TagIt.FM

Low-Level Design (LLD) Document

MM

Members: Jeremiah Thomas

Anna Malmberg

Sandhya Joshi

Zachary Wawrzaszek

Stavros Bannoura

**Revisions**

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Primary**  **Author(s)** | **Description of Version** | **Date Completed** |
| **1.0** |  |  |  |

**Review and Approval**

|  |  |  |  |
| --- | --- | --- | --- |
| **Approving Party** | **Version Approved** | **Signature** | **Date** |
| Anna Malmberg | 1.0 | Anna Malmberg | 2020-07-07 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Table of Contents

[1. Introduction 3](#_Toc44838370)

[1.1. Purpose 3](#_Toc44838371)

[1.2. Scope Of This Document 3](#_Toc44838372)

[1.3. Definitions, Acronyms And Abbreviations 3](#_Toc44838373)

[2. Object Oriented Design 4](#_Toc44838374)

[2.1. Classes 4](#_Toc44838375)

[2.2. Detailed Inheritance Hierarchies 4](#_Toc44838376)

[2.2.1. LLD Class Diagram 5](#_Toc44838377)

[2.3. Refinement Of Parent Classes To Child Classes 6](#_Toc44838378)

[2.4 Object Composition 6](#_Toc44838379)

[3. Database Design 6](#_Toc44838380)

[3.1. Chosen Database 6](#_Toc44838381)

[3.2. Tables And Relationships 6](#_Toc44838382)

[3.2.1. User 7](#_Toc44838383)

[3.2.2. User History 7](#_Toc44838384)

[3.2.3. File Tag 7](#_Toc44838385)

[3.2.4. Tag 7](#_Toc44838386)

[3.2.5. Group 8](#_Toc44838387)

[Appendix A 9](#_Toc44838388)

# Introduction

## Purpose

The client, MHC, is an insurance company and they need an application that can quickly locate important medical and financial documents of varying filetype. Customer service representatives, especially, will benefit from TagIt.FM because it will both simplify the search process and reduce the amount of time that they need to keep a customer on hold. This in turn can improve the mood of the customer and decrease frustrations for the employee.

The purpose of this project is to improve and accelerate the process of searching for files in a computer file directory, using the new, and intuitive TagIt.FM application. This document will describe the detailed, low-level design (LLD) of TagIt.FM, including the basic structure of the involved database classes, and their attributes and functions. The LLD document will be used by the developers to guide the design and development process of the application.

## 1.2. Scope Of This Document

The scope of this document is to outline the low-level design details that will be used to develop the application concept into a fully functioning product. The document will include all the object-oriented design details, like determining what the classes will do, and using pseudocode to help illustrate some of the basic coding constructs. It will discuss the inheritance hierarchies that will be involved with the classes, like what the parent classes will oversee, and what the child classes will inherit from them.

The database component of TagIt.FM is described in the last section of the low-level design document. This section will include the pseudocode, database tables and their relationships, and the functions of the database tables. There will also be a description of the normalization process of the application database, that will ensure that there are no redundancies in the database.

## 1.3. Definitions, Acronyms And Abbreviations

\*\*3NF\*\*: Third normal form

\*\*Child\*\*: A class that inherits attributes and/or methods

\*\*Class\*: A user-defined blueprint or prototype from which all objects are created that contains the set of properties and methods that an object of that type has

\*\*CURD\*\*: Create, update, read, and delete

\*\*Foreign key\*\*: A set of attributes that references a primary key

\*\*Inheritance\*\*: A mechanism where a class is derived from another class

\*\*LAN\*\*: Local area network

\*\*Library\*\*: Reusable code, including functions, that can be used for common programming tasks

\*\*Normalization\*\*: Database normalization is the process of structuring a relational database in accordance with a series of so-called normal forms in order to reduce data redundancy and improve data integrity

\*\*Object\*\*: An abstract data type that contain multiple properties and methods, or even other objects as defined by its class

\*\*Parent\*\*: The top-level class from which a child class inherits attributes and methods

