WeTok Report

Table of Contents

- WeTok Report
  - Table of Contents
  - Application Description
  - Team Members and Roles
  - Conflict Resolution Protocol
  - Application Use Cases and or Examples
    - New User Register
    - Existing User Login and Other Operations
    - Add a post
    - Interaction activities
    - Subscribe a user
    - Search
  - Application UML
  - Application Design and Decisions
    - Design Reasons
    - Data Structures
      - AVL Tree
    - Design Patterns
      - Singleton
      - Template
      - DAO
    - Grammars
    - Tokenizer and Parsers
    - Surpise Item
      - Ranking algorithm
      - Simple personalisation
  - Summary of Known Errors and Bugs
  - Testing Summary
    - AVL Tree Testings
    - Dao and Bean Testings

        - Parser and Tokenizer Tests
        - Ranking Tests
    - Implemented Features
    - Team Meetings

Application Description

Wetok is a social media app specifically aimed at person with a strong personality. You can share your status and mood with your friends, people in the same city, and even strangers anytime, anywhere. In here, you do not have to worry about cyber-violence and personal abuse, we only provide you with a pure sharing platform. As our logo shows, we talk, but we don't comment. Furthermore, you can search for tags you're interested in and subscribe people that you're interested in. You can understand what is happening in this world through wetok. Welcome to Wetok.

Team Members and Roles

UID	Name	Role
u6120911	Xinyu Kang	Integration Engineer, Data Structure Designer, Ranking Algorithm Designer
u7151386	Xinyue Hu	Database Designer, Data Structure Designer, Integration Engineer
u6684233	Yuxin Hong	Tokenizer and Parser Designer, Bug Killer， Logo Designer
u6111569	Zhaoting Jiang	UI Designer, Fragment designer

Conflict Resolution Protocol

1. Conflictors organize their views, list the pros and cons of their choice.
2. Initiate a zoom meeting with all team member.
3. Conflictors presenting the confliction and why disagree with the others.
4. Team member do a vote and share the reason.
5. As a result, merge conflictors's solutions or take more agreed solutions.

Application Use Cases and or Examples

Targets Users: Everyone who is willing to use and learn about this app

- *Users can use it to post whatever they want to say.*
- *Users can use it to find the post nearby them.*
- *Users can use it to delete their post whenever they want to.*
- *Users can use it to like others' post.*
- *Users can use it to dislike others' post.*
- *Users can use it to follow the person that they are interested in.*

Jack wants to learn about skateboarding skills that interest him most

1. Jack searched the tag about skateboarding and found that Kevin shared skateboarding skills every day
2. He followed Kevin
3. He learned new moves from Kevin's skateboard skills

Andy wants to know about the good restaurants around him

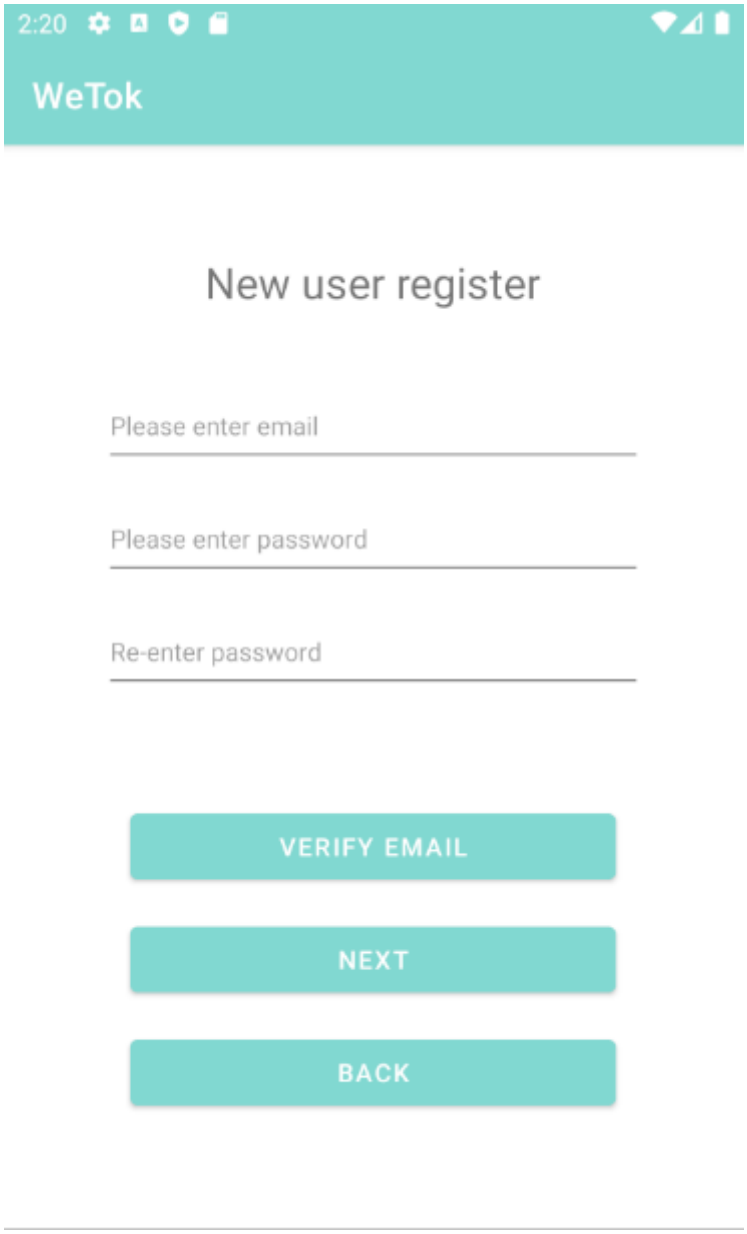
1. Andy turns on the City feature and finds that Claire shared a delicious restaurant nearby
2. He press the like button for that post.
3. He followed Claire and hope Claire can share more delicious restaurants in the future

Stan wants to delete his post

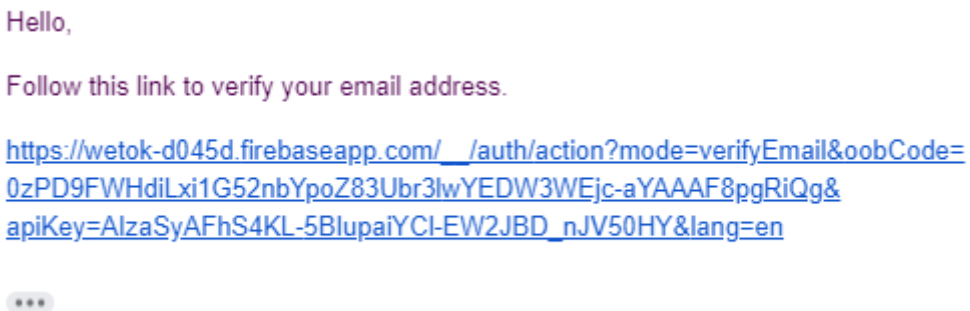
1. Stan post his relationship with his girlfriend a few weeks ago
2. He broken up with his girlfriend yesterday
3. He found his previous post in the user session
4. He deleted his post

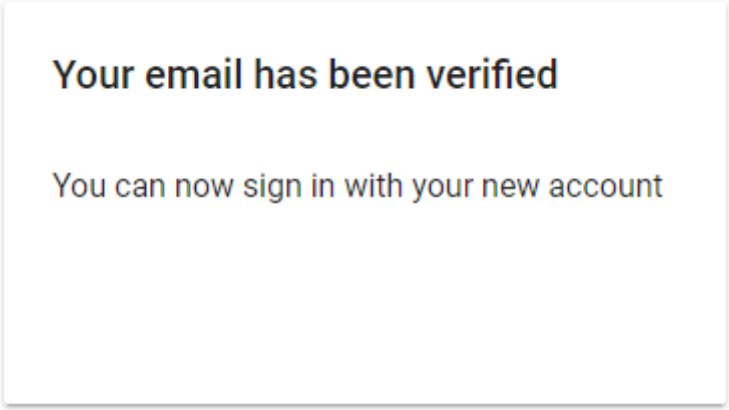
New User Register

1. Click *REGISTER* in the login page, go to register page:

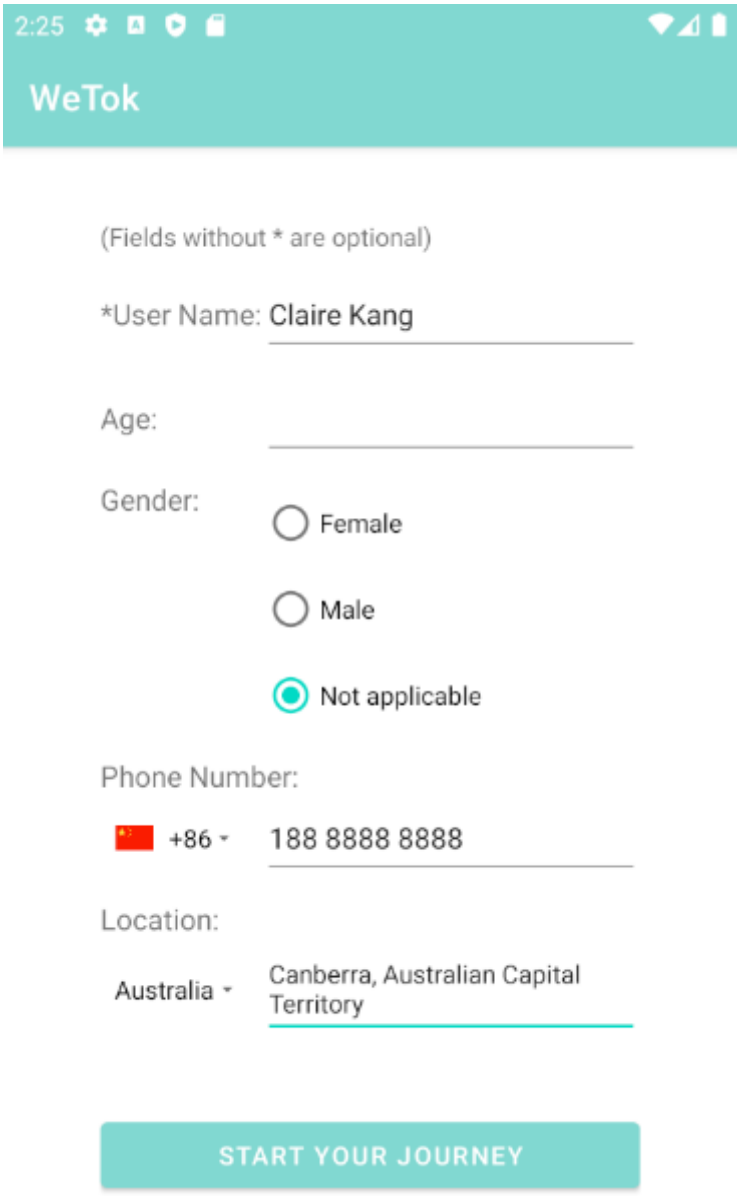


2. Enter a valid email (gmail is recommended) and your password, click *VERIFY EMAIL* and the system will remind you that "Verification email sent. Please verify your email to continue". Then check your mailbox and click the varification link:





3. After successfully varified your email, click *NEXT* and go to user information register page. All the fields are optional except for the username. You can select the country for your phone number and address.

