InstaMelody Application API

Table of Contents

1. Research 4

1.1 Database Schema 4

1.1.1 Schema Name 4

1.1.2 Schema Tables for Authentication 4

1.1.3 Schema Diagram 5

1.1.4 Data Access 6

1.2 System Analysis 7

1.2.1 Scope Definition 7

1.2.2 Problem Analysis 7

1.2.3 Requirements Analysis 7

1.3 Application Structure 8

1.3.1 Application Components 8

1.3.2 Logical Design 9

1.3.3 API Functions 10

1.3.4 Application Logging 10

2. Planning 11

2.1 Goals 11

2.2 Ground Rules and Assumptions 11

2.3 Environment and Software Specification 12

2.4 Deliverables 12

2.5 Available Resources 12

3. Development 13

3.1 User API 13

3.1.1 CreateUser 13

3.1.2 GetUser 15

3.2 Authentication API 16

3.2.1 Authenticate (login) 17

3.2.2 ValidateSession 18

3.2.3 EndSession (logout) 18

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Date** | **Name** | **Description** |
| 0.1 | 17 June 2015 | Jeff Dennis | Draft created |

# 1. Research

## 1.1 Database Schema

### 1.1.1 Schema Name

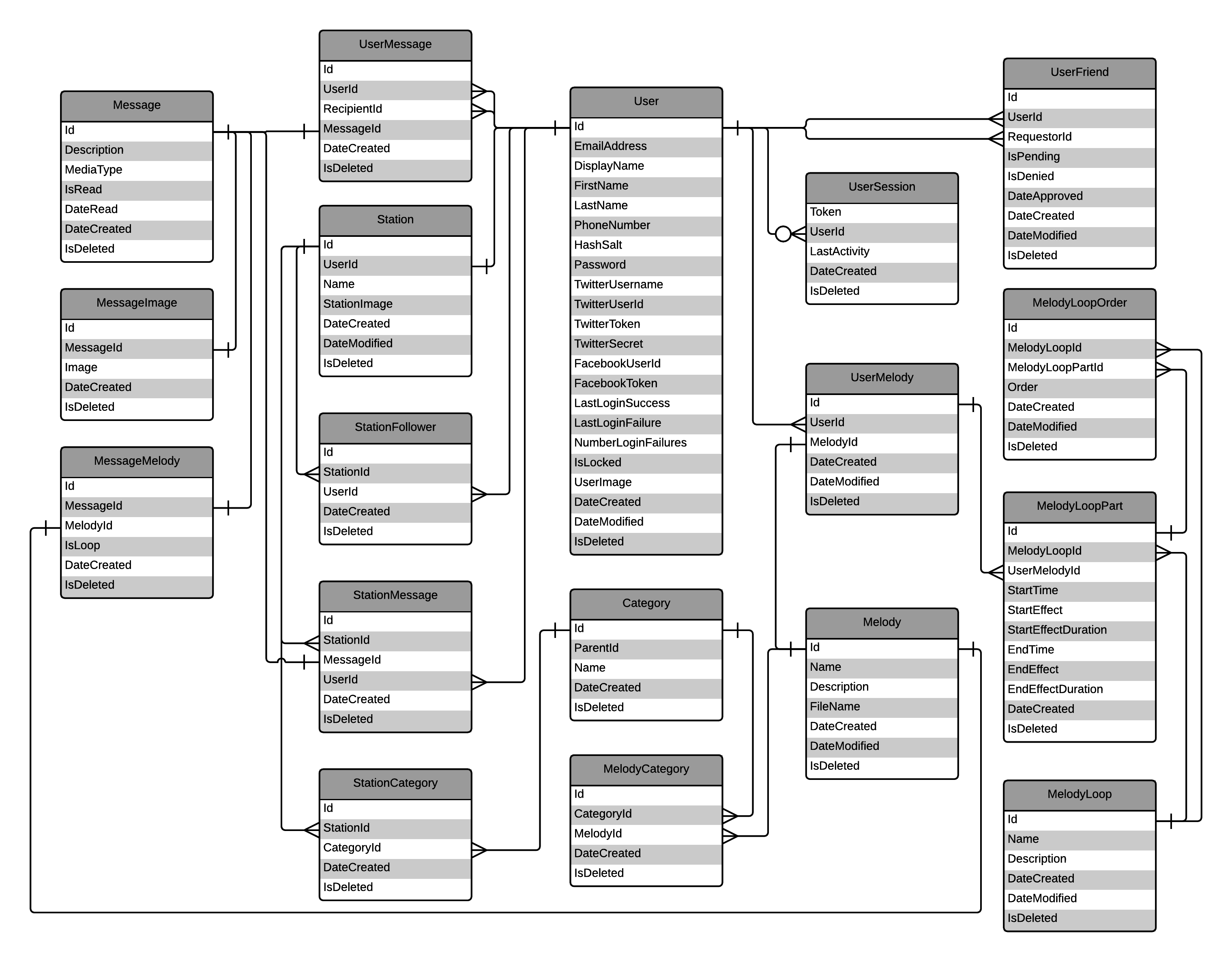
The schema name “dbo” will be used for database components that are part of this API. This schema will have interactions with other application schemas, as needed.

