag will be set in the database, which will prevent the DAL from retrieving this data from being used by any other API calls. Once the record(s) have been deleted, the BLL will send a response back to the API, which will trigger the API to return an HTTP response to the client – an HTTP 202 Accepted response in the case of the entire Loop being deleted, or an HTTP 200 response containing the remaining Loop data in the case of a Loop Part being deleted.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

## 3.6 Stations API

The function calls exposed by the Stations API allow the client to interact with every aspect of a Station.

Users create stations provide a public forum for Posts (StationMessages) and User-created Melodies to be viewed by other Users. Users can follow existing Stations, as well as Like and Comment on Posts created by the Station owner. Station Posts that contain a UserMelody can be transformed into a Melody Loop by any of the Followers of that Station, allowing collaboration between multiple individuals.

All functions exposed by this API will require a valid session token to authenticate the User session.

All errors are handled in the BLL, with error messages being translated to more friendly and informative messages, based on the use context of the call.

### 3.6.1 CreateStation

The CreateStation API will allow a client to create a new Station for a User. Users currently have no limit to the amount of Stations they can create. This API uses an HTTP PUT or POST request and requires, at minimum, a *Token* property and a *Station* object property containing a *Name* property. Additionally, the HTTP request can contain an *Image* object property containing a *FileName* property, and a *Categories* property which will contain an array of *Category* objects. Each Category object must have either an *Id* property or a *Name* and *ParentId* property defined. This information will be serialized by the API and sent to the BLL for validation.

Once the BLL receives the serialized data, it will first validate the *Token* property and send a request to the DAL to retrieve the User data associated with the *Token*. Next, the BLL will validate the *Name* within the Station object and make sure that a Station has not already been created by the requesting User with the same name. The BLL will then send a request to the DAL to save the Station data.

If an *Image* object was included in the API request, the BLL will then check to see that the FileName provided in the Image object does not exist in the database and will send a request to the DAL to save the image metadata. The BLL will then create a FileUploadToken to be passed back to the client to allow the client to upload the file to the server.

If a *Categories* array object was included in the API request, the BLL will then send a request to the DAL to find each Category in the *Categories* array using either the *Id*, or *Name* and *ParentId* properties of each Category. Once the Categories have been validated the BLL will add the created Station to each Category.

Once the BLL has validated all of the data and the DAL has saved all data to the database, the BLL will serialize all of the saved data and return it to the API to be passed back to the client in the form of an HTTP 200 response.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

### 3.6.2 UpdateStation

The UpdateStation API will allow a client to update an existing Station for a User. This API uses an HTTP POST request and requires, at minimum, a *Token* property and a *Station* object property containing an *Id* property. The *Station* object can also contain an optional *Name* property. Optionally, the HTTP request can contain an *Image* object property containing a *FileName* property, and a *Categories* property which will contain an array of *Category* objects. Each Category object must have either an *Id* property or a *Name* and *ParentId* property defined. This information will be serialized by the API and sent to the BLL for validation.

Once the BLL receives the serialized data, it will first validate the *Token* property and send a request to the DAL to retrieve the User data associated with the *Token*. Next, the BLL will validate the *Id* within the Station object and make sure that the User owns the Station matching the provided *Id*. If a *Name* property exists within the Station object, the *Name* will be validated to assure that a Station has not already been created by User with the same name.

If an *Image* object was included in the API request, the BLL will then check to see that the FileName provided in the Image object does not exist in the database and will send a request to the DAL to save the image metadata. Next, the BLL will send a request to the DAL to delete the existing Station image(if any). The BLL will then create a FileUploadToken to be passed back to the client to allow the client to upload the new Image file to the server.

If a *Categories* array object was included in the API request, the BLL will then send a request to the DAL to find each Category in the *Categories* array using either the *Id*, or *Name* and *ParentId* properties of each Category. Once the Categories have been validated the BLL will add the created Station to each Category.

Finally, the BLL will then send a request to the DAL to update any modified Station data.