\*\*Primary key\*\*: A specific choice of attributes that uniquely specify a tuple in a relation

\*\*Pseudocode\*\*: Informal high-level description of the operating principle of a computer program using the structural convention of a programming language but intended for human reading

\*\*SQL\*\*: Structured query language: a language used in programming that is designed for data held in a relational database management system

\*\*SQLite\*\*: A database that operates directly from library function calls, without a separate process or server

\*\*Tag\*\*: A string without spaces which provides categorical information about its object

\*\*UML\*\*: Unified modeling language

# Object Oriented Design

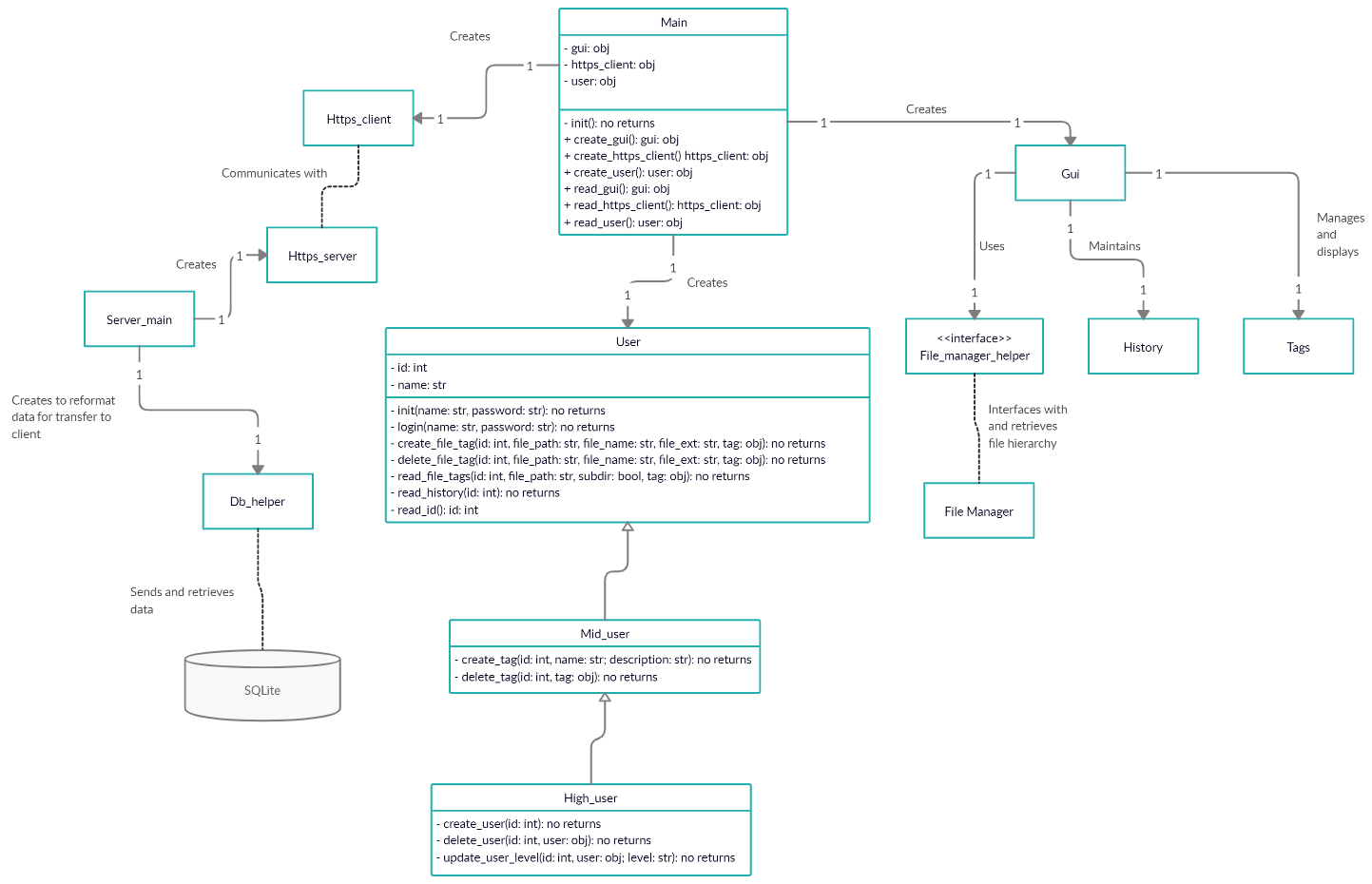
## 2.1. Classes

The classes are illustrated using pseudocode in Appendix A. This provides details on how the Main class, the User class and the User child classes will be set up, including their purpose, properties, constructors, and methods.

## 2.2. Detailed Inheritance Hierarchies

Pictured below is an inheritance hierarchy for the application with details for the Main and User class leg. It also displays how the rest of the classes and database will interact with each other.

### 2.2.1. LLD Class Diagram



## 2.3. Refinement Of Parent Classes To Child Classes

TagIT.FM will make use of four parent classes and two child classes. The Main class is used to run the User classes for the client side of the application. If there is a need for the classes to create more inheritance related tasks to the application design, they may be developed in future iterations.

There are three levels of classes in the application design that handle users, therefore the structure is designed to use the User class as the parent class. The child class Mid\_user will inherit directly from User, and High\_user will inherit from Mid\_user. The purpose of this inheritance is splitting the users into categories depending on their job roles, creating certain roles with dependent permissions within the application. The User class, as the parent class, contains the basic attributes and methods that are inherited by the two child classes. The Mid\_user class, inherits from the User class and adds new methods for creating and deleting tags. Finally, the High\_user class inherits from the Mid\_user class and adds new methods for creating and deleting user accounts and for changing the current permission level on accounts.

## 2.4 Object Composition

The principles of object composition were not a part of this project's design process. Object composition describes a class that references one or more objects of other classes in instance variables and models a "has - a" association between objects. TEAM wanted to take a different route in its design development and there are no instances of object composition used in this project.

