IAM-Core User Guide

By Same Masaad

This guide is to describe how to develop an application (probably a GUI) that uses identity management tools provided by the IAM-Core.

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.

Table of Cosntents

[1 Introduction 2](#_Toc507019518)

[1.1 Rules and information 2](#_Toc507019519)

[1.2 What the software doesn’t do 2](#_Toc507019520)

[2 Configuration Instructions 2](#_Toc507019521)

[3 Provided Tools 2](#_Toc507019522)

[3.1 Prelaunch Tests 2](#_Toc507019523)

[3.2 Datamodels 3](#_Toc507019524)

[3.3 Configuration Service 3](#_Toc507019525)

[3.4 Connections 4](#_Toc507019526)

[3.5 Data Access Objects 4](#_Toc507019527)

[3.6 Logger 4](#_Toc507019528)

# Introduction

Iam-core needs a connection with a Derby Database server to function. It also uses a built-in XML file to store the same data as a backup storing medium.

## Rules and information

* An identity can have a Display Name, Email and a Unique ID.
* A user is needed to login to the program, and can have a username, password and should be associated with an identity.
* It’s not possible to create a user that is not linked to an existing identity.
* Trying to delete an identity will automatically delete its user (if exists).
* If one storing method failed to be established, the program launches in read-only mode.
* Instead of using the built-in XML file for storing identities, a path to a custom XML file can be passed while running the JAR file. However, the built-in users’ XML will always be used.
* A configuration file must be filled with the right information and placed next to the JAR file.

## What the software doesn’t do

* All the prelaunch tests are provided, but not ran automatically.
* It’s the GUI’s responsibility to make sure that using the software is login-protected.
* If trying to create a user whose identity doesn’t exist, an exception is thrown, and the user won’t be created.
* If trying to manipulate the data while in read-only mode, an exception is thrown.

# Configuration Instructions

1. To prepare the database, manual creation of a database is required. Then, the initializer SQL file provided with the JAR package (init.sql) should be ran. The queries inside this file will create the required schema and tables and will add a default identity and user called root.
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 2 provided XML files inside a resources folder in your project called “resources”. This allows to use the built-in XML files (Note that it is mandatory to put at least the users\_db.xml)
4. Fill and place the configuration file next to the JAR file and name it “conf.properties”. Otherwise, you should provide the path while running the program from the command line:

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

Please use the provided configuration file, it has comments that describe how to fill it.

1. Add the iam-core.jar to the build path of your project in order to be able to use its tools.

# Provided Tools

## Prelaunch Tests

A set of tests are provided and recommended to be used in order to define the working logic of the program. To use them, an instance of PrelaunchTests should be initialized, when the run() method of the instance is called, the results of the tests are stored in the Global class.

Code example:

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**;

The run() method tests the configuration file, JDBC connection to the database and try to parse the XML file.

## Datamodels

Used to create instances of identities and users. Creating and setting information is done either by passing all the information while initializing, or using the setters.

//method1

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

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

//method2

Identity id2 = **new** Identity();

id2.setDisplayName("identity2");

id2.setEmail("email");

id2.setUid("FR100");

User user2 = **new** User();

user2.setUserName("user2");

user2.setPassword("password");

user2.setIdentityID(1);

Retrieving the information is done using the getters:

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

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

Note: the integer passed to setIdentityID(**int** identityID) is not the Unique ID of the identity. It is the raw ID which is generated automatically while adding a new identity. We can get it by calling getId(String UID) method provided by the UserJDBCDAO class.

Example: user2.setIdentityID(dao.getId("FR100"));

## 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.

Connectionconnection;

ResultSetrs;

**try** {

connection = JDBCConnection.*getConnection*();

preparedStatement = connection.prepareStatement("SELECT \* FROM IDENTITIES");

rs = preparedStatement.execute();

} **catch** (Exception e) {

//handle exception

} **finally** {

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

}

Document document;

**try** {

document = XMLConnection.*getIdentityXML*();

//manipulate the document

} **catch** (ParserConfigurationException | SAXException | IOException e) {

//handle exception

}

XMLConnection.*saveIdentityXML*(document);

## Data Access Objects

Provide instances to deal with managing the identities and users.

IdentityDAOManager identityDAO = **new** IdentityDAOManager();

UserDAOManager userDAO = **new** UserDAOManager();

identityDAO.create(**new** Identity("identity1", "email", "FR201"));

userDAO.update(

**new** User(**null**, **null**, userDAO.getId("FR201")),

**new** User("user1", "password", userDAO.getId("FR201"))

);

identityDAO.delete(**new** Identity(**null**, **null**, "FR201"));

//will throw DeleteException because user1 is still there

## 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

The class where the logger is working should be passed while initializing in order to log the class that created the log entry.

Logger ***LOGGER*** = **new** Logger(Main.**class**);

***LOGGER***.warning("Running in read-only mode!");