4.1.9 Package analysis: utils

This package contains utility classes.

|  |  |
| --- | --- |
| Class Name | Short Description |
| FormValidatorPokeMongo | Used for check if a field is well filled |
| LoggerThread | A thread that writes information about all the action taken by the code. |
| Logger | Public logger that use the Thread one for handling all the logs. |

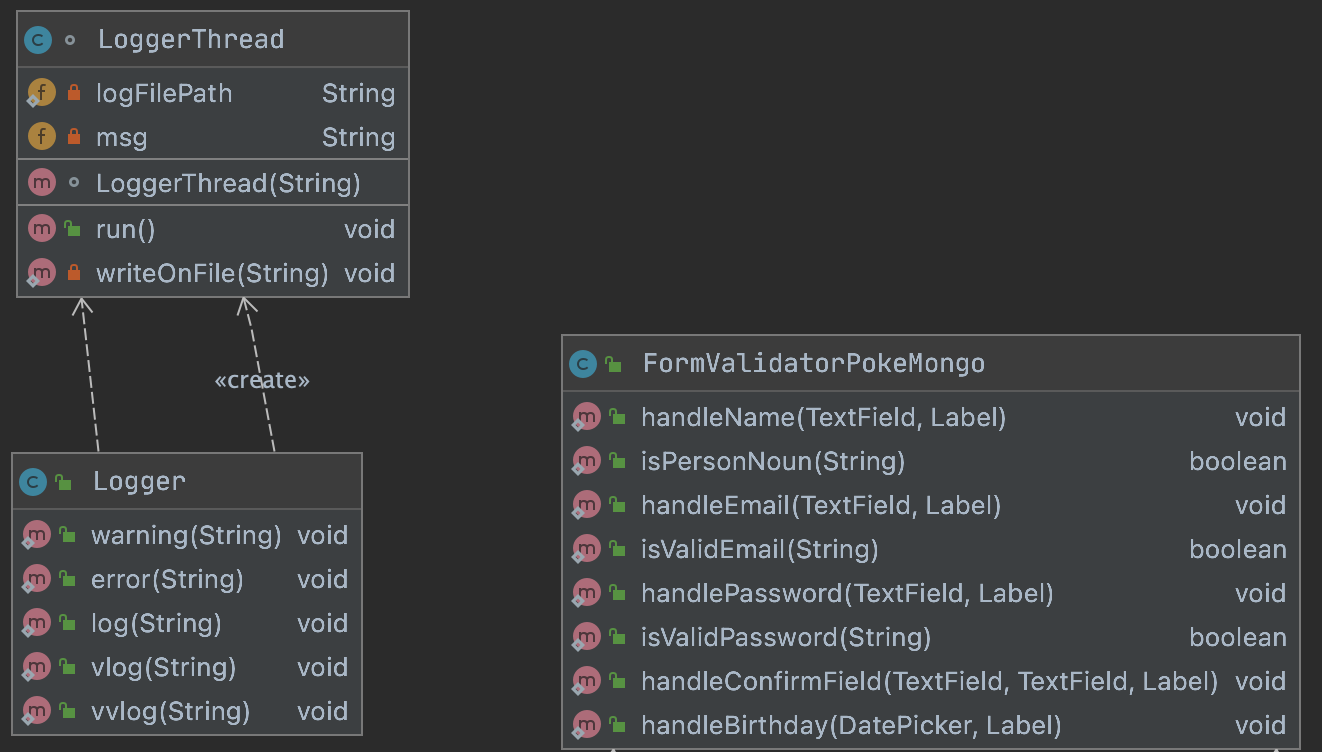
Inside this package we added other two packages useful for the implementation of the initial data, thus they are not needed to the user that uses the application (in a possible publish this part will be erased).