### 1.1.2 Schema Tables for Authentication

The database will consist of the following tables:

|  |  |  |
| --- | --- | --- |
| **Table Name** | **Description** | **Primary Key** |
| Users | Holds data to identify and authenticate users. | UNIQUEIDENTIFIER |
| Stations | Holds data to identify stations. | Id; autoincrement |
| 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 |
| UserMessages | Holds data to maintain relationships between Users and Messages. | Id; autoincrement |
| UserMelodies | Holds data to maintain relationships between Users and Melodies. | Id; autoincrement |
| MessageImages | Holds data to identify images within Messages content | Id; autoincrement |
| MessageMelodies | Holds data to identify Melodies within Messages content | Id; autoincrement |
| MelodyCategories | Holds data to maintain many-to-many relationships between Melodies and Categories. | Id; autoincrement |
| StationMessages | Holds data to maintain relationships between Stations and Messages. | Id; autoincrement |
| StationCategories | Holds data to maintain many-to-many relationships between Stations and Categories. | Id; autoincrement |
| StationFollowers | Holds data to maintain relationships between Stations and Users. | Id; autoincrement |
| MelodyLoops | Holds data to identify Melodies that are chained together to form loops. | Id; autoincrement |
| MelodyLoopParts | Holds data to maintain many-to-many relationships between Melodies and MelodyLoops. | Id; autoincrement |
| MelodyLoopOrder | Holds data to identify the order of MelodyLoopParts that make up a MelodyLoop. | 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