Once the BLL has validated all of the data and the DAL has saved all data to the database, the BLL will serialize all of the saved data and return it to the API to be passed back to the client in the form of an HTTP 200 response.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

### 3.6.3 DeleteStation

The DeleteStation API allows the client to delete User-created Station. For security purposes, only the requesting User can delete a Station in which they have created. This API will use an HTTP POST request, and will accept a *Token* property along with a *Station* object. The *Station* object will only need to contain either an *Id* property or a *Name* property. The API layer will serialize the send that data to the BLL for validation.

Once the BLL receives the serialized data, it will first validate the *Token* property and send a request to the DAL to retrieve the User data associated with the *Token*. Next the BLL will send a request to the DAL to retrieve the Station either by its *Name* or *Id* properties, and validate that the requesting User is the owner of the found Station.

If all data is validated in the BLL, the BLL will send a delete request to the DAL to delete the Station, along with the Image(if any) associated with the Station, and finally, remove the Station from any associated Categories. As with any type of API delete request, the data is never actually deleted from the database, but rather the *IsDeleted* flag will be set in the database, which will prevent the DAL from retrieving this data from being used by any other API calls.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

### 3.6.4 DeleteStationCategories

The DeleteStationCategories API allows the client to remove a User-created Station from specified Categories. For security purposes, only the requesting User can remove a Station in which they have created from any Categories. This API will use an HTTP POST request, and will accept a *Token* property along with a *Station* object, as well as a *Categories* property that will contain an array of *Category* objects. The *Station* object will only need to contain either an *Id* property or a *Name* property. Each Category object within the *Categories* property will require either the *Id* property, or the *Name* and *ParentId* properties. The API layer will serialize the send that data to the BLL for validation.

Once the BLL receives the serialized data, it will first validate the *Token* property and send a request to the DAL to retrieve the User data associated with the *Token*. Next the BLL will send a request to the DAL to retrieve the Station either by its *Name* or *Id* properties, and validate that the requesting User is the owner of the found Station. The BLL will send a request to the DAL to retrieve each Category using either the *Id* of the Category, or the *Name* and *ParentId* of the Category.

If all data is validated in the BLL, the BLL will send a delete request to the DAL to delete the Station from all requested Categories. As with any type of API delete request, the data is never actually deleted from the database, but rather the *IsDeleted* flag will be set in the database, which will prevent the DAL from retrieving this data from being used by any other API calls.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

### 3.6.5 GetStations

The GetAllStations API allows the client to retrieve a list of all created Stations. This API will use an HTTP GET request, and will require, at minimum, a *token* URL parameter. This API can also accept the following URL parameters:

* ***stationId –*** the Id of the Station
* ***name –*** the Name of the Station
* ***userId –*** the UserId of the Station
* ***displayName –*** the DisplayName of the User
* ***email –*** the EmailAddress of the User
* ***categoryId –*** the Id of the Category
* ***category –*** the Name of the Category
* ***categoryParentId –*** the ParentId of the Category

First, the API will attempt to find a valid *stationId* URL parameter and send a request to the BLL to find a specified Station with the provided *stationId.* If the *stationId* is not provided the API will attempt to find the *name* and *userId* URL parameters and send a request to the BLL to find a specified Station by the Name of the Station and the UserId of the Station owner.

If the *stationId* and *name* URL parameters are not defined, the API will then attempt to send a request to the BLL to retrieve all stations by a specified User using the *userId* URL parameter. If the *userId* URL parameter is not defined, the API will then attempt to send a request to the BLL to find all stations by a specified User using the *displayName* or *email* URL parameters.

If those URL parameters are not defined, the API will attempt to send a request to the BLL retrieve all Stations that are categorized in a specified Category using the *categoryId* URL parameter. If the *categoryId* URL parameter is not defined, the API will then attempt to send a request to the BLL to retrieve all Stations that are categorized in a specified Category using the *category* and *categoryParentId* URL parameters.

Finally, if no URL parameters are defined, the API will send a request to the BLL to retrieve all Stations that belong to the requesting User using the required *token* URL parameter*.*

If the BLL finds 1 or more Stations in any of the above scenarios, the BLL will send requests to the DAL to retrieve the Image and Category data for each Station. The BLL will send all of this data back to the API layer, where the data will be serialized and sent back to the client in the form of an HTTP 200 response.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

### 3.6.6 GetAllStations

The GetAllStations API allows the client to retrieve a list of all created Stations. This API will use an HTTP GET request, and will require a *token* URL parameter. The API layer will serialize the send that data to the BLL for validation.