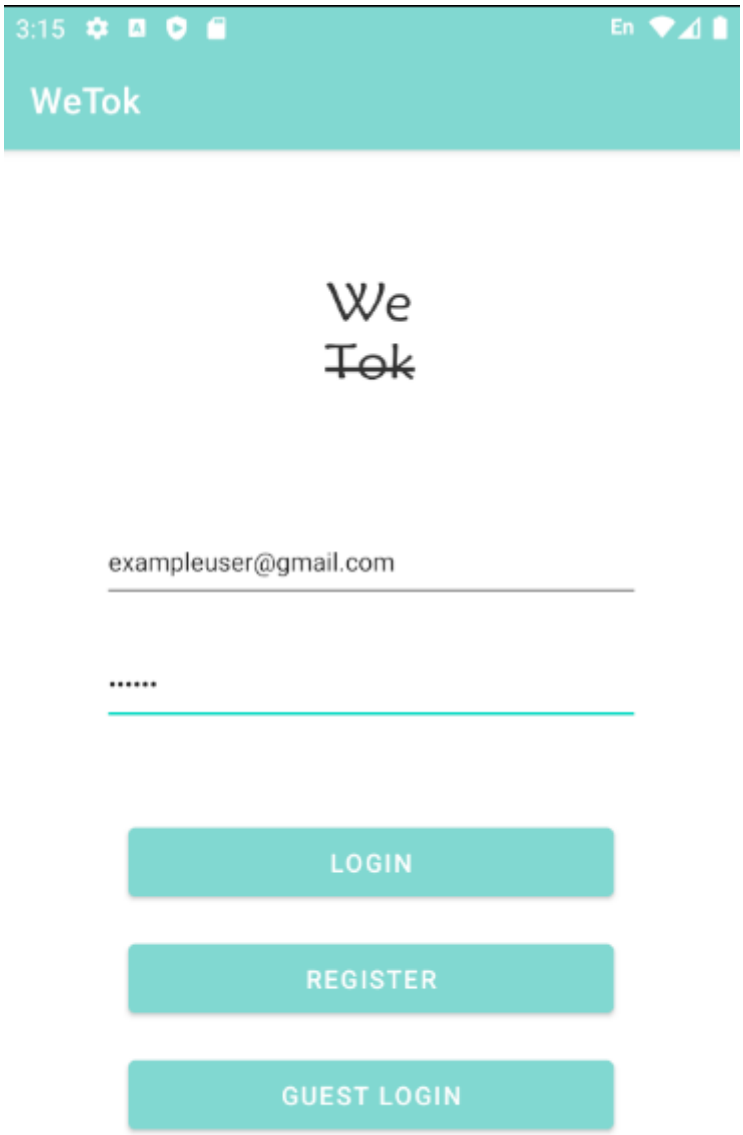


You can enter only several characters of your city's name, then you may choose your city from the system's recommendations.

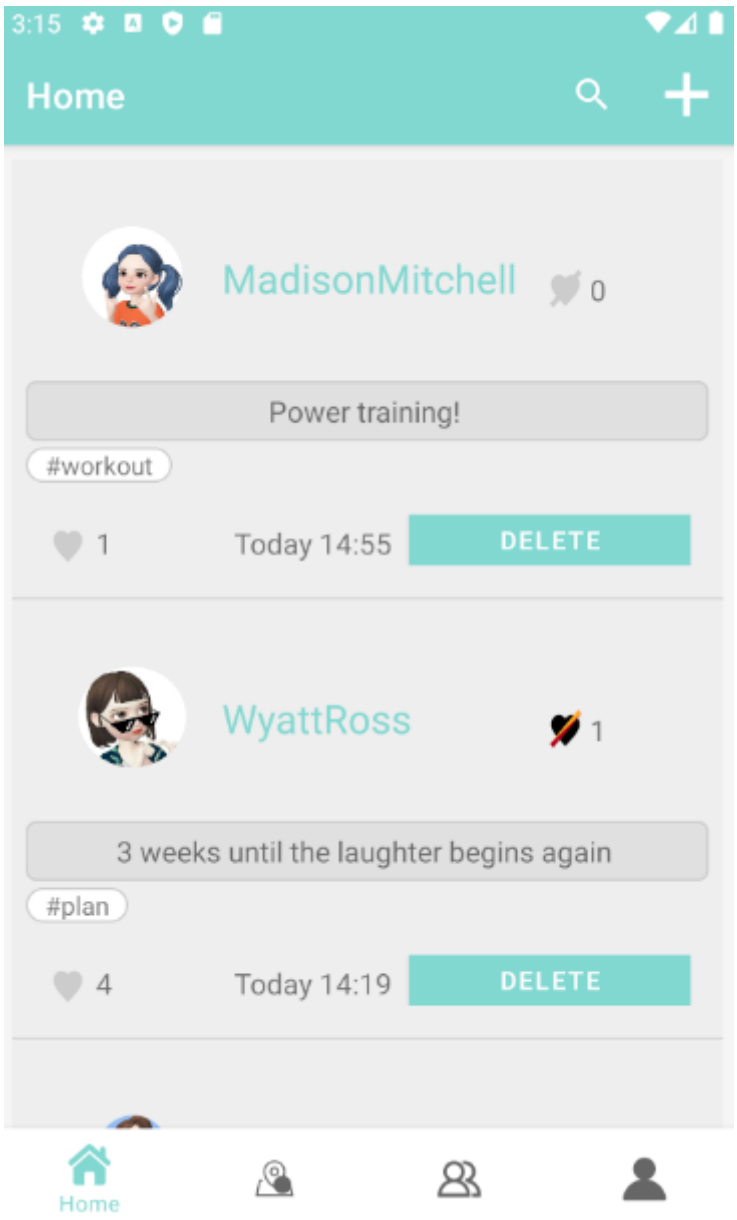
4. The final step is to click *START YOUR JOURNEY* and you will go to the main page.

Existing User Login

1. Here we'll show to how to login as an existing user. We provided an example user: email = [exampleuser.gmail.com], password = [123456] . Feel free to login as this example user for testing and exploring 😃

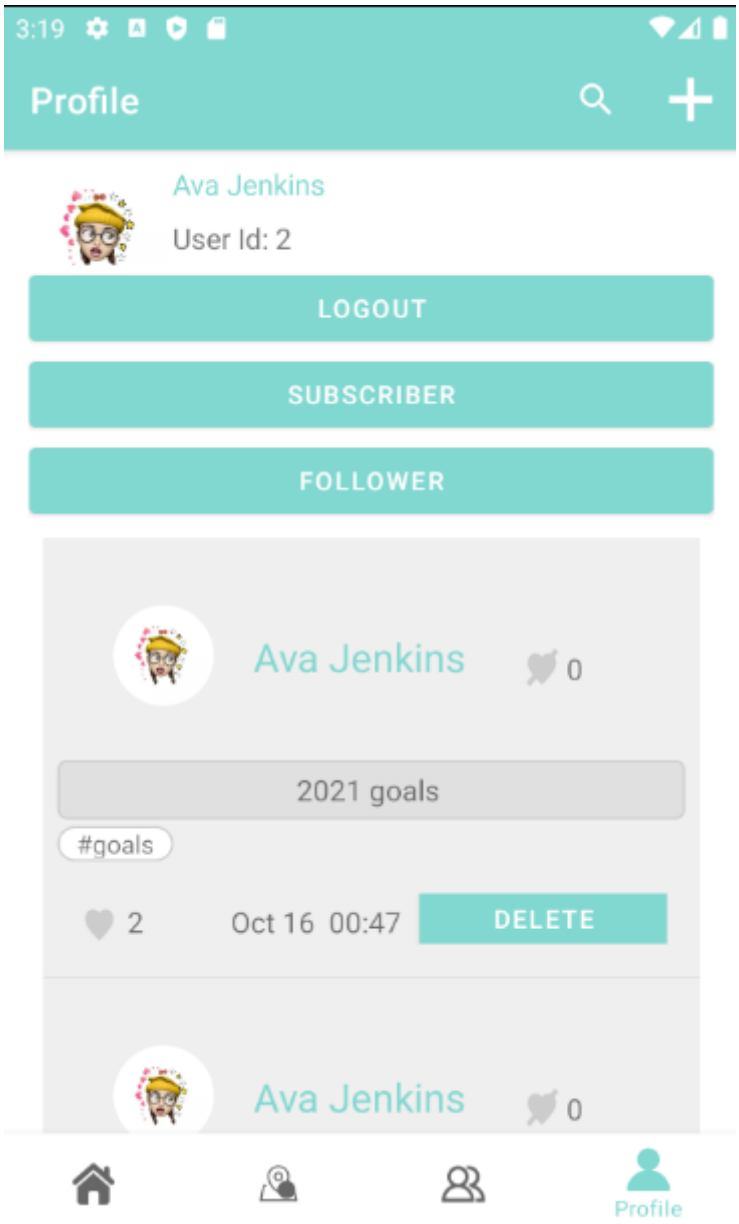


2. After you signed in, you will to go the home page of WeTok, which lists all the posts sent recently:



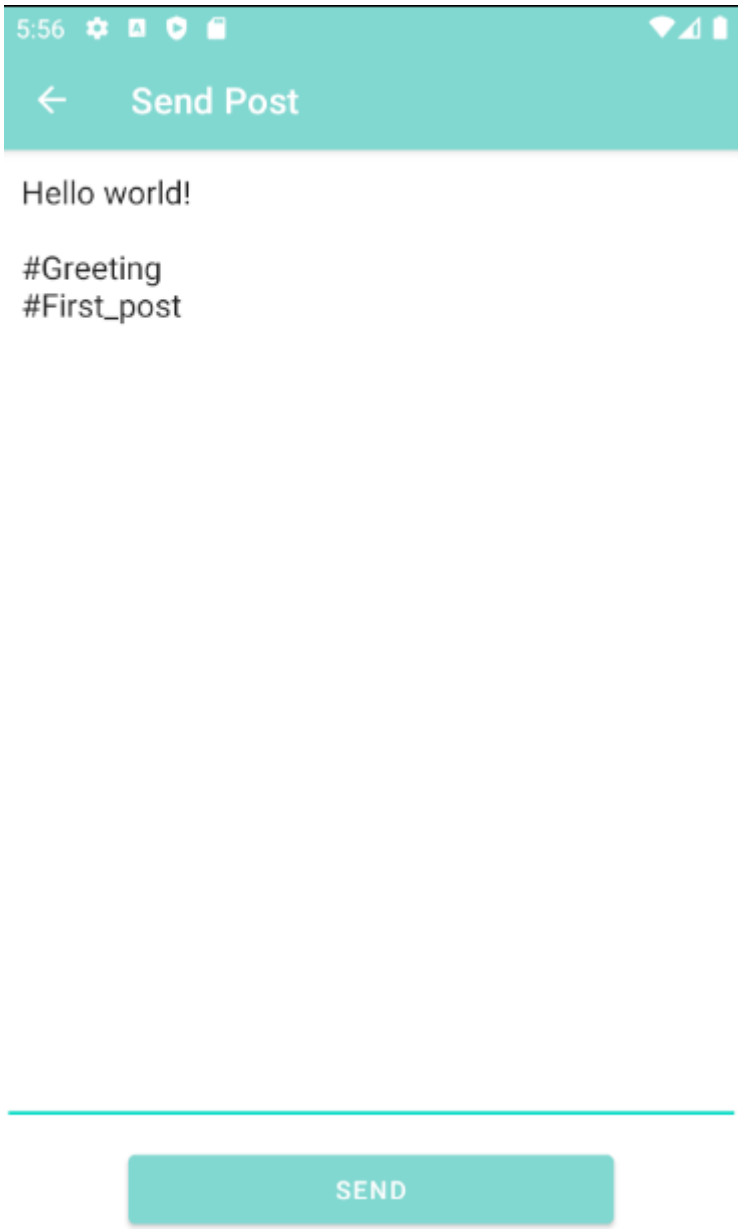
You can view the posts in your city by clicking the second icon *City* in the navigation bar, or, view the posts sent by your subscribers by clicking the third icon *Focus* in the navigation bar.

