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. |

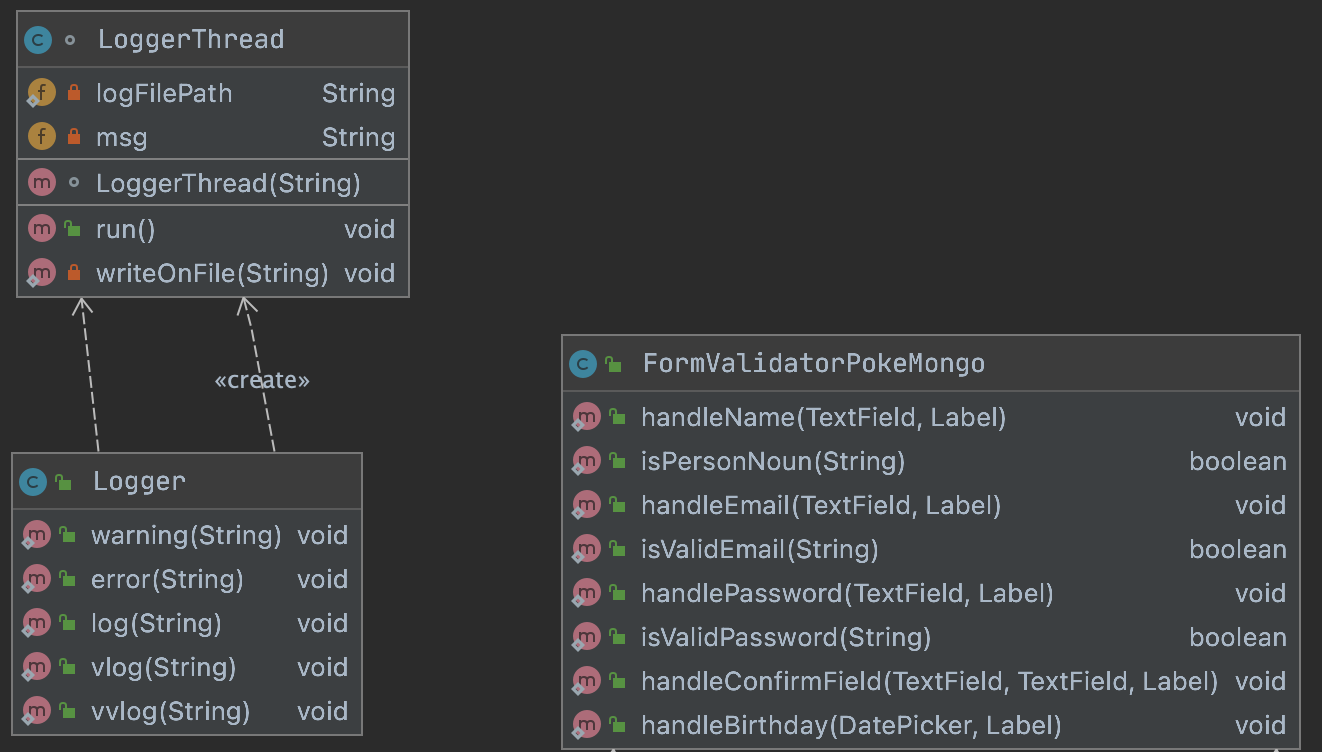
4.1.10 Obfuscation

Our package structure organization gave us the possibility to exploit code obfuscation. We use code obfuscation in the way to hide how the connection of the database is done. To do that we limited some classes to have only a package scope and, to interact with them, we use the Manager classes presented before.

4.2 Main tools

For enhancing our performances and for giving to the user a better application, some tool are used. We focus in this chapter about: GSON, caching system, password encrypt and the logger.

4.2.1 GSON

Gson is a Java library that can be used to convert Java Objects into their JSON representation. It can also be used to convert a JSON string to an equivalent Java object. Gson can work with arbitrary Java objects including pre-existing objects that you do not have source-code of.

