|  |  |  |  |
| --- | --- | --- | --- |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
|  | |  | |
| **Date:** | | 21 October 2015 | |
| **Time:** | | 1915-2215 | |
| **Venue:** | | SIS GSR 3-4 | |
|  | |  | |
| **Attendees:** | | All | |
|  | |  | |
| **Agenda:** | | 1. Code Review for Iteration 3  2. Review of Iteration 3 | |

**Code Review for Iteration 3:**

**Bootstrap of location data**

Validate location look up

The attributes remain the same, the output has to be shown in the left to right order of the header.

If it is a valid data, it will be stored into database. All csv files have their own hashmap to facilitate inputting other data.

All data will be returned in a form of TreeMap<Integer, String[]>, Integer refers to the line number and string[] are the error messages.

Validate location

Location is very similar to App. We check for macaddress, timestamp, valid date and duplicated row.

We use daoLocation, because we are referring to data from the database.

It has to be a valid app-id to be placed in the locationHashMap.

errorList.isEmpty() is to ensure that our methods is completely error-free

locationIndexMap helps us to fetch us the previous line number so that we could place it in the errormap. After checking the duplication, we will return the data.

Validate location-delete

Location delete will not contain same app.

All we need is timestamp and macaddress to prove that it is a unique data. The moment we put down location-id, it has to be the correct one, else it will be invalid. But, there could also be a case where the location-id is not present, then it will be valid.

*Validation Steps:*

1. the validation of data
2. valid data, but not found in database
3. valid record and found in database, only then we carry out the deletion from database.

There is an invalidCount and validCount.

Run an iteration for the current map, if it does not contains key, we will add 1 to invalidCount, else we will add 1 to validCount.

Then, we will do batch-delete.

**Add Data**

Location data

Everything is fetch out from database and store in location hashmap.

Same validation is carried out as the bootstrap of location.

Location-delete data

Fetch out locationlookup to ensure that the data we want to delete are present in the database

Then we will send it to DAO for deletion. It is carried out by batch.

Stateless session was used as it was faster. Those with state has an Entity tagged to it

For stateless, it uses entity but it has no association with it.

Additional Pointers:

* locationMap key values: timestamp + macaddress
* locationIndexMap: all the error data should contain the previous line number
* Location-delete.csv will not contain duplicates : location-id (optional), timestamp plus macaddress compulsory
* Stateless session batchCommit for LocationDAO.java
* What is the difference between stateless and state commits (previous algorithm)?
  + Stateless : session.insert(), session.delete()no more flush;
  + stateless uses the entity, does not attach association with it

**UI Location-delete**

Enter a mac address, start date and end date, it will delete all the records that match it.

**Smartphone Usage Heatmap**

Heatmap controller

Output is a list of ***UsageHeatMap*** object that consists of Semantic place, number of people using phone, crowd density.

getSmartphoneHeatmap method takes in timestamp and floor.

timestamp are within -15 mins and -1 sec to account for 15 minutes prior to the specified date and time – For example, if the input is 3PM for September 1st, use the data reported between 2:45PM (inclusive) to 3PM (exclusive) on the date.

retrieveLocationByStartEndDate takes in startdate, enddate, floor which return us the list of location that fulfils the criteria

2 hashmap were created: lastKnownLocation and uniqueLocationMap as we want to know the last known location of the users

semantic place will be sorted by alphabetical order.

The second condition, besides being in the semantic place, they also have to be using their phone in the semantic place. For each semantic place, we will get last known location to each semantic place. If it is the semantic place, we will look through if users use the app at least once.

The crowd density will be categorised and stored based on the number of people using smartphones (for all location\_ids in the semantic place).

**Social activeness report**

1st part: to know the alone and group time

2nd part: how much social app usage, how much portion of it belongs to a particular app in ascending order

***getSocialActivenessReport*** method that takes in start date, end date and mac address

we will get the location based on the start date and end date

mac address is for us to know this location actually belongs to which mac address

***calculateTotalDuration*** method was created to count to total duration of the person who spend in SIS in a day

it also has a ***getduration*** method within ***calculateTotalDuration*** which takes in 2 locations, to see how long the person stays in a location

***Hashmap<String, Hash<String,List<TimeObject>>***

* The key is the macaddress, the (value) string is the semantic place, the value is the time spend in the semantic place, pertaining to each user.
* TimeObject is an entity class that implement Comparable<TimeObject> to carry out sorting
* There is a method merge
* Several condition in which the TimeObject will merge, it will return true if it is merged. We merge is to cater to the overlapping.

Go through the for loop of location list, if the current mac address is equal to the next mac address,

Last user record: it will add 5 mins to it by assuming that he stays there for 5 mins.

Calculate the macaddress, then we will create a ***Hashmap<String, Hash<String,List<TimeObject>>*** called ***semanticTimeHashMap***.

We will add in the same semantic place to the same user, and assign the time object to the same user by putting it into the hash map.

We will always check if the user already has the semantic place and timeObject. If it is the same semantic place, we will add the time to the list of time object.

***TimeObject*** is dealing with timestamp.

* TimeObject (time, actualEndTime) 🡪 after catering to the last record and new semantic places.
* This is carry on till it is a new user.
* Merge all the timeObject for this semantic place if they can be merged (has overlap). This is done using the merge method in the object class

If it is a different macaddress, if the semantic place has not been added into hashmap yet, we will create, Initialise the ***semanticTimeHashMap***, and then added back to the new macaddress.

After storing all these into a hashmap, we will retrieve locationhashmap of the logged in user.

***constructSocialTimeLine*** method will take in the list of timeObject and the logged in semantic time map and the other users semantic time map.

* This methods compare the semantic place time if it is the same to other users semantic time.

Loop through login user timeObject and compare it with the other user timeObject. If the user timeObject is not before and not after the logged in user timeObject, this shows that there is an overlap.

We will store the social time line of the user which indicates that the user has an interaction with the other user at the period of time. socialTimeLine(new TimeObject(startTime, endDate)

***revisedSocialTimeLineMap<String, List<TimeObject>>:***

* This Hashmap calculates the total social time of other users interacting with the logged in user.
* All the users who has interacted with the logged in users, and the value is the list of Time object>
* Check if there is overlaps with other users, if there is merge, then we will add the time object into the social time. If there is merge, it will be i+2 to move on the next pair of comparison, so that we can merge all the objects that merged
* At the end, we will add all the difference time to form the total social time.

**Others things to note for coding:**

Common validation

* Blank field is referring to the field names
* File-Specific validation
* appcategory (refers to the blank field), when we display the output is “invalid app category”
* there is no missing token, there is only invalid token.
* We need to take note of the missing token validation, the token still has to check if it is blank or empty and show the correct validation message.

**Review of Iteration 3**

Good points:

We cater long hours to PP session hence we are able to go through smoothly for coding. However, there were a lot of mistakes made in this iteration on PP Log Schedule. Thus, we should stay closely to the schedule. For test cases, we also need to follow wiki requirements closely. For iteration 4, we are splitting up the main task into sub tasks in the pplog especially for Regression Testing, it will be ran concurrently.

The meeting was adjourned at 10 pm. These minutes will be circulated and adopted if there are no amendments reported in the next three days.

Prepared by,

Koh Chu Qian

Vetted and edited by,

TANG SHING HEI