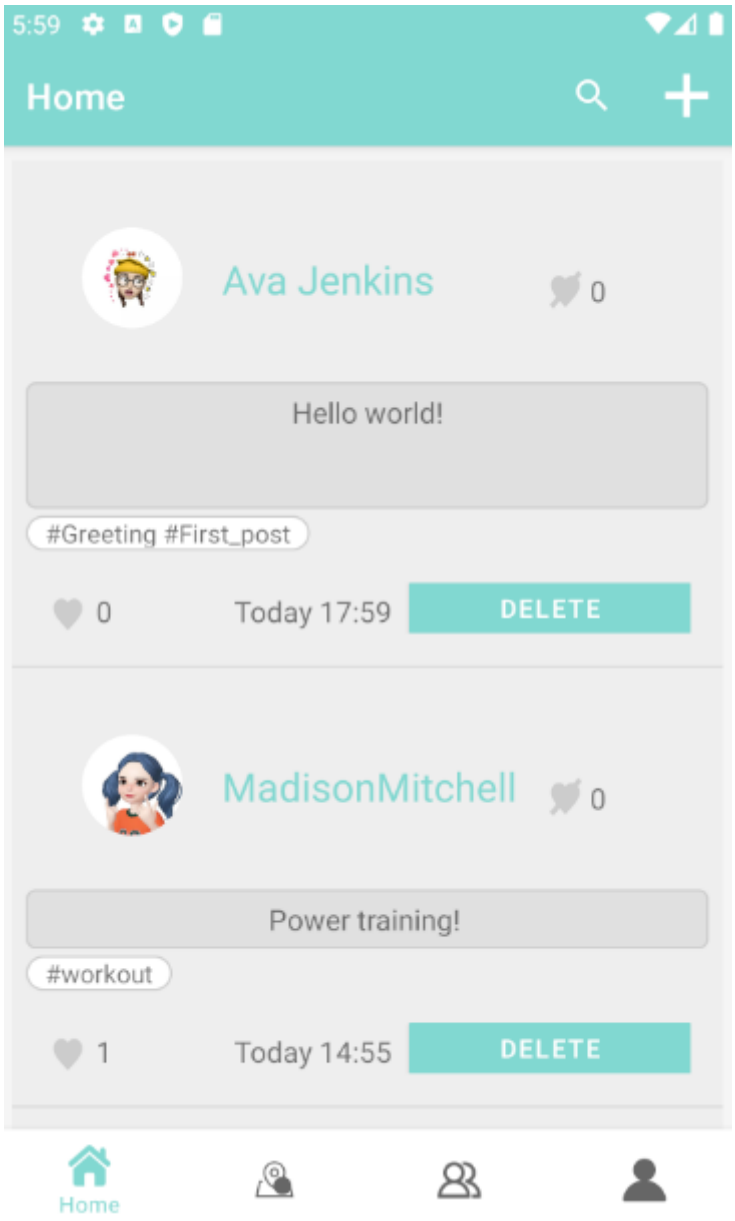
Moreover, you can click the right-most icon *Profile* in the navigation bar to view your username, user id, posts history, subscribers, and followers:



Add a post

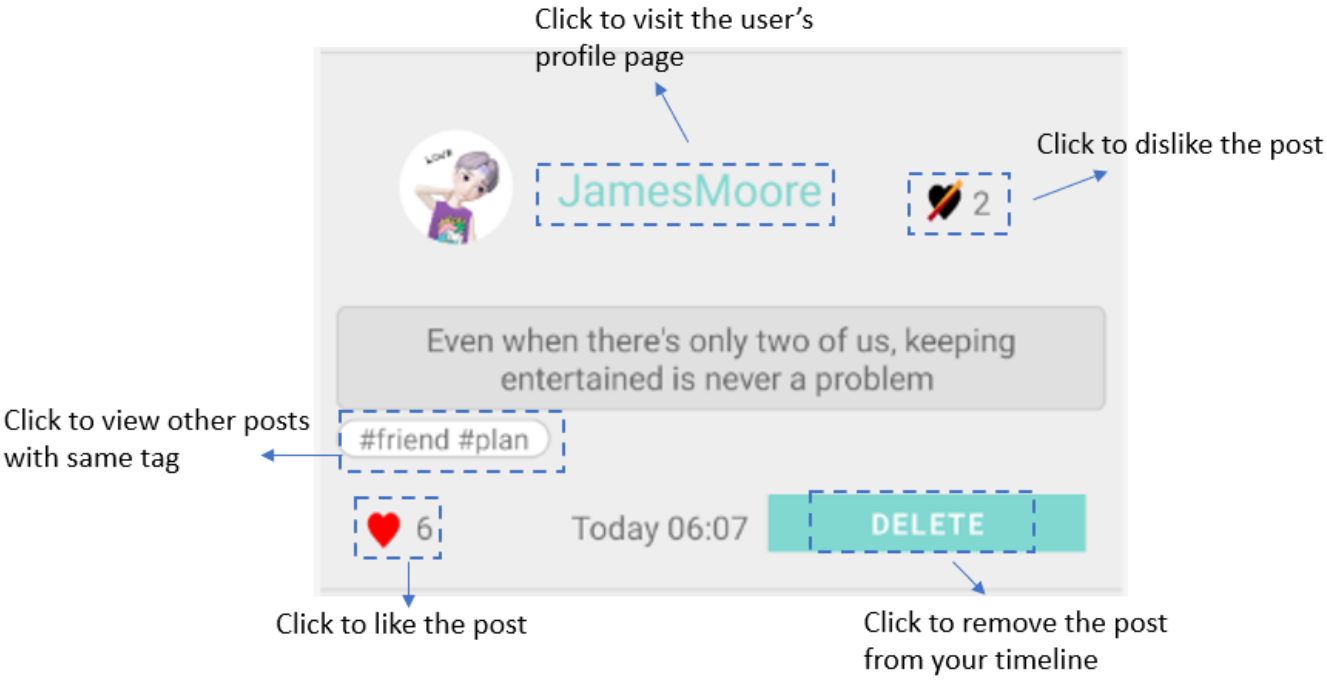
1. Click the "+" at the top right corner. Write your post and add tags if you want.
2. Click the *Send* button, then you can view your new post at home page and your profile page:


5:56
Send Post
Hello world!
#Greeting
#First_post
SEND



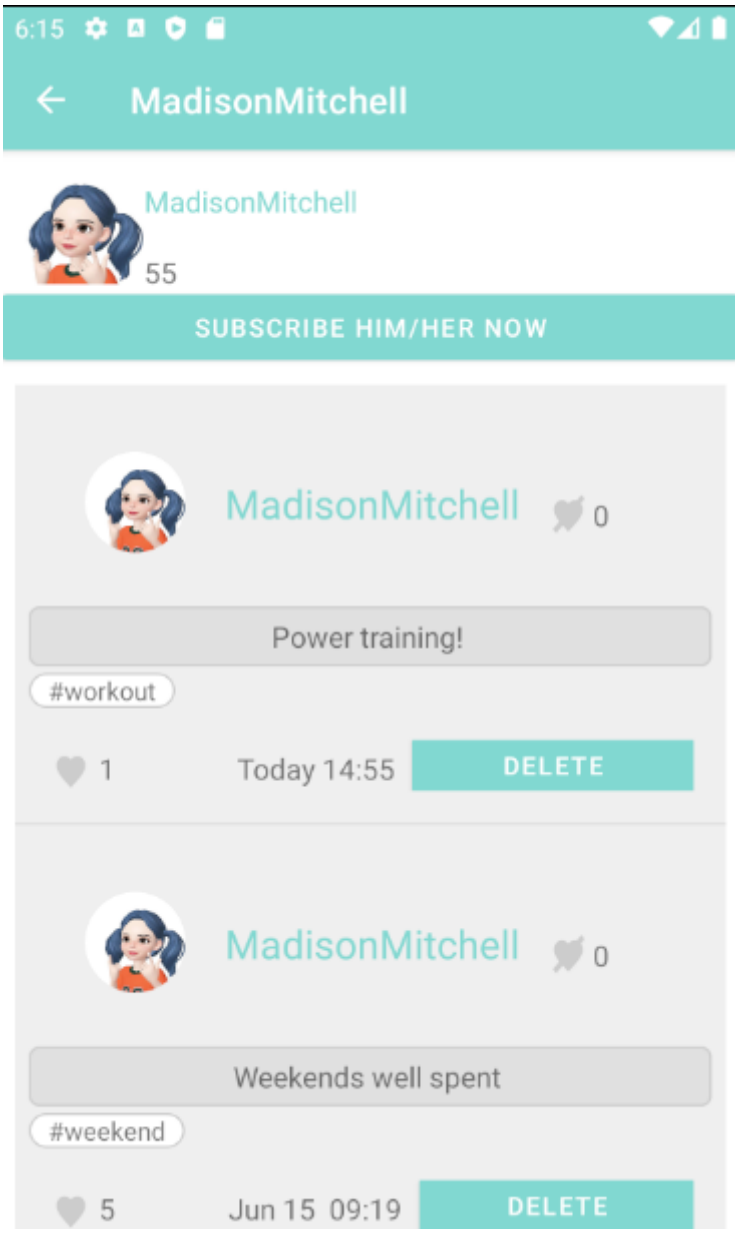
Interaction activities

For each post, the user can have following interaction activities: like, dislike, delete, follow, etc.