We use Gson mainly for communicating with MongoDb and for setting the configuration information.

4.2.2 Caching mechanism and multimedia management

The Pokemon game series became famous for the creature a player can capture and use, so we thought to include the Pokemon images also in our project.

The images about the Pokemons are not stored locally, because this will increase our project size too much. To avoid this huge increment we take all the images, only when they are needed, from a GitHub repository. Here a problem came: loading a lot of images would slow down our system and that is against our no-functional requirements. The caching comes then in handy.

public class PokeMongoImageCache implements PokeMongoCache {

//Singleton

private static PokeMongoImageCache *instance*;

private AsyncLoadingCache<String, Image> cache;

public static PokeMongoImageCache getInstance() {

if (*instance* == null) {

*instance* = new PokeMongoImageCache();

}

return *instance*;

}

PokeMongoImageCache(){

cache = Caffeine.*newBuilder*()

.expireAfterAccess(10, TimeUnit.*MINUTES*) //After this time without read/write the resource is deallocated

.maximumSize(1000) //The number of images

.buildAsync(k -> PokemonImage.*get*(k));

}

public CompletableFuture<Image> getDataIfPresent(String url){

Logger.*vlog*("Attemp to get image at: " + url);

return cache.get(url);

}

}

What the cache does is simply store asynchronously an image. Why asynchronous? If the operation would be done in a synchronous way the user has to wait that the image is properly store for seeing and using the UI. Using an asynchronous way the user can interact with the UI even if the images are already loaded, this create a better application usage.

4.2.3 Password Encryptor

Encrypting a password is typically used to protect it from eavesdropping. We encrypt it and then we sent it to the database, in this way no eavesdropper can snatch the password in transit.

How does the encryption is done? We decided to use the class org.apache.commons.codec.digest.DigestUtils, that has operations to simplify common MessageDigest tasks. Thus, we use DigestUtils.*sha256Hex()* calculate the SHA-256 digest of a string composed by the password of the user plus one other string.

public class PasswordEncryptor {

@VisibleForTesting

public static String encryptPassword(String plainPassword){

String s = "randomSalt";

String encryptedPassword = *cipher*(plainPassword, s);

Logger.*vlog*("ENCRYPTION | encrypt-pw: " + encryptedPassword);

return encryptedPassword;

}

public static String cipher(String pwd, String salt) {

return DigestUtils.*sha256Hex*(pwd+salt);

}

}

4.2.4 Logger

We thought that useful logs would provide us (especially when someone has to debug/maintain someone else’s code) with help when trying to understand what the code actually does. The Logger takes care of recording the events that happen during runtime.

public class Logger {

public static void warning(String text){

if(ConfigDataHandler.*getInstance*().configData.verbosityLevel >= 1){

LoggerThread lt = new LoggerThread("[WARNING] " + text);

lt.start();

}

}

public static void error(String text){

if(ConfigDataHandler.*getInstance*().configData.verbosityLevel >= 1){

LoggerThread lt = new LoggerThread("[ERROR] " + text);

lt.start();

}

}

public static void log(String text){

if(ConfigDataHandler.*getInstance*().configData.verbosityLevel >= 1){

LoggerThread lt = new LoggerThread("[LOG] " + text);

lt.start();

}

}

public static void vlog(String text){

if(ConfigDataHandler.*getInstance*().configData.verbosityLevel >= 2){

LoggerThread lt = new LoggerThread("[VLOG] " + text);

lt.start();

}

}

public static void vvlog(String text){

if(ConfigDataHandler.*getInstance*().configData.verbosityLevel >= 3){

LoggerThread lt = new LoggerThread("[VVLOG] " + text);

lt.start();

}

}

}

Every function create and start a new LoggerThread, it will simply print in a file the log infos.

class LoggerThread extends Thread{

private static String *logFilePath*="log/logFile.txt";

private String msg;