# Database Design

## 3.1. Chosen Database

TagIt.FM will be designed to run using a SQLite database. The SQLite database engine is small and fast, which makes it unlikely to impact any business-critical functions or slow down the LAN. SQLite works across a wide variety of operating systems, making it possible for MHC to use the application on any existing hardware. The database content will be updated continuously and automatically, which helps protect the database against losses in the event of a power outage or crash.

The SQLite database will store the user identification parameters, group permissions, filename and tag relations, and user histories for TagIt.FM.

## 3.2. Tables And Relationships

The database table is set up in 3NF using primary keys and foreign keys as the functional dependencies in the database. There are no other transitive dependencies in the table as the five related tables were kept separated to achieve 3NF.

The User History table will be separated from the User table. The User History table will organize the user history and put it into a separate table that has two foreign keys that follow 3NF. The User History table can then add users separately from the tags and delete tags without affecting the users. The User table will be normalized into columns containing the user information, that is, names, dates, and password cryptography. This table will connect to the User History table and the Group table using foreign keys. The Group table will include information that involves CURD tasks for tags, users, and groups. The Tag table will only have a primary key and no foreign keys. The File Tag table will include one primary key and one foreign key. The File Tag table will contain all the file information that is used to organize the tags based on the file path, the name, and the extension type. This table will allow administrators to delete a tag without affecting the other tables.

### 3.2.1. User

The primary key of the User table will be id(int). It will be in charge of transferring data when the other tables are joined. The main purpose of the User table will be to store information for the user in six attributes that handle the account data. The data attributes in the User table will be name(str), creation\_date(date), salt(str), hash(str), update\_date(date) as well as the foreign key group.id(int). The username of the user will be assigned to name, and the account creation date to creation\_date. The salt and hash cryptography attributes will be used to handle password encryption. Each password will have a unique hash to handle the encoding, which will provide a more secure profile. The update\_date attribute will record when the user modified a tag. The foreign key used in the User table will be group.id, which is the relation between the user table and the group table that will handle CURD tasks of the tags, users, and groups. The foreign key will connect to the Boolean values in the Group table and will handle the AND, OR, NOT or AND NOT values that are utilized in the search feature of the application.