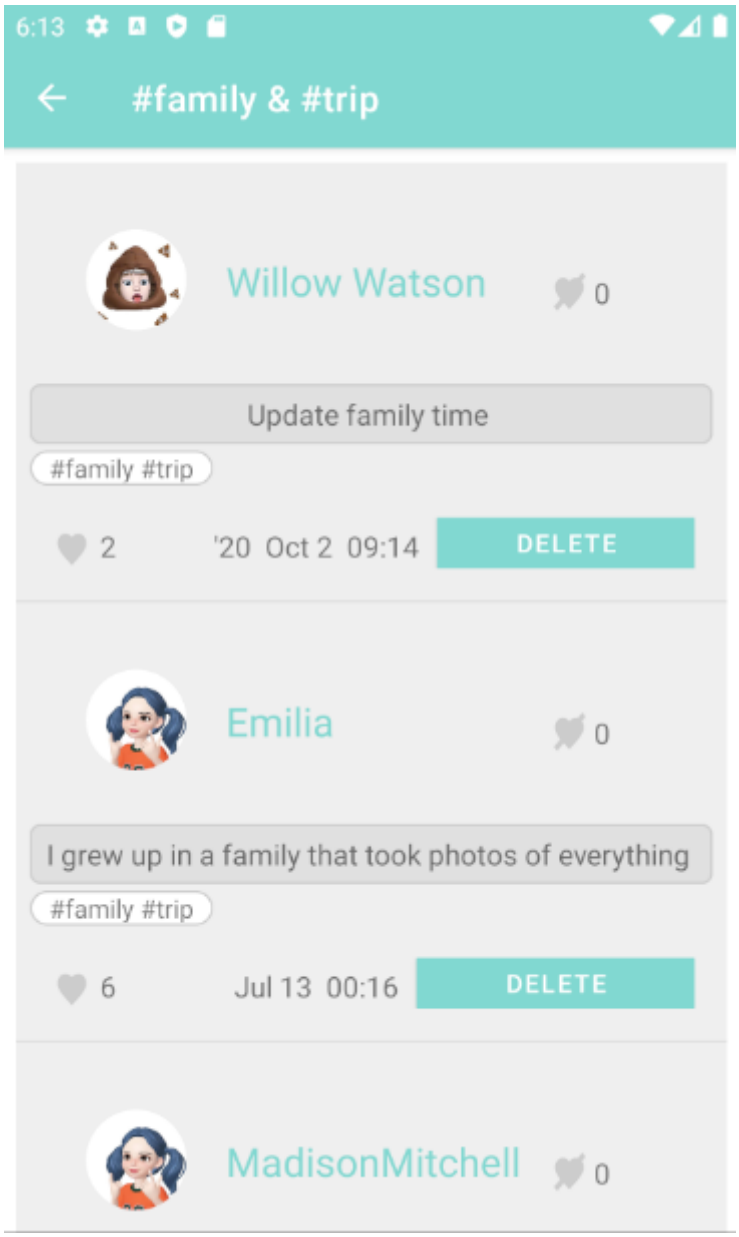
Subscribe a user

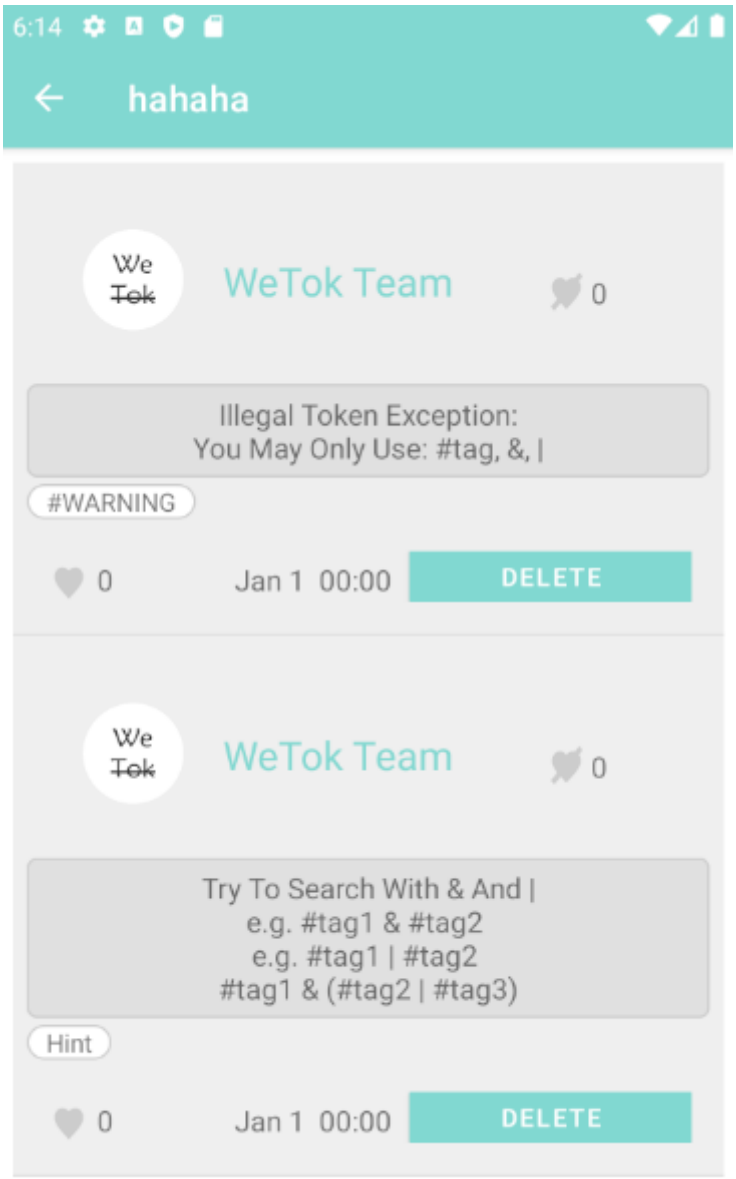
To subscribe a user, you need to first go to his/her profile page, and click the button *SUBSCRIBE HIM/HER NOW*.



Search

To search for posts with certain tags, click the magnifying glass icon at the top right corner, and write down your query in correct grammar. The retrieved posts will be ranked by our ranking algorithm and presented to you. Entering query with invalid grammar will return the warning and hint from the Wetok group.

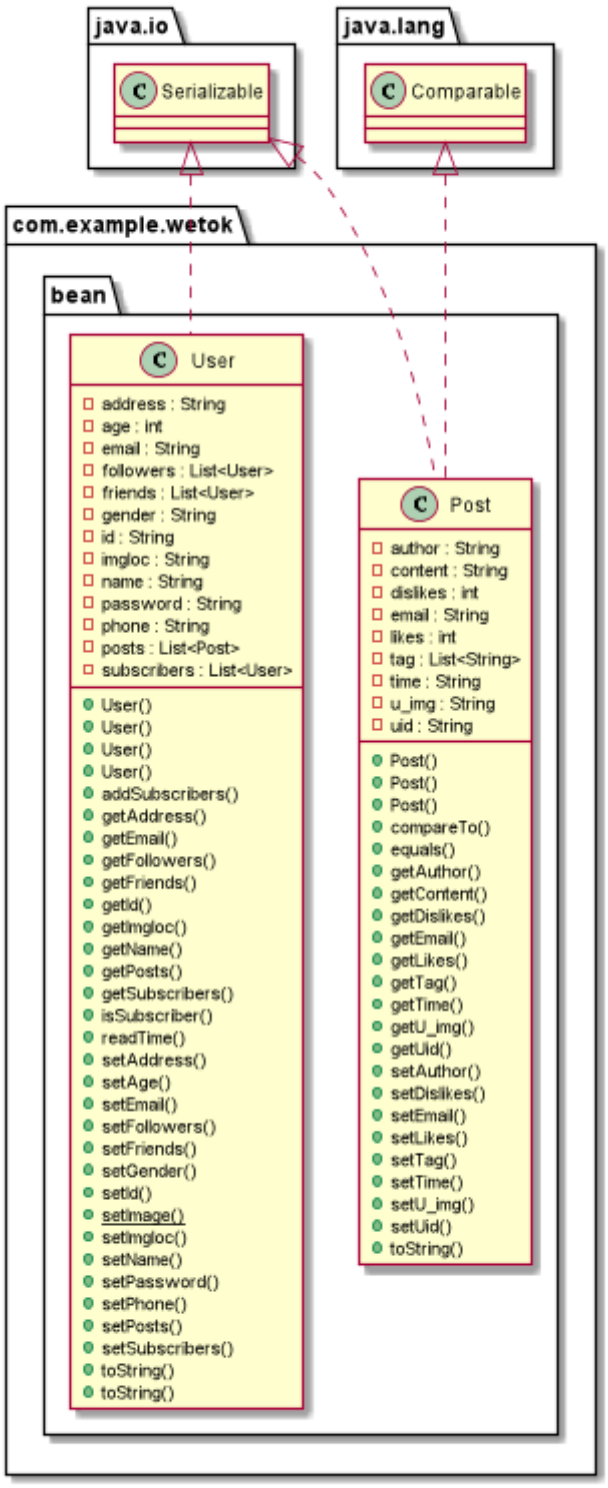

6:13
#family & #trip
Willow Watson
0
Update family time
#family #trip
2
'20 Oct 2 09:14
DELETE
Emilia
0
I grew up in a family that took photos of everything
#family #trip
6
Jul 13 00:16
DELETE
MadisonMitchell
0



Application UML

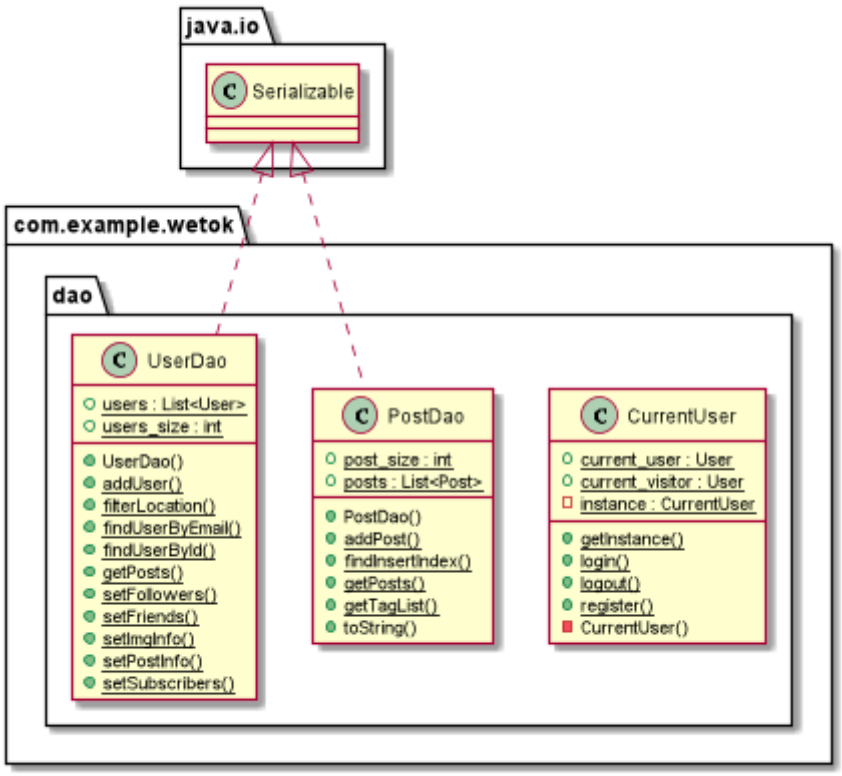
Since the UML of the whole project is too huge, we created the UMLs for the 8 packages inside *com.exmaple.wetok* : package *bean*, package *dao*, package *searchTree*, package *parserAndTokenizer*, package *ranking*, package *fragment*, package *view*, package *resource*:

BEAN's Class Diagram



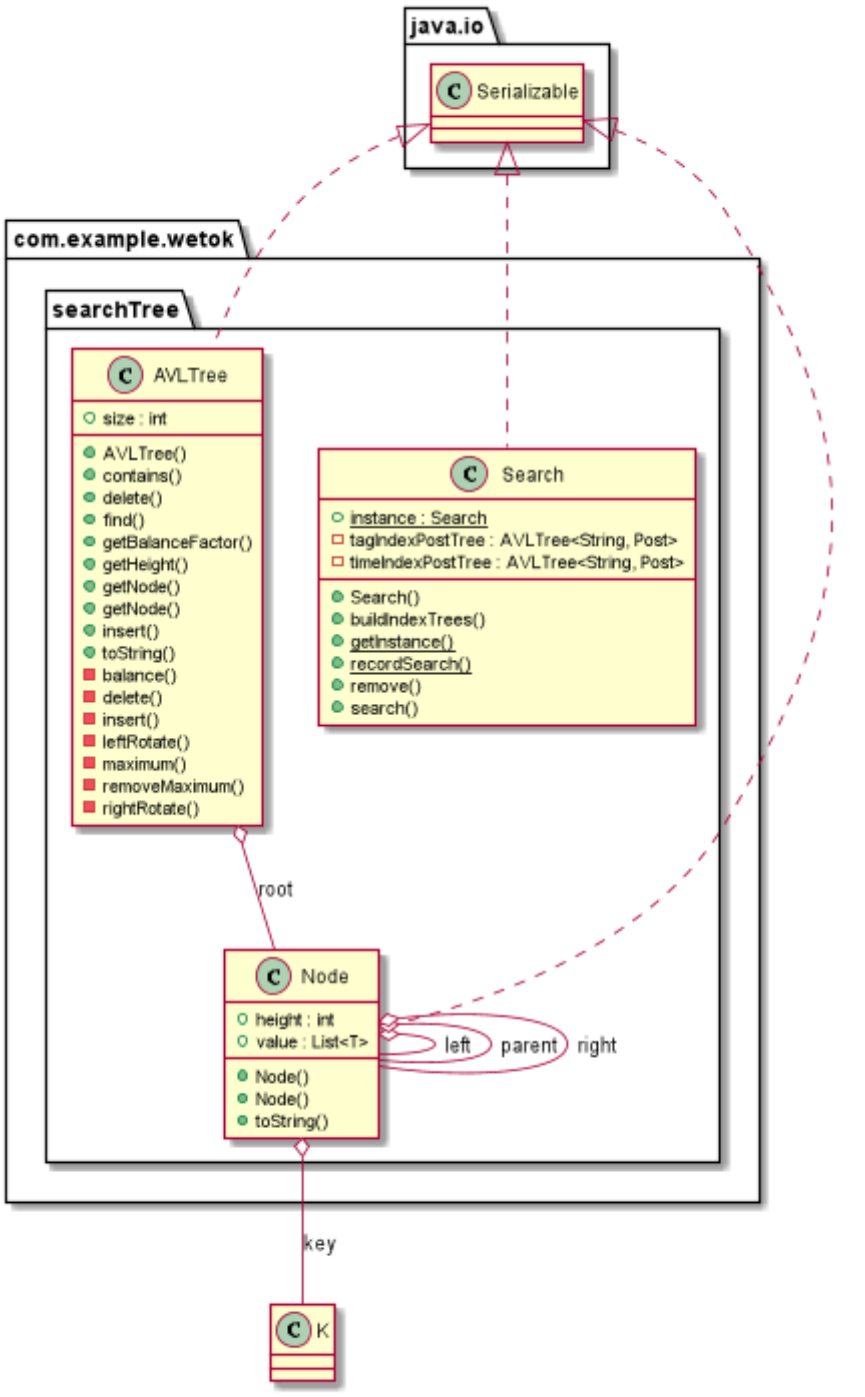
PlantUML diagram generated by SketchIt! (https://bitbucket.org/pmesmeur/sketch.it)
For more information about this tool, please contact philippe.mesmeur@gmail.com