Once the BLL receives the serialized data, it will first validate the *token* URL parameter value and send a request to the DAL to retrieve the User data associated with the *token*. Next the BLL will send a request to the DAL to retrieve all active Stations. Once the BLL has retrieved the Stations, it will collect the Image and Category data associated to each Station. Finally, the BLL will return all data to the API, where it will be serialized and sent to the client in an HTTP 200 response.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

### 3.6.7 GetStationFollowers

The GetStationFollowers API allows the client to retrieve a list of followers for a specified Station. This API will use an HTTP GET request, and will require a *token* URL parameter and an *id* URL parameter that will contain the Id of the Station. The API layer will serialize the send that data to the BLL for validation.

Once the BLL receives the serialized data, it will first validate the *token* URL parameter value and send a request to the DAL to retrieve the User data associated with the *token*. Next the BLL will send a request to the DAL to retrieve the requested Station. Once the BLL has retrieved the Station, it will send another request to the DAL to retrieve an array of User records that follow the Station. For security reasons, any sensitive User information will be removed from every User in the result set. Finally, the BLL will return all data to the API, where it will be serialized and sent to the client in an HTTP 200 response.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

### 3.6.8 FollowStation

The FollowStation API will let the client send a request for a User to follow a Station. This API will use an HTTP POST request, and will require the *Token* property and a *Station* object property, with the *Station* object containing either an *Id* property, or the *Name* and *UserId* properties. The API layer will serialize the send that data to the BLL for validation.

Once the BLL receives the serialized data, it will first validate the *Token* property value and send a request to the DAL to retrieve the User data associated with the *Token*. Next the BLL will send a request to the DAL to retrieve the requested Station. Once the BLL has retrieved the Station, it will send another request to the DAL to add the requesting User as a Station follower. Finally, the BLL will retrieve the Station Image and Station Categories (if any) and return this information to the API. The API will serialize this information and send it back to the client in an HTTP 200 response.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

### 3.6.9 UnfollowStation

The UnfollowStation API will let the client send a request for a User to unfollow a Station. This API will use an HTTP POST request, and follows the same data structure and data flow as the FollowStation API (section 3.6.8).

The only difference in these two calls is that after the UnfollowStation request has been validated at the BLL, the BLL will send a request to the DAL to remove the User as a Follower of the requested station. The BLL will then return a response to the API layer, and the API will send an HTTP 202 Accepted response back to the client.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

### 3.6.10 CreateStationPost

The CreateStationPost API will allow the client send a public Message to a Station. This API will use an HTTP PUT or POST request, and will require a *Token* property, a *Station* object property, and a *Message* object property. The Station object property will require either an *Id* property or the *Name* and *UserId* properties. The *Message* object property follows the same requirements as any other API sending a *Message* object (please refer to section 3.4 Messages API). The API layer will serialize the send that data to the BLL for validation.

Once the BLL receives the serialized data, it will first validate the *Token* property value and send a request to the DAL to retrieve the User data associated with the *Token*. Next, the BLL will send a request to the DAL to retrieve the requested Station. After the BLL has retrieved the Station record, it will validate that the requesting User is a follower of the requested Station. Next, the BLL will send a request to the DAL to create a new Message record. If the Message record contains Image, Video, or UserMelody metadata, those records will be added to the database, and the BLL will create a FileUploadToken that will be returned to the client. Finally, the BLL will associate the newly created Message to the Station, and will send a new StationMessage object back to the API. The API will serialize this information and send it back to the client in an HTTP 200 response.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

### 3.6.11 CreateStationMessage

The CreateStationMessage API will allow the client to send a private Message to a Station. This API will use an HTTP PUT or POST request, and follows the same data structure and data flow as the CreateStationPost API (see section 3.6.10).

There are two differences between the CreateStationPost and CreateStationMessage APIs:

1. The CreateStationPost API creates and returns a new StationMessage object with the *IsPrivate* property set to false, while the CreateStationMessage API creates and returns a new StationMessage object with the *IsPrivate* property set to true.
2. The CreateStationPost API validates at the BLL that the requesting User is a follower of the requested Station, while the CreateStationMessage API does not.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

### 3.6.12 CreateStationPostReply

The CreateStationMessageReply API will allow a client to reply to a received Station Message. This API will use an HTTP POST request, and requires a *Token* property, a *StationMessage* object property, and a *Message* object property. The *StationMessage* object property only requires an *Id* property. The *Message* object property follows the same requirements as any other API sending a *Message* object (please refer to section 3.4 Messages API). The API layer will serialize the send that data to the BLL for validation.