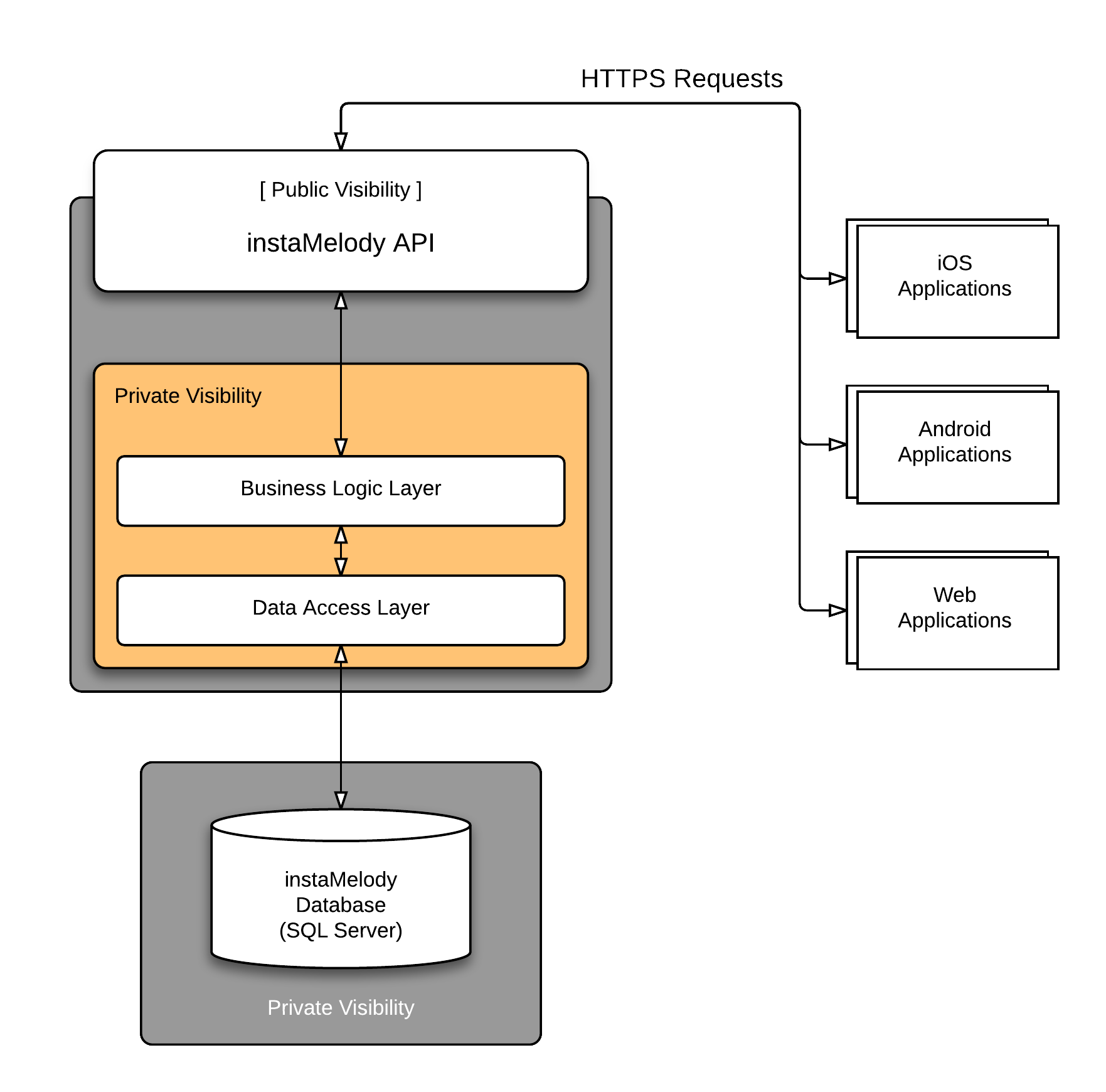
* Functional design documentation
* Technical design documentation
* Establish standard authentication models and behaviors
* Allow users to create new user accounts
* Allow users to create melodies and send them to other users in the form of messages
* Allow users to create stations and post melodies 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

#### 1.3.3.2 Authentication API Functions

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

### 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 |
| Ahmed Bakir | Front-end/mobile programming, testing |
| Jeff Dennis | API Programming, unit 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.