### 3.2.2. User History

The User History table will have a primary key of id(int). This table will save modifications to tags and its responsibilities will include data per user based on when a tag was created, assigned, or deleted. The User History table will include the count(int) attribute, which will be in charge of keeping track of changes in the user history. This data table will include two foreign keys which are user.id(int) and tag.id(int). These foreign keys will be connected to the User table and the Tag table, respectively. The user.id foreign key will be used when a new username and password are created, and the table stores the user history table data. Additionally, each user's tag information will be related with the Tag table through the tag.id(int) foreign key.

### 3.2.3. File Tag

The primary responsibility of the File Tag table will be to handle the meta data that will be associated with any given tag. It will join with the Tag table, which then relates to the User and User History tables. The table will include four attributes, including one foreign key. The attributes will be file\_path(str), file\_name(str), file\_ext(str), and the tag.id(int) foreign key. The file\_path attribute will include the directory path to a tagged the file and the file\_name(str) will contain the name of the file. The file\_ext attribute will organize the extensions associated with the tagged file. The foreign key connects to the File Tag table to the Tag table.

### 3.2.4. Tag

The Tag table will contain id(int) primary key, which will connect to the File Tag table for specific tag information, name(str) and description(str), which will contain the name and detailed description of the tag. The Tag table has one primary key and no foreign key because this table will not be modified by other database objects, rather it will be used in relation to the User History and File Tag tables. Its primary relation will be to the File Tag table. The File Tag table will rely on the Tag table to transfer information regarding each tag, ie the file extension, the file name, or the file location in the directory. This table will also update and delete tags in 3NF format, so that any tags undergoing CURD tasks will not be removed entirely from the Tag table rather only removed from the joined table.

### 3.2.5. Group

The Group table will be primarily involved with the CURD roles. The primary key of the Group table is id(int). The Group table includes the name(str) attribute that will hold the name of created user groups. It will handle all the CURD modifications to tags, like updating the tags, reading, and deleting the tags in relation to each user. The table will handle these tasks using Boolean logic in order to make it more efficient when someone is trying to find specific information in the database. The Group table will handle the different user levels within the application. The application will include a regular user, a mid-level user and a high-level user and the Group table will confirm what access level a user has before authorizing changes to tags.



**PSEUDOCODE**

class Main()

'''

*The main controller, which initializes the app and calls other class constructors.*

- gui: obj

- https\_client: obj

- user: obj

- init(): no returns

+ create\_gui(): gui: obj

+ create\_https\_client(): https\_client: obj

+ create\_user(): user: obj

+ read\_gui(): gui: obj

+ read\_https\_client(): https\_client: obj

+ read\_user(): user: obj

'''

'''

*The methods (besides """init()""") are public so if an error occurs—or a user logs out and wants to log back in—they can be restarted individually.*

'''

function init()

'''

*Initialize the GUI, HTTPS server, and user on instantiation.*

*No returns*

'''

create\_gui()

create\_https\_client()

create\_user()

function create\_gui()

'''

*Create the app's GUI and update the associated private variable.*

*No returns*

'''

gui = new Gui()

function create\_https\_client()

'''

*Create the app's HTTPS client and update the associated private variable*.

*No returns*

'''

https\_client = new Https\_client ()

function create\_user ()

'''

*Create the app's user and update the associated private variable.*

*No returns*

'''

user = new User()

function read\_gui()

'''

*Read the memory location of the gui object.*

*Return gui: obj*

'''

return gui

function read\_https\_client ()

'''

*Read the memory location of the HTTPS client object.*

*Return https\_client: obj*

'''

return https\_client

function read\_user(gui: obj)

'''

*Read the memory location of the user object from the GUI.*

*Return user: obj*

'''

return user

class User()

'''

*The basic user, with methods shared by higher-privileged users.*

- id: int

- name: str

- init(name: str, password: str): no returns

- login(name: str, password: str): no returns

- create\_file\_tag(id: int, file\_path: str, file\_name: str, file\_ext: str; tag: obj): no returns