DAO's Class Diagram



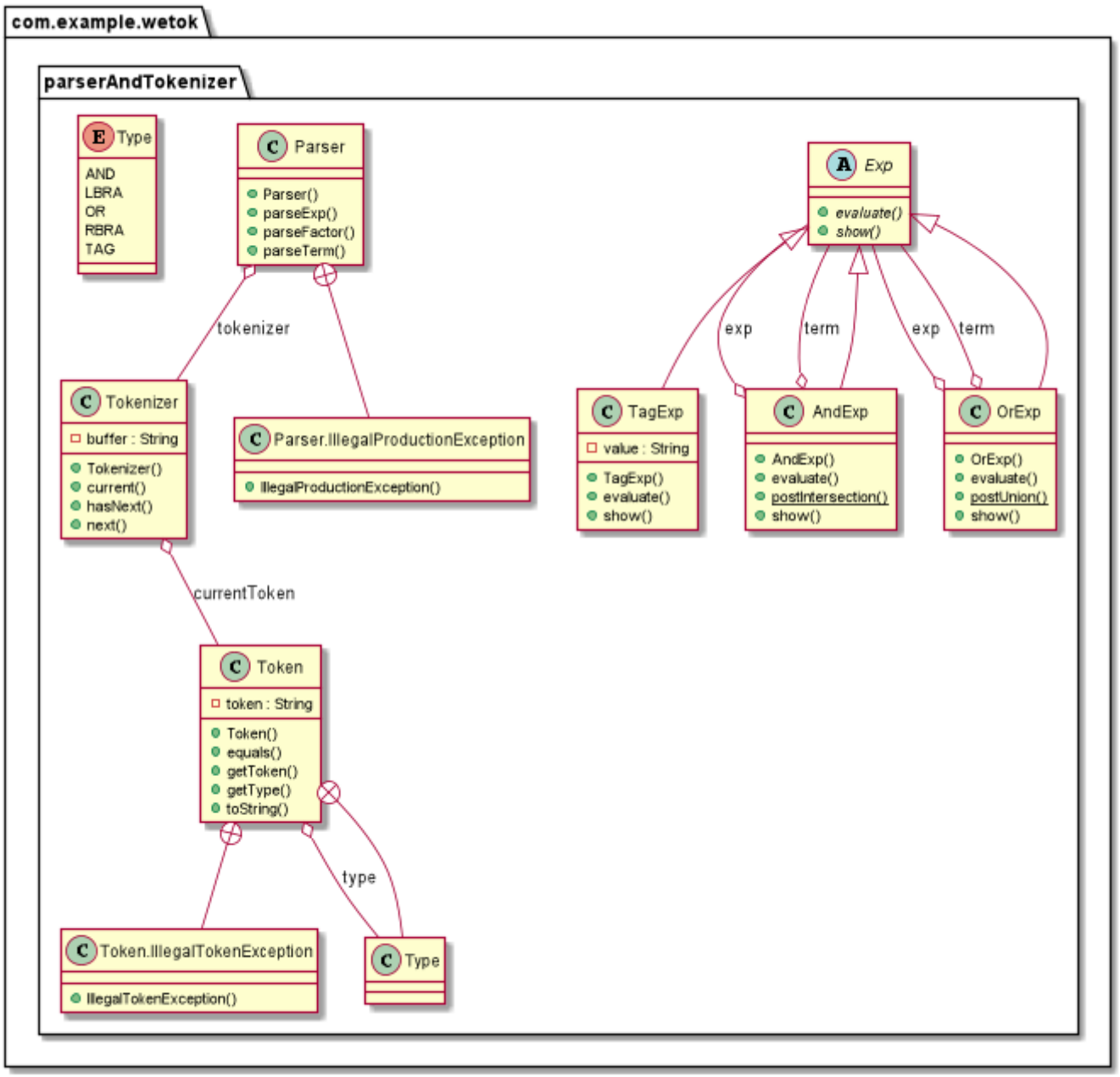
PlantUML diagram generated by Sketchit! (https://bitbucket.org/pmesmeur/sketch.it)
For more information about this tool, please contact philippe.mesmeur@gmail.com

SEARCHTREE's Class Diagram



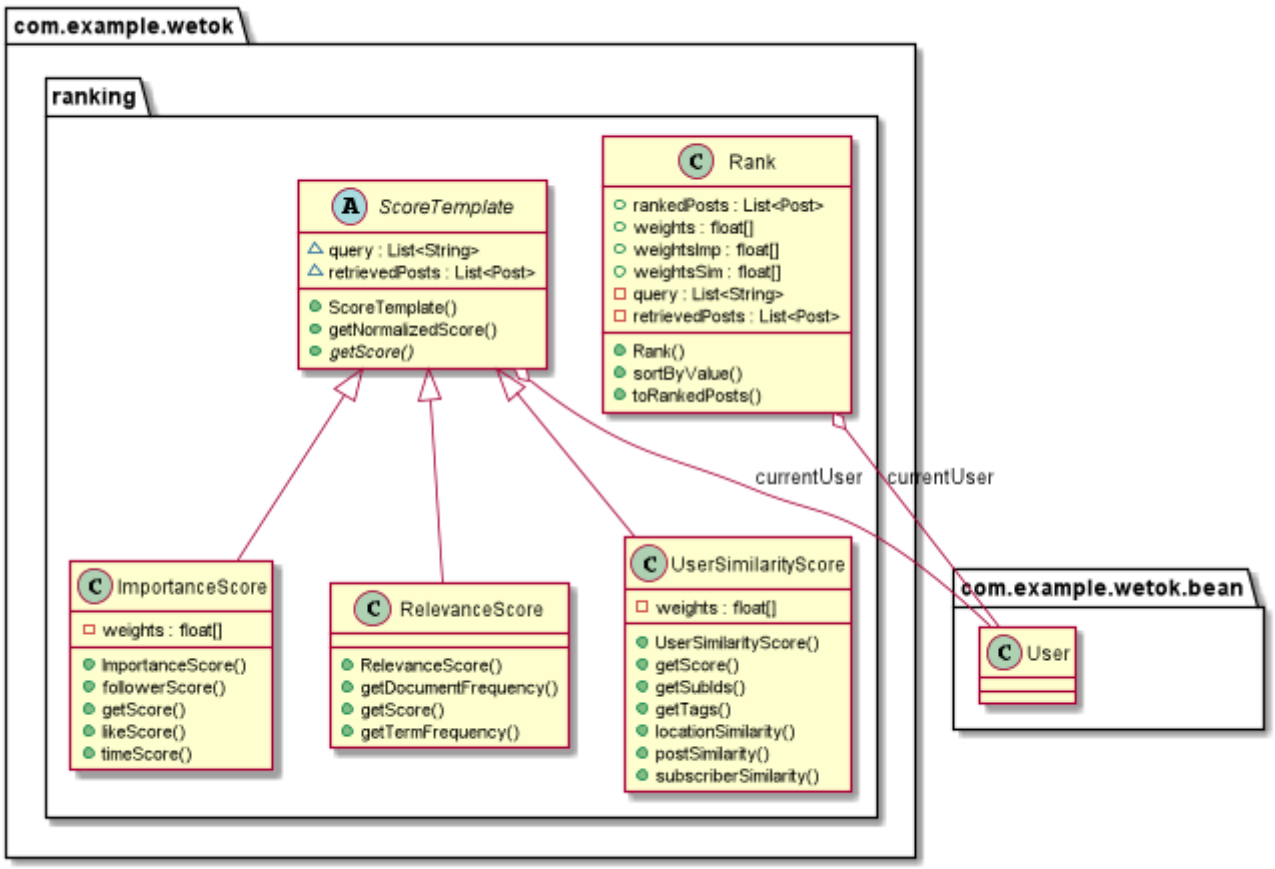
PlantUML diagram generated by SketchIt! (https://bitbucket.org/pmesmeur/sketch.it)
For more information about this tool, please contact philippe.mesmeur@gmail.com

PARSERANDTOKENIZER's Class Diagram



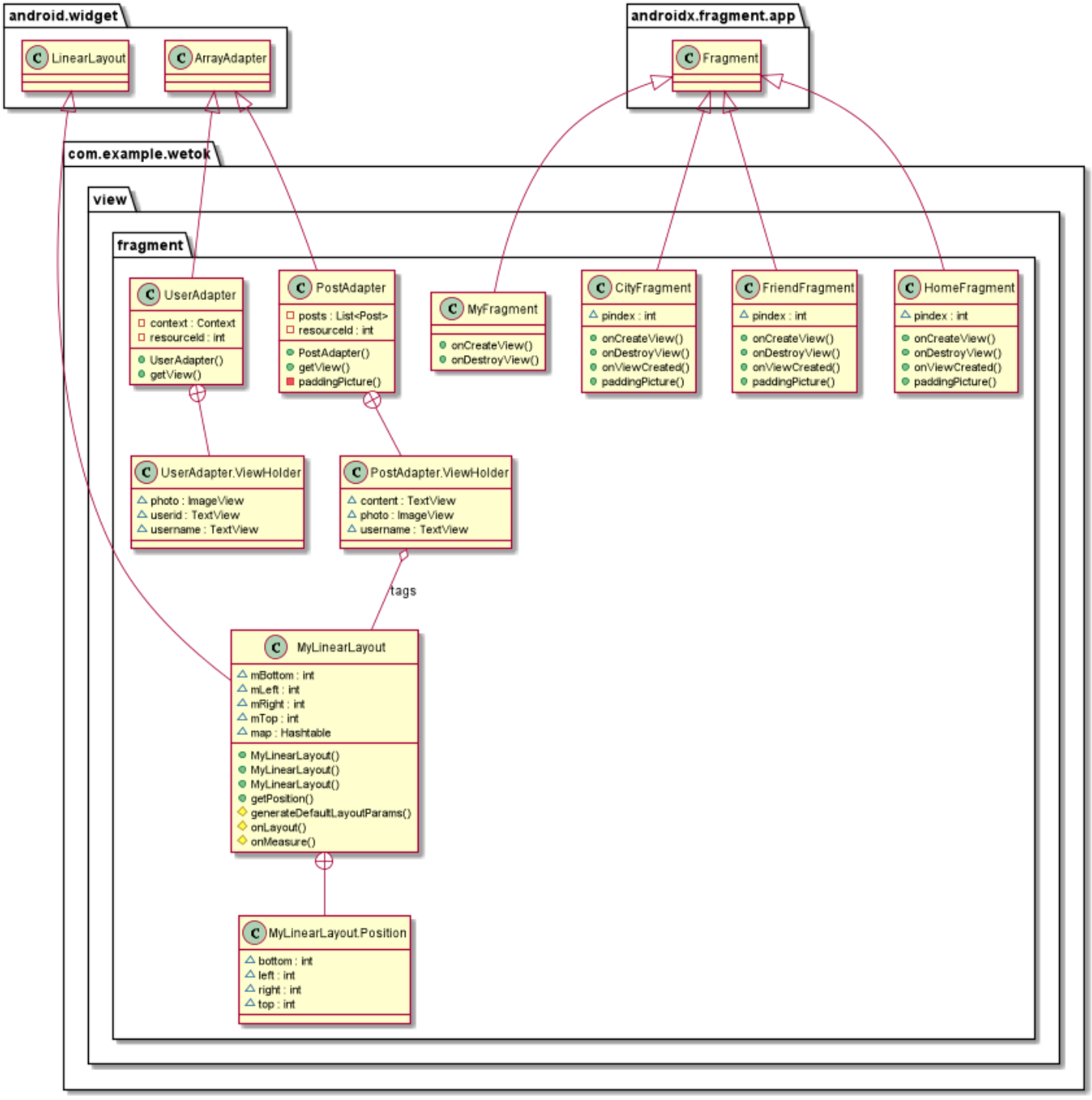
PlantUML diagram generated by SketchIt! (https://bitbucket.org/pmesmeur/sketch.it)
For more information about this tool, please contact philippe.mesmeur@gmail.com