LoggerThread(String msg){

this.msg=msg;

}

public void run(){

String newMsg = Instant.*now*().toString() + " " + msg + "\n";

writeOnFile(newMsg);

}

private static synchronized void writeOnFile(String msg){

try {

Files.*write*(Paths.*get*(*logFilePath*), msg.getBytes(), StandardOpenOption.*APPEND*);

}

catch (IOException i){

i.printStackTrace();

}

}

}

4.2.5 Form Validator

The FormValidatorpokMongo checks if the user fills the textfields in the proper way, if he doesn’t that it show up an error as a label. The validator functions makes use of Regular Expression, in this way me ensure that the field is well written.

public class FormValidatorPokeMongo {

*/\*\**

\* In this section are present the event handler for the 'setOnKeyReleased' event in the form.

\*/

public static void handleName(TextField nameTF, Label invalidNameLabel){

if(FormValidatorPokeMongo.*isPersonNoun*(nameTF.getText()))

invalidNameLabel.setVisible(false);

else

invalidNameLabel.setVisible(true);

}

*/\*\**

\* Check if the string contains only letters, spaces, dots and apostrophes.

\*/

@VisibleForTesting

public static boolean isPersonNoun(String possibleNoun){

Pattern pattern = Pattern.*compile*("^[a**-**zA**-**Z '.]+$");

Matcher matcher = pattern.matcher(possibleNoun);

//Logger.vvlog("isPersonNoun() -> " + possibleNoun + (matcher.find() ? "-> V" : "-> X"));

return matcher.find();

}

public static void handleEmail(TextField emailTF, Label invalidEmailLabel){

if(FormValidatorPokeMongo.*isValidEmail*(emailTF.getText()))

invalidEmailLabel.setVisible(false);

else

invalidEmailLabel.setVisible(true);

}

*/\*\**

\* Check if the email follows the format example@domain.tld

\*/

@VisibleForTesting

public static boolean isValidEmail(String possibleEmail){

Pattern pattern = Pattern.*compile*("^[\\w-\\.]+@([\\w-]+\\.)+[\\w-]{2,4}$");

Matcher matcher = pattern.matcher(possibleEmail);

//Logger.vvlog("isValidEmail() -> " + possibleEmail + (matcher.find() ? "-> V" : "-> X"));

return matcher.find();

}

public static void handlePassword(TextField passwordTF, Label invalidPasswordLabel){

if(FormValidatorPokeMongo.*isValidPassword*(passwordTF.getText()))

invalidPasswordLabel.setVisible(false);

else

invalidPasswordLabel.setVisible(true);

}

*/\*\**

\* Checks if the password contains minimum eight characters, at least one letter and one number.

\*/

@VisibleForTesting

public static boolean isValidPassword(String possiblePassword){

Pattern pattern = Pattern.*compile*("^(?=.\*[A**-**Za**-**z])(?=.\*\\d)[A**-**Za**-**z\\d]{8,}$");

Matcher matcher = pattern.matcher(possiblePassword);

//Logger.vvlog("isValidPassword() -> " + possiblePassword + (matcher.find() ? "-> V" : "-> X"));

return matcher.find();

}

public static void handleConfirmField(TextField fieldTF, TextField confirmFieldTF, Label invalidConfirmFieldLabel){

String password = fieldTF.getText(), confirmPassword = confirmFieldTF.getText();

if(password.equals(confirmPassword))

invalidConfirmFieldLabel.setVisible(false);

else

invalidConfirmFieldLabel.setVisible(true);

}

*/\*\**

\* Checks if the birthday date selected is valid: future dates cannot be picked

\*/

public static void handleBirthday(DatePicker birthdayDP, Label invalidBirthdayLabel){

LocalDate localDate = birthdayDP.getValue();

LocalDate today = LocalDate.*now*();

System.*out*.println(today);

if(localDate.isAfter(today)){

invalidBirthdayLabel.setVisible(true);

