IAM-Core Technical Specification

By Samer Masaad

Table of Contents

[1 Short Description 2](#_Toc507004220)

[2 Program Analysis 2](#_Toc507004221)

[2.1 Major Features 2](#_Toc507004222)

[2.1.1 Storing Medium 2](#_Toc507004223)

[2.1.2 Identity Management 2](#_Toc507004224)

[2.1.3 Logging 2](#_Toc507004225)

[2.1.4 Compatibility 2](#_Toc507004226)

[2.1.5 Password protected login 2](#_Toc507004227)

[2.2 Application Feasibility 2](#_Toc507004228)

[2.3 Data Description 3](#_Toc507004229)

[2.3.1 Exceptions 3](#_Toc507004230)

[2.4 Scope of the application 3](#_Toc507004231)

[2.4.1 Limitations 3](#_Toc507004232)

[2.4.2 Possible Future Features 3](#_Toc507004233)

[3 Conception 4](#_Toc507004234)

[3.1 Algorithms 4](#_Toc507004235)

[3.1.1 Creating an identity: 4](#_Toc507004236)

[3.1.2 Creating a user: 4](#_Toc507004237)

[3.1.3 Searching for user/identity: 4](#_Toc507004238)

[3.1.4 Update a user/identity: 4](#_Toc507004239)

[3.1.5 Delete a user/identity: 4](#_Toc507004240)

[3.2 Data Structure 5](#_Toc507004241)

[3.3 Application Hierarchy 7](#_Toc507004242)

[3.4 Application Tools 8](#_Toc507004243)

[3.4.1 Launcher 8](#_Toc507004244)

[3.4.2 Datamodels 8](#_Toc507004245)

[3.4.3 Configuration Service 8](#_Toc507004246)

[3.4.4 Connections 9](#_Toc507004247)

[3.4.5 Data Access Objects 9](#_Toc507004248)

[3.4.6 Logger 9](#_Toc507004249)

[4 Configuration Instructions 9](#_Toc507004250)

# Short Description

The IAM-Core Program is a Java identity management software that provide a way to create/manage identities and users as well as storing them in both Derby Database and XML file.

This guide is to describe the functionality of the tools.

Note: The iam-core.jar is not runnable. It’s just a set of methods and tools that can be used. The runnable version with GUI is iam-gui.jar.

# Program Analysis

## Major Features

### Storing Medium

Derby Database is used to store all the data related to the system. A backup medium, which is an XML file, is used if the connection to the database failed to be established. However, the program launches in read-only mode if one of the two mediums failed. This is done to keep the data synchronized between the database and the XML file.

### Identity Management

The program can create, search, update and delete identities as well as users.

### Logging

A logger is used to log all the events, information, warnings and errors which are occurring during the program run.

### Compatibility

All the properties are taken from a properties file that can be edited and filled to have the machine’s configurations (database credentials, logger and XML path, etc...). Thus, it can be compatible with any machine that runs Java.

### Password protected login

Only users are allowed to login to the program, their passwords are MD5-Hashed before being sent to the storing mediums.

## Application Feasibility

* In order for the program to work, JRE should be installed and properly working on the computer.
* A working Derby Database connection should also be working. However, if the schema and tables are not created, the software will create them after promoting the user.
* The JAR package has a built-in XML file that is used by default as a backup storing medium. However, if an XML file was provided through the configuration file, it will be used instead.

**Note:** For data integrity reasons, only the identity XML file can be provided from outside the JAR package. The built-in user XML will always be used.

* The default configuration file path is the same folder as the JAR’s and should be named: “conf.properties”. If the configuration file is not in the same folder, the path can be passed as a VM Variable while running the program from the command line. If the file is not found or can’t be used as a configuration file, the program doesn’t launch.

## Data Description

An identity has a Display Name, Email and a Unique ID, a user has a Username, Password and should be linked with an identity using the Identity ID. These fields can be accessed by the setters and getters from anywhere. Every user should have an associated identity, otherwise, an exception is thrown and should be handled by the GUI. The Display Name and Email for an identity can be String, all characters are allowed. Unique IDs and Usernames should not contain spaces and also can contain all kinds of characters.