The **autoLog** packages contains a java file named *AutoLogin*:

public class AutoLogIn {

public static void main(String[] args) {

// TAKE ALL THE USERS

UserManager userManager = UserManagerFactory.*buildManager*();

ArrayList<User> allUsers = userManager.getEveryUser();

for (User user: allUsers) {

if (Math.*random*() < .7)

continue;

userManager.login(user.getUsername(),user.getName() + user.getSurname() + "000");

user.addTeam(*retrieveTeam*(user));

// Update the point in mongodb

userManager.updatePoints(user, user.getPoints());

}

}

private static Pokemon[] retrieveTeam(User user) {

TeamManager teamManager = TeamManagerFactory.*buildManager*();

return teamManager.getUserTeam(user);

}

}

The main method retrieves all the user from the db and for each of them calculate a value using Math.random(). If this value is less than 0.7 the for loop continues with the next iteration (this is not for waiting not too many hours). If the if turn out to be true, we login in in the user account, we retrieve his/her **Team** and then we calculate and update the points.

This class comes in handy for generating some useful data by our own.

The dbPopulators package contains a java file named UserPopulator. This class has a main method that creates randomly users to be added in MongoDb (as described in chapter 2.3). What it does is picking a random name, surname and country from the related txt files and some other random information that are required for registering and instantiates a new User object, that is stored in the db by a specific database method.

4.2 API & SPI

In the package named *persistence* we, for the sake of code obfuscation (as told 4.1.1), created a API/SPI relation. The main Service Provider Interface is the class Database that defines remote connections and structures of basic CRUD operations. Here a look up:

public interface Database {

*/\*\**

\* starts the connection with the remote Database

\*/

void startConnection();

*/\*\**

\* closes the connection with the remote Database

\*/

void closeConnection();

*/\*\**

\* insert 1 ore more objects into the database

\* **@param** toInsert ArrayList of arbitrary objects to insert into the Database

\* **@return** true if at least one element has been added

\*/

boolean insert(ArrayList<Object> toInsert);

*/\*\**

\* insert 1 object into the database

\* **@param** toInsert ArrayList of arbitrary objects to insert into the Database

\* **@return** true if exactly one element has been added

\*/

boolean insert(Object toInsert);

*/\*\**

\* remove 1 ore more objects from the database

\* **@param** o is an Object to remove or a query to submit

\* **@return** true if at least one element has been removed

\*/

boolean remove(Object o);

*/\*\**

\* gets all the elements

\* **@return** a list of retrieved elements

\*/

ArrayList<Object> getAll();

*/\*\**

\*

\* **@param** filter query or filter to submit to the db

\* **@return** a list of retrieved elements

\*/

ArrayList<Object> getWithFilter(Object filter);

*/\*\**

\*

\* **@param** target query or object to update

\* **@param** newValue new value for the target

\* **@return** true if at least one object is updated

\*/

boolean update(Object target, Object newValue);

}

From this interface we created different implementations, one for every database used, which are Neo4jDbDatabase and MongoDbDatabase. Although, this two classes are abstract, thus they are extended for creating query specific private classes (e.g., PokemonManagerOnMongoDb handles all the query related to Pokémon on MongoDb).

For making use of those private classes, we have to use some APIs called Factory. We have a factory for each query specific class type:

|  |  |
| --- | --- |
| Query specific type | Factory |
| Managing pokemon | PokemonManagerFactory |
| Managing posts | PostManagerFactory |
| Managing team | TeamManagerFactory |
| Managing user information | UserManagerFactory |
| Managing the network between users | UserNetworkManagerFactory |

Every query specific type has also an own interface that works as SPI.

The structure of a factory class is the following, here the PokemonManagerFactory:

public class PokemonManagerFactory {

public static PokemonManager buildManager(){

String technology = *getConfiguration*();

Logger.*vlog*("Obtaining technology: " + technology);

switch (technology){

case "MongoDb":

return new PokemonManagerOnMongoDb();

default:

try{

throw new IllegalArgumentException();

}catch (IllegalArgumentException iae){

iae.printStackTrace();

Logger.*error*("Invalid database technology or missing implementation");

};

return null;

}

}

public static String getConfiguration(){

return ConfigDataHandler.*getInstance*().configData.pokemonDbArchitecture;

}

}

What it actually does is checking the configuration technology, retrieved that, it switches in order to use the one requested. If it’s one of the presented in the case period then it will retrieve the appropriate class object, otherwise it will throw an exception.