} else {

invalidBirthdayLabel.setVisible(false);

}

}

}

4.3 Business logic

The business logic used in PokeMongo is the following. To ensure a high variability between Teams we decided to cope the catch rate of each Pokemon in a way that if it is held by a lot of user it will decrease, thus will let users to try to catch other Pokemon, with the same rarity but held by less people. Although, the catch rate is related also to the points, if a Pokemon has a lower catch rate the points are higher, so catching other Pokemon (not the famous ones) may look like illogical at first, but keep in mind that the probability that having a rare Pokemon is very low as the “CATCH’EM ALL” page, so it is quite impossible that every user has the same rare Pokemons.

4.3.1 Points computing

The points related to a Pokemon are computed in a very simple way and they are strictly linked to the catch rate of a Pokemon (a value between 3 and 255).

The total points held by a user is computed as follow:

The multiplier is a value of 1.5 that is applied when in the team are presented all Pokemon that has different types from each other, otherwise the multiplier will be 1.

The total points is computed only when a user login or when he catches/remove a Pokemon.

4.3.2 Dynamic Catch Rate computing

The dynamic catch rate is expatiate as follow:

4.4 Analytics queries

In this chapter are present how the analytic are really compute in our system.

4.4.1 User Rankings

We have three different types of ranking that regards the users: the World Best Team, the Friend Best Team and the “Country” Ranking (where country is a specific country, e.g. Italy).

4.4.1.1 World Best Team

The World Best Team consider all the user in the world and retrieve a limited number of user who has the highest value of points. This is done querying MongoDb as follow (function present in UserManagerOnMongoDB):

public List<User> bestWorldTeams() {

Logger.*vlog*("COMPUTING BEST WORLD TEAMS");

Bson match = *match*(*and*(*eq*("admin", false), *lte*("lastLogin", getDateThresholdForRanking())));

Bson sort = *sort*(*descending*("points", "birthDay"));

Bson limit = *limit*(ConfigDataHandler.*getInstance*().configData.numRowsRanking);

Bson project = *project*(*fields*(*excludeId*(), *include*("username", "teamName", "points", "birthDay", "country")));

return aggregate(Arrays.*asList*(match, sort, limit, project));

}

What is done is matching users that aren’t admin and that have the “lastLogin” field within a certain range of days. Then we order the result in a descending way, considering the points and the birth day, this one just to order users that have the same value for points in order to greeting younger players. As said before we limit the number of result (this can be modified in the configuration file) and we project only the feature we are interested into.

We have limited the ranking based on “lastLogin”, because the user points are only updated when the user use the application, in order to have a less computational effort in the server databases. This, also, makes the user points something that lose meaning after a certain periodo of time.

4.4.1.2 Friend Best Team

This kind of ranking is made thanks to the combination of MongoDb and Neo4j, because friends are retrieve from Neo4j and all information about them are, then, retrieve from MongoDb. To do that in our code we use two function, one from UserNetworkManagerOnNeo4j

public List<String> getFollowersUsernames(User target){

List<String> followersUsernames = new ArrayList<String>();

String query = "MATCH (to:User)-[h:FOLLOW]->(from:User) WHERE from.username = $username RETURN to.username";

ArrayList<Object> res = getWithFilter(query, *parameters*("username", target.getUsername()));

for(Object o: res){

Record r =(Record)o;

String username = r.get("to.username").asString();

followersUsernames.add(username);

}

return followersUsernames;

}

And the other from UserManagerOnMongoDb

public List<User> bestFriendsTeams(List<String> friendsUsername) {

Logger.*vlog*("COMPUTING BEST FRIENDS TEAMS");

Bson sort = *sort*(*descending*("points", "birthDay"));

Bson limit = limit(ConfigDataHandler.getInstance().configData.numRowsRanking);

Bson match = match(and(eq("admin", false), in("username", friendsUsername), lte("lastLogin", getDateThresholdForRanking())));