### Exceptions

The fr.epita.iam.exceptions.DataException.java is the super class for:

* CreationException – thrown when an error occurs during creation of an identity/user
* DeleteException – thrown when an error occurs during deletion of an identity/user
* SearchException – thrown when an error occurs during searching for an identity/user
* UpdateException – thrown when an error occurs during update of an identity/user
* ReadOnlyException – thrown when trying to manipulate the data while in read-only mode
* NoIdentityFoundException – thrown when trying to create a user who doesn’t have an identity
* DuplicateException – thrown when a duplicate username or identity found

## Scope of the application

### Limitations

* While launching, the program checks if data on database and XML are identical, if not there is no way to synchronize yet.
* SQL Queries are hard-coded.
* Adding other data (such as address, phone, etc…) to the identities or users is not a possible option.
* Although a user can’t change another user’s info without knowing the password, all users have the same privileges. This means all users have same access rights and no users have privileges to reset passwords if forgotten.
* Checking if the user has the right to change another (or same) user’s information is the responsibility of the GUI classes. (Ex. asking for the password while changing the username of a user)

### Possible Future Features

* Synchronize between database and XML instead of entering read-only mode.
* Add user privileges, allowing users have more rights than others.
* Add a Map to the identities so an identity instance can have flexible data.

# Conception

## Algorithms

### Creating an identity:

1. Get identity information from user (Display name, Email and Unique ID).
2. Check if program running in read-only mode.
3. If yes, throw ReadOnlyException.
4. Check if Unique ID already exists.
5. If so, throw DuplicateException.
6. Create the identity in the database.
7. Create the identity in the XML file.

### Creating a user:

1. Get user information from user (Username, Password and Unique ID of the associated identity).
2. Check if program running in read-only mode.
3. If yes, throw ReadOnlyException.
4. Check if another user already exists and pointing to the same identity.
5. If so, throw DuplicateException.
6. Hash the password using MD5 algorithm.
7. Create the user in the database.
8. Create the user in the XML file.

### Searching for user/identity:

1. Get the criteria the user wants to search for. (can’t use a user’s password as a criteria)
2. If the database connection is working, use it.
3. Otherwise, search from the XML file.
4. Search the storing medium for the specific criteria.
5. Return a list of users/identities which match the criteria.