RANKING's Class Diagram

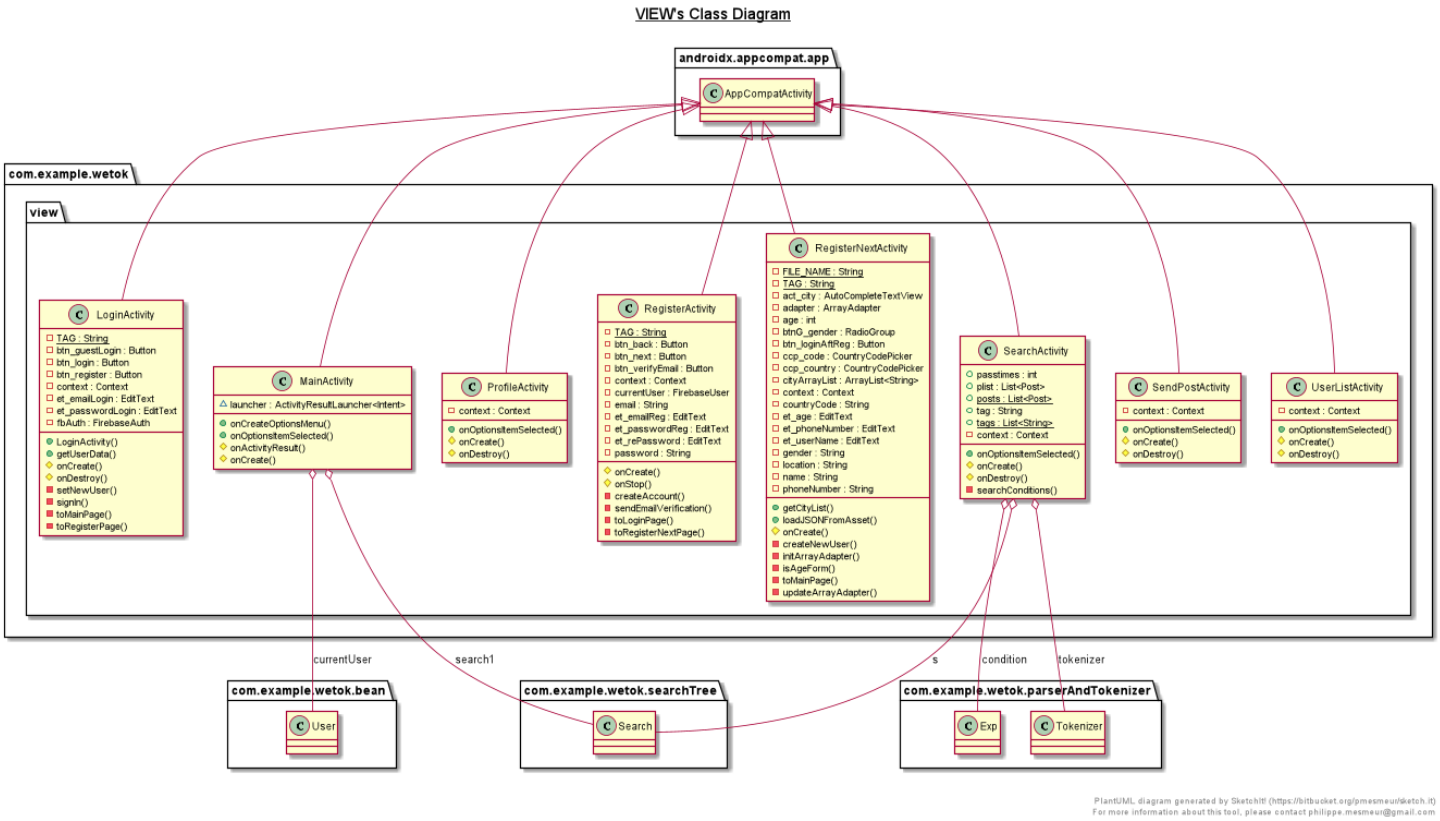


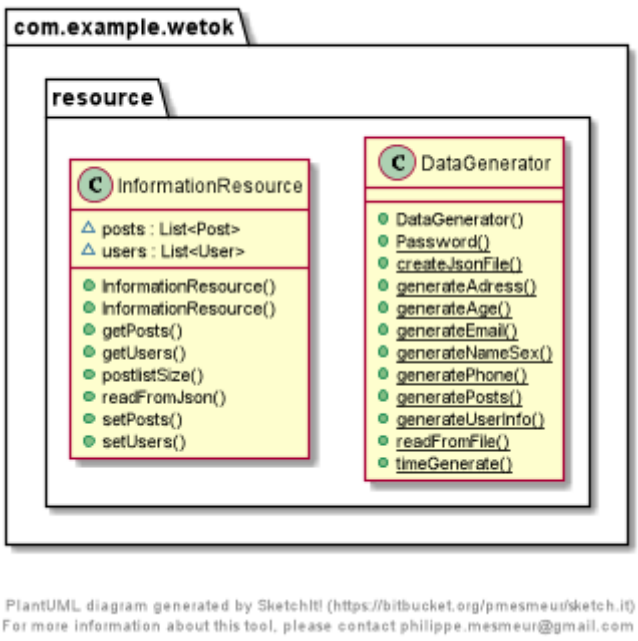
PlantUML diagram generated by SketchIt! (https://bitbucket.org/pmesmeur/sketch.it)
For more information about this tool, please contact philippe.mesmeur@gmail.com

FRAGMENT's Class Diagram



PlantUML diagram generated by SketchIt! (https://bitbucket.org/pmesmeur/sketch.it)
For more information about this tool, please contact philippe.mesmeur@gmail.com





Application Design and Decisions

Design Reasons

1. Data Structures

1.1 AVL Tree

- Objective: It is used for storing posts for valid and invalid search feature.
- Locations: *AVLTree.java*, *Node.java*, *Search.java*
- Reasons:
  - Compared to Binary Search Tree, AVLTree and RBTree query efficiency is more consistent;the best and worst time complexity of queries are both O. (logN).
  - AVLTree outperforms RBTree in terms of query efficiency. Take the user behavior into account.Many users like search topics of their interests.Clearly, AVLTree is better in tune with user requirements.
  - The cost difference between AVLTree and RBTree when inserting nodes is not significant, and the time complexity is O. (1). Futher more, It is more efficient than Arraylist for insertion with a time complexity O(1),We don't need to access the item by index for this feature.
  - RBTree has a lower time and space cost than AVLTree when it comes to removing nodes.Since the amount of data is relatively small, user behaviors in WeTok (such as like, follow, view, dislike, etc) are considerably more common than the behavior of deleting post. In this senario, we decided to choose AVLTree as our datastructure.

2. Design Patterns

2.1 Singleton

- Objective: It is used for making sure of that there is exactly one instance of the current user
- Locations: *CurrentUser.java*
- Reasons:
  - It can effectively avoid the case of multiple-current-users error
  - We need a class storing the current user instance that can be access from the whole project

2.2 Template

- Objective: It is used for computing the ranking scores of the retrieved posts from different dimensions.
- Locations: *ScoreTemplate.java* (abstract class), *RelevanceScore.java* (concrete class), *ImportanceScore.java* (concrete class), *UserSimilarity.java* (concrete class)
- Reasons:
  - We want to sort the posts based on three criteria. There is a common process (i.e., calculate scores, normalize scores) to scoring the posts but each criterion has its own scoring logic.
  - The template method can clearly define the structure and make the code more readable and reuseable.

2.3 DAO

- Objective: It is used for storing the users and posts data read from persistent files.
- Locations: *UserDao.java*, *PostDao.java*
- Reasons:
    - We want to decouple domain logic from persistence mechanisms and avoid exposing details of the data storage.
    - The DAO method allows JUnit test to run faster as it allows to create Mock and avoid connecting to database to run tests.

3. Grammars

- Objective: Process multiple condition at once with *AND* and *OR* operator.
- Locations: *Parser.java*.
- Reasons: Process multiple-tag search.

4. Tokenizer and Parsers

- Objective: Parse expression in tokens: *TAG, AND, OR, LBRA, RBRA*. Search multiple tag at once with *AND* and *OR* operator.
- Locations: *Tokenizer.java*, *Parser.java*.
- Reasons: Process multiple-tag search.

5. Surpise Item

5.1 Ranking algorithm

- Objective: It is used for ranking the retrieved posts
- Locations: Package *ranking* : *Rank.java*, *ScoreTemplate.java*, *RelevanceScore.java*, *ImportanceScore.java*, *UserSimilarityScore.java*
- Reasons:
    - This is an interesting feature and closely related to society
    - We want to rank the retrieved posts based on more than the post time
    - There are members in our group who have learned information retrieval

5.2 Simple personalisation

- Objective: It is used for logging users' addresses to improve the timeline creation
- Locations: Package *view* : Package *fragment* : "CityFragment.java*
- Reasons:
    - This is the simplest way to personalise the timeline creation.
    - We have thought about maximising the chances of accurately providing search results or information in a timeline by using user posts, user interactions and so on.

    - However, in the ranking algorithm we considered the similarity between the current user of the sender of the posts as a negative factor of search ranking to break *Filter Bubbles*. We do not want to "offset" the effect of the ranking algorithm.
    - So, in our app the users can choose to enable (by click on *city* in the navigation bar) or unable the simple location personalisation according to their preferences .

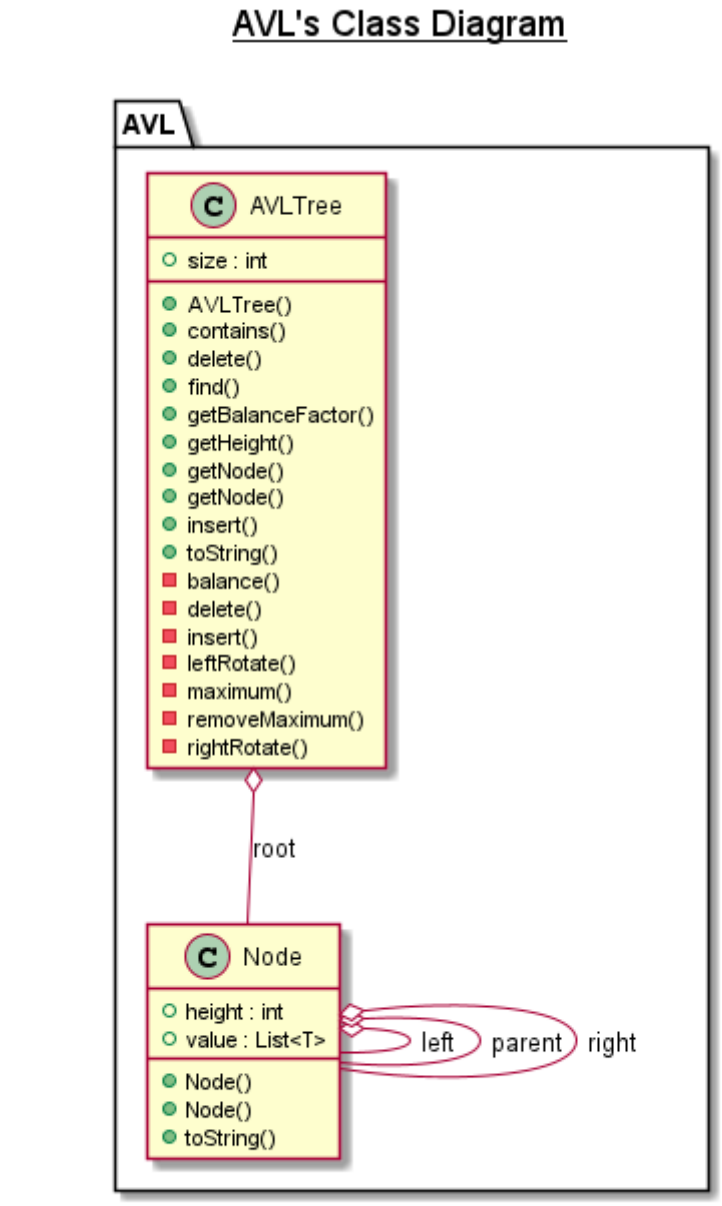
Data Structures

AVL Tree

We selected AVL tree as the data structure of posts operations. AVL tree is a self-balancing binary search tree. It controls the height of the tree and prevents it from becoming skewed. For operations considered they have time complexity:

- Rotations to achieve balanced tree *O(1)*
- Insertion *O(log(n))*
- Deletion *O(log(n))*
- Search *O(log(n))*
- Max/Min *O(log(n))*