Bson project = *project*(*fields*(*excludeId*(), *include*("username", "teamName", "points", "birthDay", "country")));

return aggregate(Arrays.asList(match, sort, limit, project));

}

4.4.1.3 Country Best Team

This is similar to the World Best Team, but we use an additional match for the country.

public List<User> bestCountryTeams(String country) {

Logger.*vlog*("COMPUTING BEST TEAMS FOR COUNTRY " + country);

Bson match = *match*(*and*(*eq*("country", country), *eq*("admin", false), *lte*("lastLogin", getDateThresholdForRanking())));

Bson sort = *sort*(*descending*("points", "birthDay"));

Bson project = *project*(*fields*(*excludeId*(), *include*("username", "teamName", "points", "birthDay", "country")));

Bson limit = *limit*(ConfigDataHandler.*getInstance*().configData.numRowsRanking);

return aggregate(Arrays.*asList*(match, sort, limit, project));

}

4.4.2 Pokémon Rankings

We only perform two types of ranking regarding the Pokemons, the World Best Pokemon and “Country” Best Pokemon.

4.4.2.1 World Best Pokemon

Most used Pokemons in the whole world. We compute them using Neo4j (TeamManagerOnNeo4j):

public ArrayList<Pokemon> getBestPokemon() {

ArrayList<Pokemon> pokemonArrayList = new ArrayList<>();

String query = "MATCH ()-[h:HAS]->(p:Pokemon) return p.name, count(h) AS held, p.sprite ORDER BY held DESC LIMIT " + + ConfigDataHandler.*getInstance*().configData.numRowsRanking;

ArrayList<Object> res = getWithFilter(query);

return getPokemons(pokemonArrayList, res);

}

4.4.2.2 Country Best Pokemon

Most used Pokemons in a specific country. We compute them using Neo4j (TeamManagerOnNeo4j)

public ArrayList<Pokemon> getBestPokemon(String country) {

ArrayList<Pokemon> pokemonArrayList = new ArrayList<>();

String query = "MATCH (u:User)-[h:HAS]->(p:Pokemon) WHERE u.country = $country return p.name, count(h) AS held, p.sprite ORDER BY held DESC LIMIT " + ConfigDataHandler.*getInstance*().configData.numRowsRanking;

ArrayList<Object> res = getWithFilter(query, *parameters*("country", country));

return getPokemons(pokemonArrayList, res);

}

4.4.3 Usage Statistics

We created few function for handling some simple statistics that can be useful for the admin user, in order to see how many the people in the world use the application and how often. We decided to include three types of statistic: total number of users, users that logged during a single day, users that logged during a single day in a specific country (we limit the number of country to the top 15). Because all the information needed is stored in MongoDb we query the database in order to create a new document with all the information specific for the analytics of a single day.

4..4.3.1 Total User

public long getUserNumber() {

Bson match = *match*(*ne*("admin", true));

Bson count = *group*("$admin", *sum*("userNumber", 1));

Bson project = *project*(*fields*(*include*("userNumber")));

Document result = aggregate(Arrays.*asList*(match, count, project)).get(0);

return result.getInteger("userNumber").longValue();

}

4.4.3.2 Today login

public long getTodayLogin(){

Calendar lastDay = Calendar.*getInstance*();

lastDay.setTime(new Date());

lastDay.add(Calendar.*DATE*, -1);

String yesterday = new SimpleDateFormat ("yyyy-MM-dd'T'HH:mm:ss.SSS'Z'", Locale.*US*).format(lastDay.getTime());

Bson match = *match*(*and*(*gte*("lastLogin", yesterday), *ne*("admin", true)));

Bson count = *group*("$admin", *sum*("loginNumber", 1));

Bson project = *project*(*fields*(*include*("loginNumber")));

Document result = aggregate(Arrays.*asList*(match, count, project)).get(0);

return result.getInteger("loginNumber").longValue();