### Update a user/identity:

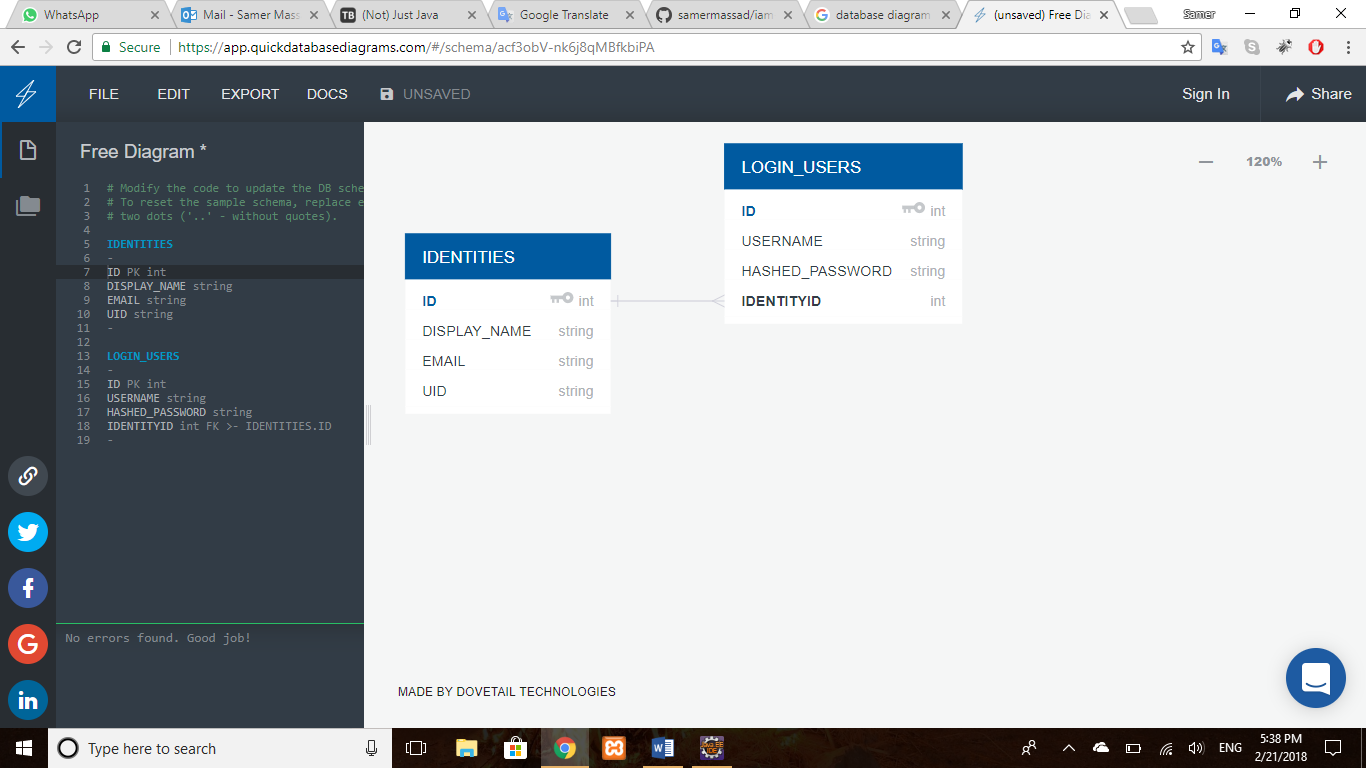
1. Get from the user the ID of the identity/user who needs to be updated, as well as the edited values. (Identity’s Unique IDs and user’s corresponding identities can’t be changed)
2. Check if program running in read-only mode.
3. If yes, throw ReadOnlyException.
4. If the username is changed, check if the new one already exists.
5. If so, throw DuplicateException.
6. If the password is changed, hash the password using MD5 algorithm.
7. Update the user in the database.
8. Update the user in the XML file.

### Delete a user/identity:

1. Get from the user the ID of the identity/user who needs to be deleted.
2. Check if program running in read-only mode.
3. If yes, throw ReadOnlyException.
4. If the password is changed, hash the password using MD5 algorithm.
5. Update the user in the database.
6. Update the user in the XML file.

## Data Structure

Data is stored in the database as following:



The IDENTITIES table stores the information of the identities. The Unique ID (UID) is used to provide a unique human-friendly IDs, while the primary key “ID” is used for providing links between the identities and their users.

The LOGIN\_USERS table stores the credentials of the users. The IDENTITYID field is used as a foreign key to link each user to his identity. This link prevents the deletion of an identity if it’s corresponding user is still there in the LOGIN\_USERS table.

The same data is also stored in the backup medium; the XML file. The files should have this tags in order to be successfully parsed and used by the program:

The identity XML:

<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<identities>

<identity>

<property name="displayName">identity1</property>

<property name="uid">1</property>

<property name="email">email@email.com</property>

</identity>

<property name="displayName">identity2</property>

<property name="uid">FR207</property>

<property name="email">email@email.com</property>

</identity>

…

…

…

</identity>

<identities>

The user XML:

<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<users>

<user>

<property name="username">user1</property>

<property name="hashedPassword">{hashedpassword}</property>

<property name="identityID">1</property>

</user>

<property name="username">user2</property>

<property name="hashedPassword">">{hashedpassword}</property>

<property name="identityID">FR207</property>

</user>

…

…

…

</user>

<users>

## Application Hierarchy

**iam-core**

* fr.epita.iam.launcher
  + Global.java
  + PrelaunchTests.java
* fr.epita.iam.datamodel
  + Identity.java
  + User.java
* fr.epita.iam.services.configuration
  + ConfigurationService.java
* fr.epita.iam.services.connections
  + JDBCConnection.java
  + XMLConnection.java
* fr.epita.iam.services.identity.dao
  + IdentityDAO.java
  + IdentityDAOManager.java
  + IdentityJDBCDAO.java
  + IdentityXMLDAO.java
* fr.epita.iam.services.user.dao
  + UserDAO.java
  + UserDAOManager.java
  + UserJDVCDAO.java
  + UserXMLDAO.java
* fr.epita.iam.exceptions
  + DataException.java
  + CreationException.java
  + DeleteException.java
  + SearchException.java
  + UpdateException.java
  + ReadOnlyException.java
  + NoIdentityFoundException.java
  + DuplicateException.java