It is an efficient data structure and here is the UML of our implementation:

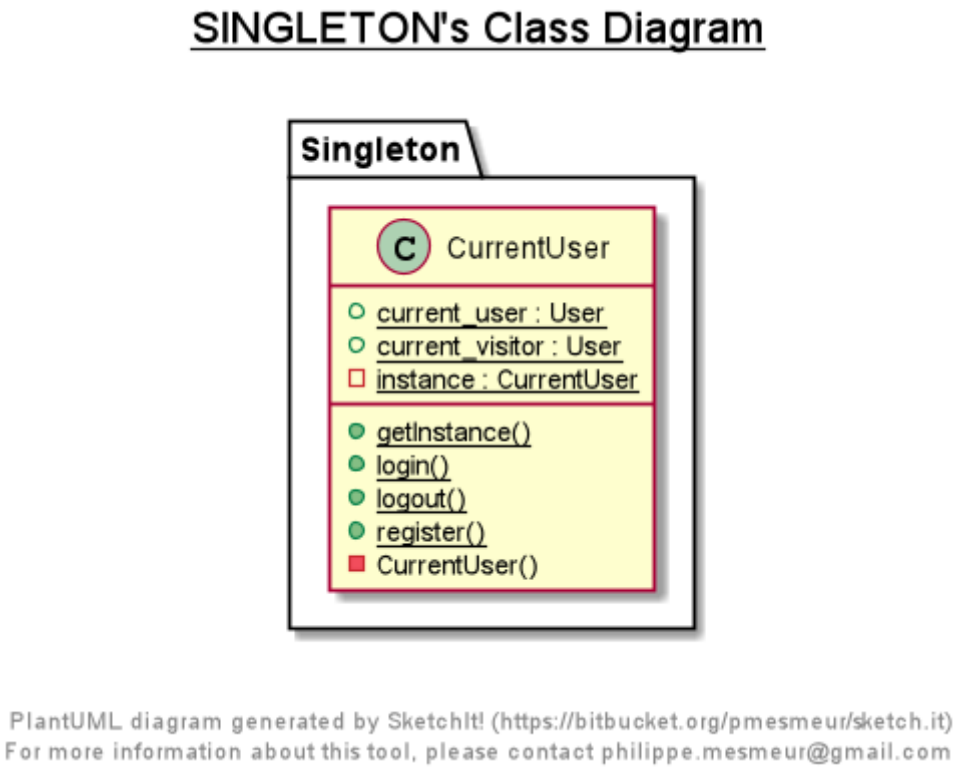


Design Patterns

We used three design patterns: Singleton, Template, DAO

Singleton

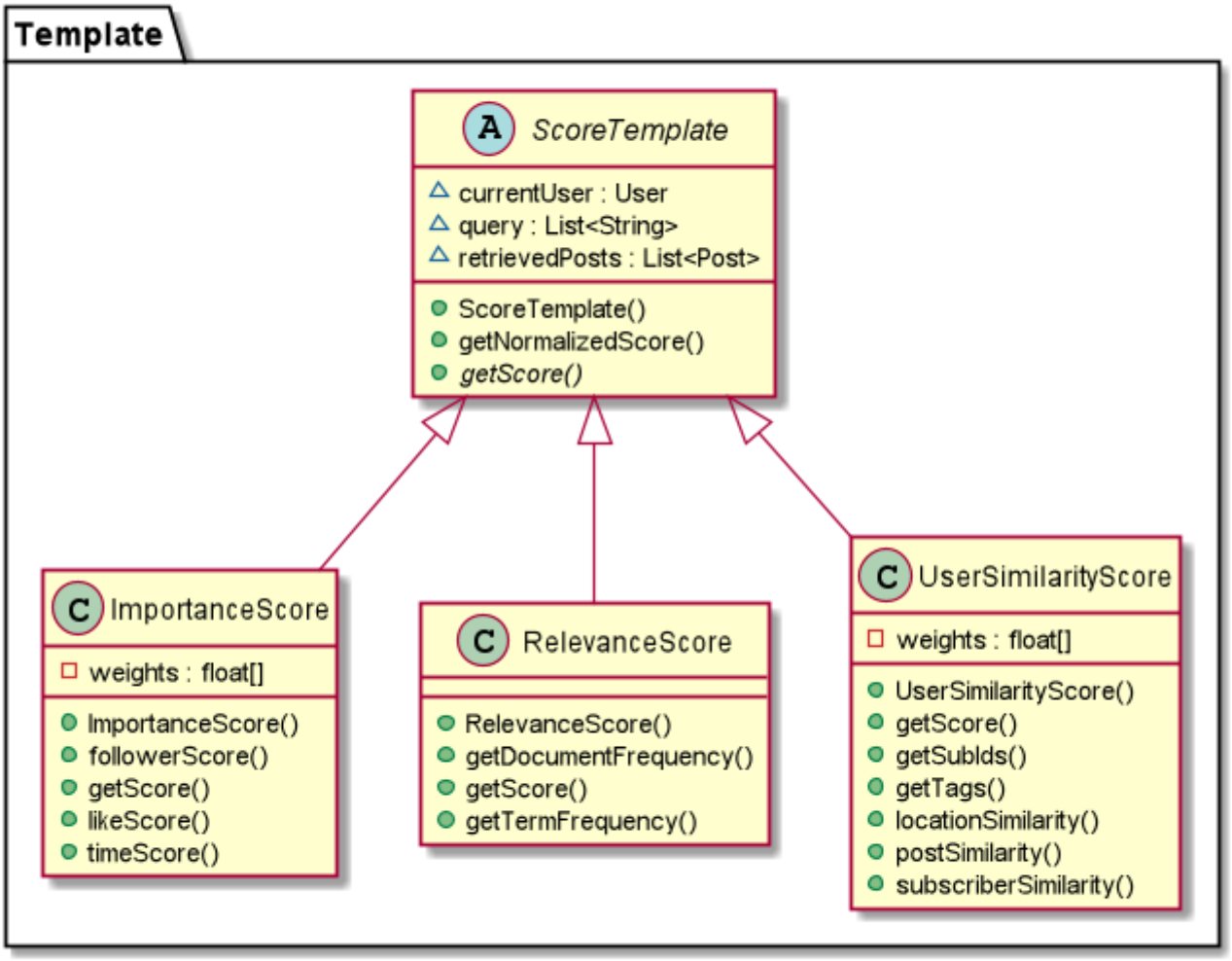
- Location: *CurrentUser.java*
- We used Singleton design pattern in CurrentUser to make sure there must be exactly one instance of the current user.



Template

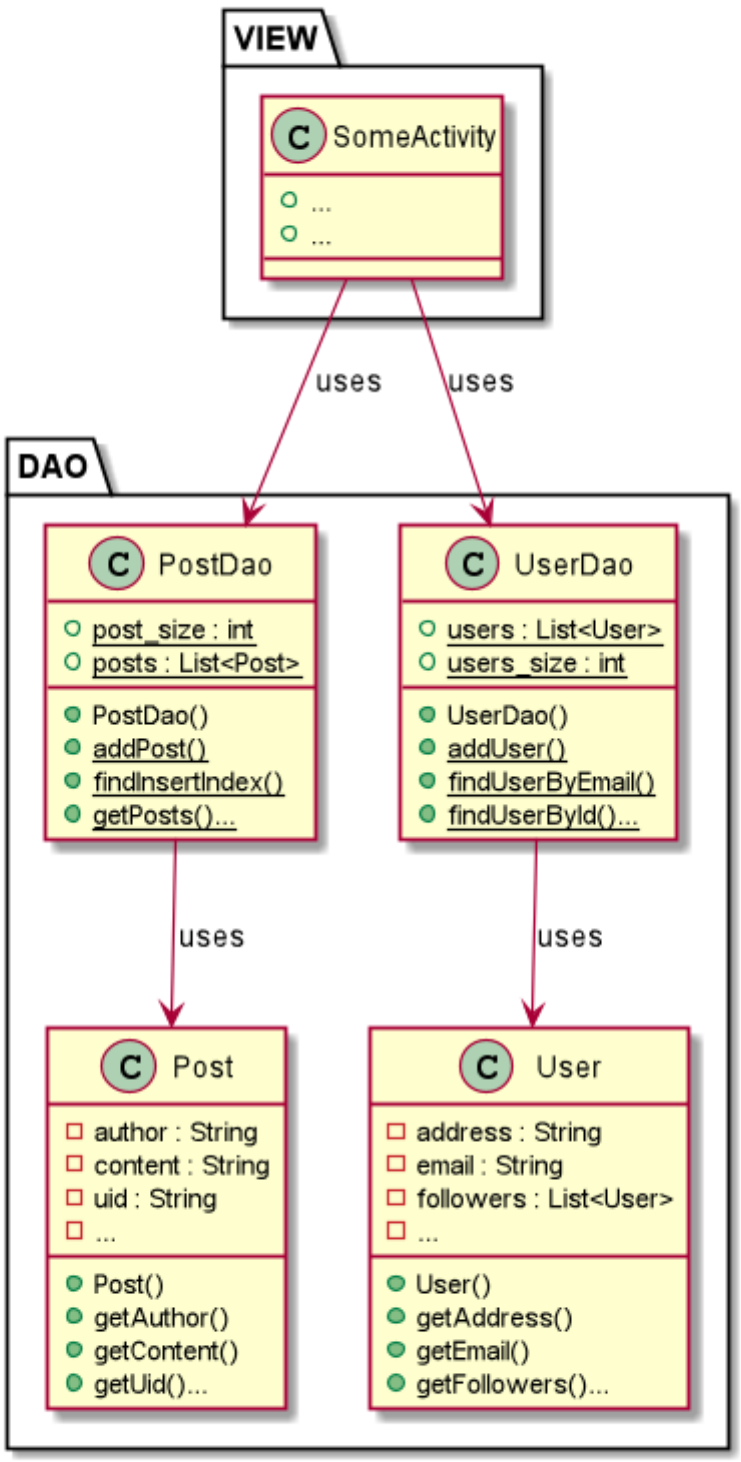
- Location: *ScoreTemplate.java* (abstract class), *RelevanceScore.java* (concrete class), *ImportanceScore.java* (concrete class), *UserSimilarity.java* (concrete class)
- We want to sort the posts based on three criteria. There is a common process (i.e., calculate scores, normalize scores) to scoring the posts but each criterion has its own scoring logic.
- So, we used a score template to define the skeleton of the scoring algotirthm, and let subclasses redefine certain steps of the algotirthm without changing the main structure.

TEMPLATE's Class Diagram



DAO

- Location: *UserDao.java*, *PostDao.java*
- We want to decouple domain logic from persistence mechanisms and avoid exposing details of the data storage. The two DAO classes provided convenient access to user data in whole project.


DAO's Class Diagram
VIEW
SomeActivity
...
...
uses
uses
DAO
PostDao
post_size : int
posts : List<Post>
PostDao()
addPost()
findInsertIndex()
getPosts()...
UserDao
users : List<User>
users_size : int
UserDao()
addUser()
findUserByEmail()
findUserById()...
uses
uses
Post
author : String
content : String
uid : String
...
Post()
getAuthor()
getContent()
getUid()...
User
address : String
email : String
followers : List<User>
...
User()
getAddress()
getEmail()
getFollowers()...
PlantUML diagram generated by SketchIt! (https://bitbucket.org/pmesmeur/sketch.it)
For more information about this tool, please contact philippe.mesmeur@gmail.com

Grammars

Production Rules

<exp> ::= <term> | <term> '|' <exp>

<term> ::= <factor> | <factor> '&' <term>

<factor> ::= <tag> | '(' <exp> ')'

According to this gramma, we can parse *AND* and *OR* operations. For example, suppose we have the expression **condition1 & condition2 | condition3**, human will process it as **(condition1 & condition2) | condition3**, so does our gramma will do. Another example is **condition1 & (condition2 | condition3 & condition4)**, human will process it as **condition1 & (condition2 | (condition3 & condition4))**, so does our gramma will do. In application, the advantages of our gramma is to filter tag of post with *AND* and *OR* operation. It takes tag as its condition and only return post that satified given condition in expression.