#### 3.1.1.1 CreateUser Example

POST /v0.1/User/New  
PUT /v0.1/User/New

*{*

*"EmailAddress" : "test@testy.com",*

*"DisplayName" : "testeraccount",*

*"FirstName" : "testy",*

*"LastName" : "testerson",*

*"PhoneNumber" : "123-555-1234",*

*"Password" : "TestPW",*

*}*

*HTTP/1.1 200 OK*

*{*

*"Id": "5651a0ad-b934-4be8-88f2-c380f884adf5",*

*"EmailAddress": "test@testy.com",*

*"DisplayName": "testeraccount",*

*"FirstName": "testy",*

*"LastName": "testerson",*

*"PhoneNumber": "123-555-1234",*

*"HashSalt": "",*

*"Password": "",*

*"TwitterUsername": "",*

*"TwitterUserId": "",*

*"TwitterToken": "",*

*"TwitterSecret": "",*

*"FacebookUserId": "",*

*"FacebookToken": "",*

*"LastLoginSuccess": "0001-01-01T00:00:00",*

*"LastLoginFailure": "0001-01-01T00:00:00",*

*"NumberLoginFailures": 0,*

*"IsLocked": false,*

*"UserImage": "",*

*"DateCreated": "2015-06-19T23:20:35.747",*

*"DateModified": "2015-06-19T23:20:35.747",*

*"IsDeleted": false*

*}*

### 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 using ADO.NET. 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.

ADD DATA FLOW DIAGRAM HERE

#### 3.1.2.1 GetUser Example

GET /v0.1/User?token=9d0ab021-fcf8-4ec3-b6e3-bb1d0d03b12e&id=5651a0ad-b934-4be8-88f2-c380f884adf5

GET /v0.1/User?token=9d0ab021-fcf8-4ec3-b6e3-bb1d0d03b12e&emailAddress=test@testy.com

GET /v0.1/User?token=9d0ab021-fcf8-4ec3-b6e3-bb1d0d03b12e&displayName=testeraccount

*HTTP/1.1 200 OK*

*{*

*"Id": "5651a0ad-b934-4be8-88f2-c380f884adf5",*

*"EmailAddress": "test@testy.com",*

*"DisplayName": "testeraccount",*

*"FirstName": "testy",*

*"LastName": "testerson",*

*"PhoneNumber": "123-555-1234",*

*"HashSalt": "",*

*"Password": "",*

*"TwitterUsername": "",*

*"TwitterUserId": "",*

*"TwitterToken": "",*

*"TwitterSecret": "",*

*"FacebookUserId": "",*

*"FacebookToken": "",*

*"LastLoginSuccess": "2015-06-19T23:36:15.33",*

*"LastLoginFailure": "0001-01-01T00:00:00",*

*"NumberLoginFailures": 0,*

*"IsLocked": false,*

*"UserImage": "",*

*"DateCreated": "2015-06-19T23:35:50.7",*

*"DateModified": "2015-06-19T23:35:50.7",*

*"IsDeleted": false*

*}*

## 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 request (POST) that will hold a *DisplayName* or *EmailAddress* and *Password* in JSON form. 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 *User* object to hold the *DisplayName* or *EmailAddress* and *Password* 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: an updated User object in the case of success, or an exception in the case of a failure. 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

#### 3.2.1.1 Authenticate Example

POST /v0.1/Auth/User

*{*

*"EmailAddress" : "test@testy.com",*

*"DisplayName" : "testeraccount",*

*"Password" : "TestPW",*

*}*

*HTTP/1.1 200 OK*

*{*

*"Token": "9d0ab021-fcf8-4ec3-b6e3-bb1d0d03b12e"*

*}*

### 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 *id* and the *token* 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 UserAuth object. If no matching record is found, then it will return a default UserAuth object (UserId property set to zero). 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

#### 3.2.2.1 ValidateSession Example

GET /v0.1/Auth?token=9d0ab021-fcf8-4ec3-b6e3-bb1d0d03b12e&id=5651a0ad-b934-4be8-88f2-c380f884adf5

*HTTP/1.1 200 OK*

*{*

*"Token": "9d0ab021-fcf8-4ec3-b6e3-bb1d0d03b12e"*

*}*

### 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 *UserId* properties. If both 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

#### 3.2.3.1 EndSession Example

POST /v0.1/Auth/End

*{*

*"Token" : "9d0ab021-fcf8-4ec3-b6e3-bb1d0d03b12e ",*

*"UserId" : "5651a0ad-b934-4be8-88f2-c380f884adf5"*

*}*

*HTTP/1.1 200 OK*

*{*

*"Message" : "User 5651a0ad-b934-4be8-88f2-c380f884adf5 successfully logged out."*

*}*