**logger**

* fr.epita.logger
  + Logger.java

**iam-gui**

* fr.epita.iam.gui
  + Main.java
  + GUIMethods.java
  + Login.java
  + Home.java
  + IdentityAdd.java
  + IdentitySearch.java
  + IdentityEdit.java
  + UserAdd.java
  + UserSearch.java
  + UserEdit.java

## Application Tools

### Launcher

The PrelaunchTests.java provides a set of tests that a GUI can use to determine the behavior of the application. The results of the tests are stored in the Global.java and can be retrieved anytime.

PrelaunchTests tests = **new** PrelaunchTests();

**if**(tests.run()) {

//program can launch

**if**(Global.*isReadOnly*()) {

//program can be launched in read-only mode

}

**if**(Global.*isDBWorking*()) {

//database is working

}

**if**(Global.*isXMLWorking*()) {

//xml parsed successfully

}

} **else**

//program can’t launch

**return**;

tests.run() first checks the configuration file, if not found or had syntax error the method returns false. Then the connection with the database is tested, the status is then stored in the **private** **static** **boolean** *DB* which can be retrieved by calling Global.*isDBWorking*(). Similarly, the XML parsing status is stored in **private** **static** **boolean** *XML* which can be retrieved by calling Global.*isXMLWorking*().

### Datamodels

Used to create instances of identities and users.

Identity id1 = **new** Identity("identity1","email","uid");

User user1 = **new** User("username","password",1);

System.***out***.println(id1.getDisplayName()); //identity1

System.***out***.println(user1.getHashedPassword()); //hashed value of “password”

### Configuration Service

Provides an instance to get properties from file.

ConfigurationService confService = ConfigurationService.*getInstance*();

String host = confService.getConfigurationValue("db.host");

String password = confService.getConfigurationValue("db.password");

String username = confService.getConfigurationValue("db.username");

### Connections

Initializes connection to database or gets the parsed document of XML.

Connection connection = JDBCConnection.*getConnection*();

JDBCConnection.*close*(connection, preparedStatement, rs);

Documentdocument = XMLConnection.*getIdentityXML*();

XMLConnection.*saveIdentityXML*(document);

JDBCConnection.*getConnection*() gets the host, username and password from the ConfigurationService and creates a connection to the database using the JDBC and the Derby driver and returns it.

JDBCConnection.*close*(Connection c, PreparedStatement ps, ResultSet rs) closes each parameter if not null.

XMLConnection.*getIdentityXML*() parses and returns the XML file specified in the properties file.

XMLConnection.*saveIdentityXML*(Document document) outputs the new document overriding the previous one.

### Data Access Objects

Provide instances to deal with managing the identities and users.

User DAOs and identity DAOs are separate due to the huge differences between the User class and the Identity Class and their representation in the database.

It is possible to deal with the JDBC and XML DAOs directly, but a manager is provided that manages the work of both ways and run some very important tests before manipulating the database. All the DAOs share the same behavior, so all of them implement an interface.

### Logger

Logs info, errors and warning to a file whose path is provided from the configuration file. However, if the path is not provided, the default path for the logger is the home directory of the OS user, a folder called .iam-core will be created in:

**In windows:** %Homedrive%%Homepath%/.iam-core/application.log

**In Linux:** $HOME/.iam-core/application.log

**In macOS:** $HOME/Documents/.iam-core/application.log

# Configuration Instructions

1. To prepare the database, run the cleaner and initializer SQL files provided with the JAR package.
2. If you are providing a custom XML, make sure it’s well formed and doesn’t have any entries different from the database’s.
3. Place the configuration file next to the JAR file and name it “conf.properties”. Otherwise, you should provide the path while running the runnable program from the command line:

java -Dconf=${pathToConfigFile} -jar ${jarName}

1. If you’re running the program for the first time, the queries in init.sql will create a user with

Username: root

Password: root

After login, it’s recommended to delete the root user and create new users.