}

4.4.3.3 Country Today Login

The same thing of the “Total Login” but considering countries and limiting the results as said above. This is done for not storing useless data.

public Map<String, Long> getLastLoginsByCountry() {

Calendar lastDay = Calendar.*getInstance*();

lastDay.setTime(new Date());

lastDay.add(Calendar.*DATE*, -1);

String yesterday = new SimpleDateFormat ("yyyy-MM-dd'T'HH:mm:ss.SSS'Z'", Locale.*US*).format(lastDay.getTime());

Bson match = *match*(*and*(*gte*("lastLogin", yesterday), *ne*("admin", true)));

Bson count = *group*("$country", *sum*("lastLogin", 1));

Bson sort = *sort*(*descending*("lastLogin"));

Bson limit = *limit*(15);

Bson project = *project*(*fields*( *include*( "\_id", "lastLogin")));

List<Document> result = aggregate(Arrays.*asList*(match, count, sort, limit, project));

Map<String, Long> map = new HashMap<>();

for(Document d: result)

map.put(d.getString("\_id"), d.getInteger("lastLogin").longValue());

return map;

}

4.4.4 Dynamic Catch Rate

The dynamic catch rate is saved in the Pokemon json as an array of 30 values, in which the first one indicates the current catch rate of the Pokemon (the other 29 values are using for plotting the catch rate chart). Let’s analyze it step by step.

The first thing we have to do is to retrieve from the databases all the Pokemon.

PokemonManager pokemonManager = PokemonManagerFactory.*buildManager*();

TeamManager teamManager = TeamManagerFactory.*buildManager*();

ArrayList<Pokemon> pokemons = pokemonManager.getEveryPokemon();

Then for each of them we retrieve how many trainers has it.

List<Pair<String, Integer>> trainersPerPokemon = teamManager.getUsersNumberThatOwnsAPokemonNotFiltered();

Next step is to compute the catch rate as describe in chapter 4.3.2 for every Pokemon.

List<PokemonAndCatchRate> catchRatesOfPokemons = new ArrayList<>();

int index = 0;

int numTrainers;

double new\_catch\_rate;

Pokemon oldPokemon;

List<Double> capture\_rates;

for(Pokemon p: pokemons){

oldPokemon = new Pokemon(p.getName(), p.getTypes(), p.getId(), p.getCapture\_rate(), p.getCapture\_rates(), (int)p.getHeight(), (int)p.getWeight(), p.getBiology(), p.getPortrait(), p.getSprite());

Pair<String, Integer> currentTrainers = trainersPerPokemon.get(index);;

if(trainersPerPokemon.get(index).getKey().equals(p.getName())){

numTrainers = currentTrainers.getValue();

index++;

} else {

numTrainers = 0;

}

new\_catch\_rate = p.getCapture\_rate()\*(1 - (numTrainers\*1.0)/(userNumber));

Then we want to ensure that in the array are only present 29 values, not one more, because we want at the end of the operation to have precisely 30 entries.

capture\_rates = p.getCapture\_rates();

if(capture\_rates.size() >= 30){

while(capture\_rates.size() < 30)

capture\_rates.remove(0);

}

Then we add the new value computed to a list that will be used to update the catch rate of every single Pokemon using the operation called at last.

capture\_rates.add(new\_catch\_rate);

for(Double d: capture\_rates){

System.*out*.println(p.getName() + " " + d.doubleValue());

}

long start3 = System.*currentTimeMillis*();

long count = pokemonManager.updatePokemon(oldPokemon, p);

long end3 = System.*currentTimeMillis*();

System.*out*.println("Updated " + count + " rows. Duration: " + (end3 - start3));

catchRatesOfPokemons.add(new PokemonAndCatchRate(p.getName(), new\_catch\_rate));

}

teamManager.updateCatchRateOfPokemon(catchRatesOfPokemons);