Once the BLL receives the serialized data, it will first validate the *Token* property value and send a request to the DAL to retrieve the User data associated with the *Token*. Next, the BLL will send a request to the DAL to retrieve the requested StationMessage. After the BLL has retrieved the StationMessage record, it will check to see if the requested StationMessage has the IsPrivate property set to false. Next, the BLL will check to see if the requesting User is a Follower of the Station to which the retrieved StationMessage belongs.

Finally, the BLL will send a request to the DAL to create a new Message record. If the Message record contains Image, Video, or UserMelody metadata, those records will be added to the database, and the BLL will create a FileUploadToken that will be returned to the client. Finally, the BLL will associate the newly created Message to the Station, and will send a new StationMessage object back to the API. The API will serialize this information and send it back to the client in an HTTP 200 response.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

### 3.6.13 GetStationPosts

The GetStationPosts API will allow the client to retrieve an array of StationPosts for a specified Station. This API will use an HTTP GET request and will require a *token* URL parameter. In addition, the API will also require either an *id* URL parameter or a *postId* URL parameter. The API layer will serialize the send that data to the BLL for validation.

Once the BLL receives the serialized data, it will first validate the *token* URL parameter value and send a request to the DAL to retrieve the User data associated with the *token*. If the *postId* URL parameter was defined in the API call, the BLL will send a request to the DAL to retrieve a specific post using the supplied *postId*. If the *id* URL parameter was defined, the BLL will send a request to the DAL to retrieve all Posts for the Station that has the supplied *id*. The BLL will then send this data back to the API, where the API will serialize this data and send it back to the client in the form of an HTTP 200 response.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

### 3.6.14 GetStationMessages

The GetStationMessages API will allow the client to retrieve an array of StationPosts for a specified Station. This API will use an HTTP GET request and will require a *token* URL parameter. In addition, the API will also require either an *id* URL parameter or a *messageId* URL parameter. The API layer will serialize the send that data to the BLL for validation.

Once the BLL receives the serialized data, it will first validate the *token* URL parameter value and send a request to the DAL to retrieve the User data associated with the *token*. If the *messageId* URL parameter was defined in the API call, the BLL will send a request to the DAL to retrieve a specific StationMessage using the supplied *messageId*. The BLL will verify that the requesting User is either the User who sent the requested StationMessage, or the User is the owner of the Station to which the requested StationMessage belongs.

If the *id* URL parameter was defined, the BLL will send a request to the DAL to retrieve all StationMessages for the Station that has the supplied *id*. The BLL will then verify that the requesting User is the owner of the Station to which the requested StationMessages belong.

Once the BLL has retrieved the requested Message(s), it will then send this data back to the API, where the API will serialize this data and send it back to the client in the form of an HTTP 200 response.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

### 3.6.15 LikeStationPost

The LikeStationPost API will allow the client to send a request to like or favorite a StationPost. This API will use an HTTP POST request, and will require a *Token* property and a *StationMessage* object property containing an *Id* property. The API layer will serialize the send that data to the BLL for validation.

Once the BLL receives the serialized data, it will first validate the *Token* property value and send a request to the DAL to retrieve the User data associated with the *Token*. Next, the BLL will send a request to the DAL to retrieve the requested StationMessage. After the BLL has retrieved the StationMessage record, it will check to see if the requested StationMessage has the IsPrivate property set to false. Next, the BLL will check to see if the requesting User is a Follower of the Station to which the retrieved StationMessage belongs.

Finally, the BLL will send a request to the DAL to store a Like for the requested StationMessage. The BLL will then retrieve all StationMessage data, including any Likes or Reply messages, and return that data to the API. The API will then serialize the retrieved data and send it back to the client in the form of an HTTP 200 response.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

### 3.6.16 UnlikeStationPost

The UnlikeStationPost API will allow the client to send a request to un-like or un-favorite a liked StationPost. This API will use an HTTP POST request, and will follow the same data structure and data flow as the LikeStationPost API (see section 3.6.15).