- delete\_file\_tag(id: int, file\_path: str, file\_name: str, file\_ext: str; tag: obj): no returns

- read\_file\_tags(id: int, file\_path: str, subdir: bool, tag: obj): no returns

- read\_history(id: int): no returns

- read\_id() id: int

'''

'''

*All display will be handled by the GUI; hence nothing needs to be returned directly by/to the user. The methods are private because the User class retains control of the main non-GUI thread, only passing control when necessary.*

'''

function init(name: str = None, password: str = None)

'''

*Initialize the user object, calling login().*

*No returns*

'''

if name AND password # Handle users who are created by an admin but are not actually logging in

return

else

name = prompt('Username: ')

password = prompt('Password: ').censor

login(name, password)

function login(name: str, password: str)

'''

*Log the user in with a given name and password, else destroy the user object.*

*No returns*

'''

client = main.read\_https\_client()

response = client.send\_request(login, (name, password))

if response = int # A user's ID (UUID) is always an int so this is an easy test to see if login was successful

id = response

else

self.destroy() # This isn't an app-defined method but should be part of the language

function create\_file\_tag(id: int, file\_path: str, file\_name: str, file\_ext: str; tag: obj)

'''

*Create a file tag on a given file using a valid tag object.*

*No returns*

'''

gui = main.read\_gui()

if id # All methods that require login require an ID, hence the check; without the check and some separate login system, a user could potentially call methods before their """login()""" function returned

gui.create\_file\_tag(file\_path, file\_name, file\_ext, tag)

function delete\_file\_tag(id: int, file\_path: str, file\_name: str, file\_ext: str; tag: obj)

'''

*Delete a file tag on a given file using a valid tag object.*

*No returns*

'''

gui = main.read\_gui()

if id

gui.delete\_file\_tag(file\_path, file\_name, file\_ext, tag)

function read\_file\_tags(id: int, file\_path: str, tag: obj)

'''

*Read the file tags in a given dir using a valid tag object.*

*No returns*

'''

gui = main.read\_gui()

if id

gui.read\_file\_tags(file\_path, tag)

function read\_history(id: int)

'''

*Read the user's history.*

*No returns*

'''

gui = main.read\_gui()

if id

gui.read\_history(int)

function read\_id()

'''

*Read the user ID from the user object.*

*Return id: int*

'''

return id

class Mid\_user(User)

'''

*The mid-level user, with additional methods.*

- create\_tag(id: int, name: str; description: str): no returns

- delete\_tag(id: int, tag: obj): no returns

'''

'''

All tag objects are handled by the GUI, too, hence nothing needs to be returned directly by/to the user.

'''

function create\_tag(id: int, name: str; description: str = 'NULL')

'''

*Create a tag object with a name and optional description.*

*No returns*

'''

gui = main.read\_gui()

if id

name = prompt('Tag name: ')

description = prompt('Tag description: ') OR 'NULL'

gui.create\_tag(name, description)

function delete\_tag(id: int, tag: obj)

'''

*Delete a valid tag object.*

*No returns*

'''

gui = main.read\_gui()

if id

gui.delete\_tag(tag)

class High\_user(Mid\_user)

'''

*The high-level user, with additional methods.*

- create\_user(id: int): no returns

- delete\_user(id: int, user: obj): no returns

- update\_user\_level(id: int, user: obj; level: str): no returns

'''

function create\_user(id: int)

'''

*Create a user with a given name and password.*

*No returns*

'''

if id

name = prompt('Username: ')

password = prompt('Password: ').censor

User(name, password)

function delete\_user(id: int, user: obj)

'''

*Delete a valid user object.*

*No returns*

'''

client = main.read\_https\_client()

if id

client.send\_request(delete\_user, (user))

function update\_user\_level(id: int, user: obj; level: str)

'''

*Update a valid user object up or down to a given level (class).*

*No returns*

'''

client = main.read\_https\_client()

if id

client.send\_request(update\_user\_level, (user, level))