There are two differences between the LikeStationPost API and the UnlikeStationPost API:

1. The UnlikeStationPost API does not validate that the requesting User is a Follower of the requested Station, while the LikeStationPost API does.
2. The LikeStationPost API creates a Like record in the database for the requested StationMessage, while the UnlikeStationPost API sets the IsDeleted property of the Like record in the database to true.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

### 3.6.17 DeleteStationMessage

The DeleteStationMessage API will allow the client to send a request to delete a Station Message or a Station Post. This API will use an HTTP POST request, and will require a *Token* property and a *StationMessage* object property containing an *Id* property. The API layer will serialize the send that data to the BLL for validation.

Once the BLL receives the serialized data, it will first validate the *Token* property value and send a request to the DAL to retrieve the User data associated with the *Token*. Next, the BLL will send a request to the DAL to retrieve the requested StationMessage.

The BLL will then check to see if the retrieved StationMessage has the *IsPrivate* property set to true. If so, the BLL will validate that the requesting User is the owner of the Station to which the retrieved StationMessage belongs before sending a request to the DAL to delete the StationMessage. If the *IsPrivate* property is set to false, the BLL will validate that the requesting User is either the owner of the Station to which the retrieved Station message belongs, or if the requesting User is the sender of the retrieved StationMessage before sending a request to the DAL to delete the StationMessage. As with any type of API delete request, the data is never actually deleted from the database, but rather the *IsDeleted* flag will be set in the database, which will prevent the DAL from retrieving this data from being used by any other API calls.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE

## 3.7 Uploads API

The function calls exposed by the Uploads API will allow Image, Audio, and Video files to be uploaded and saved to the server, which can then be retrieved through exposed API calls, outlined above.

All function calls within the Uploads API will require two tokens - the first of which is a valid session token which is required in most other API calls, and the second is a FileUploadToken, which is returned from several other API functions (outlined in sections above). In order for any file upload to be accepted and saved on the server, both tokens must be provided and validated with every API call to upload a file. Failing to provide either a valid session token or file upload token will result in an error.

All errors are handled in the BLL, with error messages being translated to more friendly and informative messages, based on the use context of the call.

### 3.7.1 UploadFile

The UploadFile API will allow a User upload a Video, Image, or Audio file to the server. This function will be called in the form of an HTTP POST request, with the body of the request only containing the multi-part/form-data content of the file to be uploaded. In the URL of the request, the uploading User’s Session *Token* and the *FileUploadToken* must be included.

Ex: /v0.1/Upload/{Session Token}/{File Upload Token}

This data will be sent to the BLL where it will be validated. The BLL will get the requesting User data using the provided *Token*, and will send a request to the DAL to retrieve the requested *FileUploadToken*. The BLL will verify that the *FileUploadToken* belongs to the requesting User, and that the *FileUploadToken* has not expired. Once the User and *FileUploadToken* have been validated, the body of the UploadFile API request will begin to be processed.

While processing the body of the FileUpload API request, there is a series of checks that the form data must pass before the file is accepted. The checks are as follows:

1. The body of the request is processed using a technique called “Mime Sniffing”, where the data of the requested file is checked to verify that the body of the request matches the mime-type that is defined in the header of the request. This is put in place as a security measure to ensure that viruses and other malware aren’t uploaded to the server.
2. The file name of the uploaded file is checked against the *FileName* of the *FileUploadToken* that is stored in the database. If the file name in the header of the HTTP request does not match the filename stored in the database, the upload will fail.

If the above checks pass, the file is then uploaded to the server, and the file path and metadata for the uploaded file are retrieved and sent back to the API. The API will then format the uploaded file data into an HTTP 200 message and send the file data back to the client.

Regardless of whether or not the FileUpload API passes or fails, the *FileUploadToken* used in the API request will be deleted. This is yet another security measure put in place to ensure that files cannot attempt to be uploaded multiple times with successive failures.

If any errors occur at any level, they will be passed back up to the API and be returned to the client in the form of an HTTP response containing the error description.

ADD DATA FLOW DIAGRAM HERE