Tokenizer and Parsers

Generally speaking, the advantages of our design is we to search multiple tags at once and this is also the design approach of our gramma. Operations *AND* and *OR* can be aplied on multiple tags search, that is, the intersection and union of single search result. By search *#tag1&#tag2*, the intersection of individual search result of *#tag1* and *#tag2* will be returned. By search *#tag1|#tag2*, the union of of individual search result of *#tag1* and *#tag2* will be returned. And of course the operators can be freely used to create more expressions. By default, the precedence of *AND* operation is greater then *OR* operation. And precedence can be changed parentheses *LBRA* and *RBRA*.

Surpise Item

Ranking algorithm

Based on the *Pariser talk*, we decided to sort the posts by three criteria: relecvance; importance; user similarity.

- **Relevance**

  The class *RelevanceScore* evaluates the relecance of the query and the retrieved posts by calculating tf-idf scores. The score of post d given query q = (t_1, t_2, ..., t_m) is:

$$score_{tfidf}(d,q) = \sum_{i=1}^{m} tfidf_{t_i,d}$$

$$tfidf_{t,d} = tf_{t,d} \times idf_t$$

$$idf_t = log\frac{N}{1+df_t}$$

df_t = the number of posts in all posts that contain a term t

$tf_{t,d}$ = the number of occurrences of tag t in post d

N = the total number of posts

We expect the posts containing more "rare words" that appeared in the query to be more relevant. The rare word here refers to the word that does not exist in many posts.

- **Importance**

  The class *ImportanceScore* evaluates the importance of the retrieved posts by calculating the post time, post likes, and the user's followers:

$$score_{\text{importance}} = w_{\text{time}} \times score_{\text{time}} + w_{\text{follower}} \times score_{\text{follower}} + w_{\text{like}} \times score_{\text{like}}$$

$$score_{\text{time}} = \frac{1}{1 + \#\text{days between query and post}}$$

$$score_{\text{follower}} = min(\frac{1}{4} log_{10}(\#\text{followers of post}), 0)$$

$$score_{\text{like}} = min(\frac{1}{4} log_{10}(\#\text{likes of post}), 0)$$

$w_{\text{time}}, w_{\text{follower}}, w_{\text{like}}$ are initially set to be $0.7, 0.2, 0.1$

  We expect the posts with more likes, the posts sent by users with more followers, and the latest posts to be more important to society, and thus have higher ranking scores.

- **User Similarity**

  The class *UserSimilarityScore* evaluates the similarity of the current user and the senders of the posts by evaluating their subscribers, posts, and addresses:

$$score_{\text{similarity}} = w_{\text{subscriber}} \times score_{\text{subscriber}} + w_{\text{post}} \times score_{\text{post}} + w_{\text{address}} \times score_{\text{address}}$$

$$score_{\text{subscriber}} = \frac{1}{1 + \#\text{common subscribers}}$$

$$score_{\text{post}} = \frac{1}{1 + \#\text{common tags from past posts}}$$

$$score_{\text{address}} = 1 \text{ if current user and post sender live in same area, } = 0 \text{ otherwise}$$

$$w_{\text{subscriber}}, w_{\text{post}}, w_{\text{address}} \text{are initially set to be } 0.5, 0.4, 0.1$$

The post sent by the user who is considered as more 'similar' with the current user will have a lower ranking score.We expect to use this algotithm to offset some negative effects of the *'Filter Bubbles'*.

- **Overall Score**

$$score = w_{\text{tfidf}} \times score_{\text{tfidf}} + w_{\text{importance}} \times score_{\text{importance}} + w_{\text{similarity}} \times score_{\text{similarity}}$$

$$w_{\text{tfidf}}, w_{\text{importance}}, w_{\text{similarity}} \text{are initially set to be } 0.6, 0.2, 0.2$$

All of the weights introduced in this algorithm were intuitively chose by our group members. In practice we should use machine learning techniques to train the best weights.

We expect that the posts having strong relevance with the query, the posts which are important to society, and the posts sent by the users who are "different" with the user, to have better ranking in the search results presenting.

More and more intelligent recommendation systems create the *'Filter Bubbles'* for users. We intend to "break" the bubble by implementing the ranking algorithm from users similarity dimension. The score of the posts created by the users who are very "similar" with you will be scaled down. People will have more chances to receive different viewpoints from social media.

Simple personalisation

In order to create unique personalization features, we recorded the location of the user while recording the user login data. Based on the user's location, we added a search function, which is the city search function. The local search function searches for all posts in the same city based on the current user's location. This will make it easier for users to see what's going on around them.

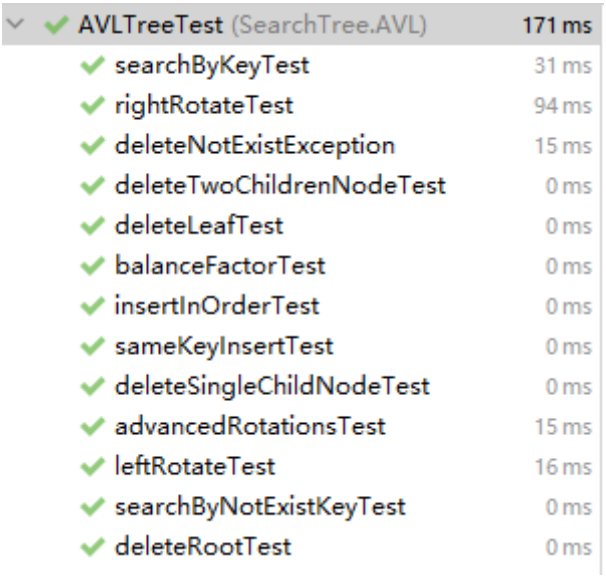
Summary of Known Errors and Bugs

1. *Scroll Down Problem:*
    1. Problem
        - *Click like/dislike button will make it color, but scroll down posts will make button back to gray.*
        - *After delete all posts in a page, this page can not scroll down any more, util refresh to load new posts.*
    2. Possible Reason:
        - *By default the like/dislike button are gray until user click, but scroll down will refresh the view with more posts.*
        - *After delete all posts, the area of ListView become almost zero, user is not able to interact with the component.*
2. *Initialize App Problem:*
    1. Problem
        - *When a simulator is used for long time, WeTok app may fail to open, the error message is: Fatal signal 11 (SIGSEGV)...*
    2. Possible Reason and Solution:
        - *It might because the memory or storage usage.You may try to wipe data or use another simulator.*
3. *Search Problem:*
    1. Problem
        - *After search post by tags, it need to click twice back arrow to return previous page.*
    2. Possible Reason and Solution:
        - *It's because the page when you're typing tags is a individual page. You may just click twice to go back.*
4. *Delete Post Problem:*
    1. Problem
        - *Posts cannot be deleted in other user's personal profile.*
    2. Possible Reason and Solution:
        - *Originnally it will break the app, so we disable the function in other's profile page. You may subscribe him/her, then delete their posts in Focus page.*

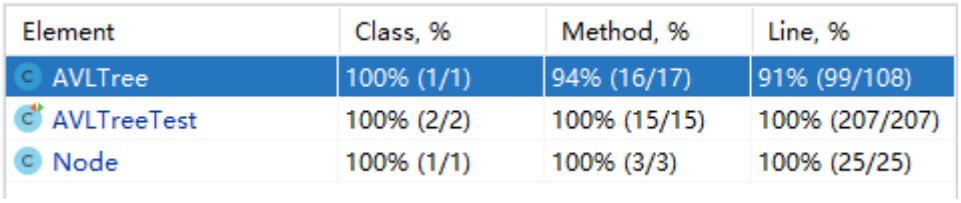
Testing Summary

AVL Tree Testings

Number of test cases: 13



Code coverage:



Element	Class, %	Method, %	Line, %
AVLTree	100% (1/1)	94% (16/17)	91% (99/108)
AVLTreeTest	100% (2/2)	100% (15/15)	100% (207/207)
Node	100% (1/1)	100% (3/3)	100% (25/25)

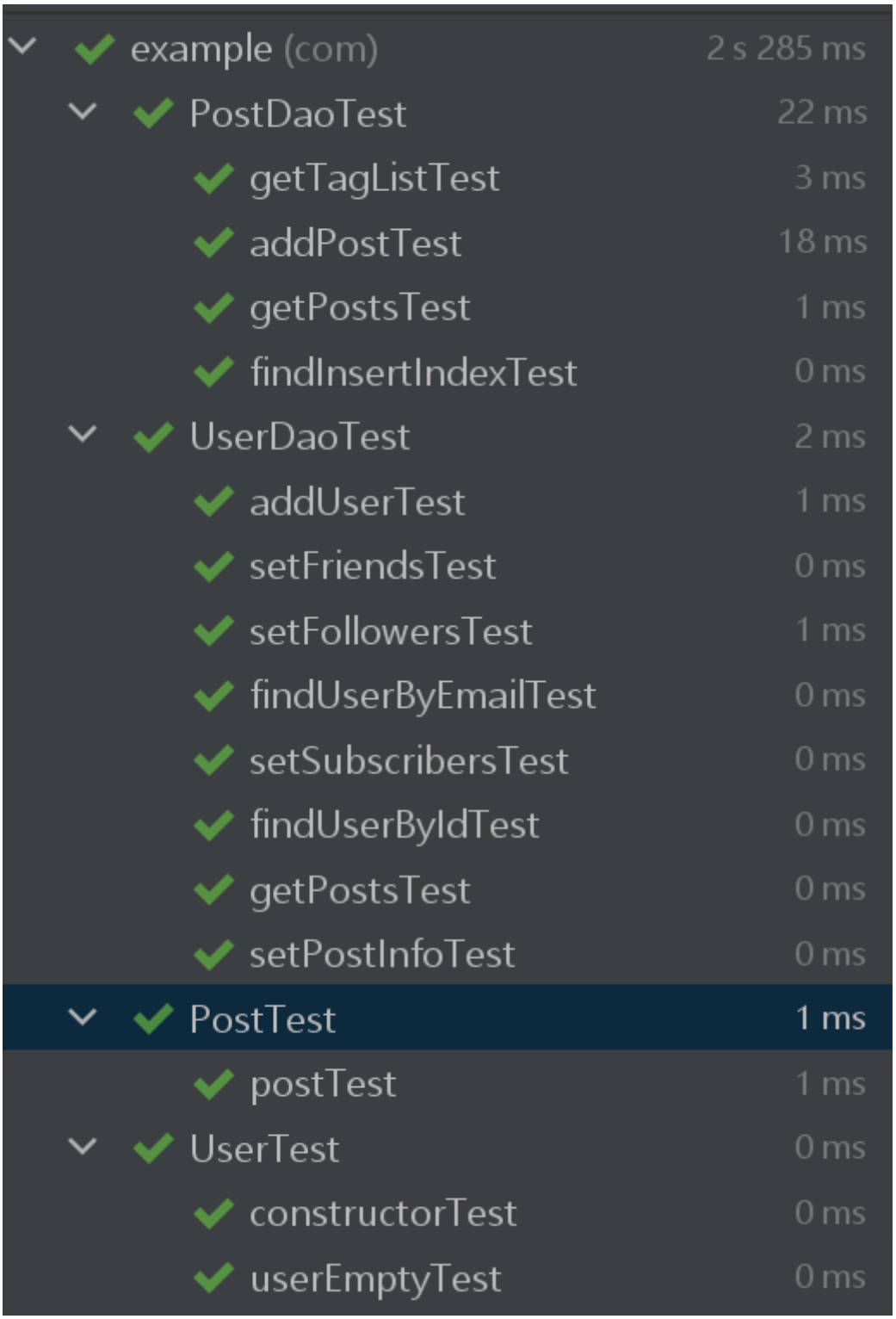
Types of tests created:

1. *sameKeyInsertTest* : test whether the AVL tree can insert multiple values into one nodes
2. *insertInOrderTest* : test whether the AVL tree can correctly insert nodes in order
3. *leftRotateTest* : test whether the AVL tree can correctly left rotate
4. *rightRotateTest* : test whether the AVL tree can correctly right rotate
5. *balanceFactorTest* : test whether the class can generate correct balance factors
6. *advancedRotationsTest* : test whether the AVL tree can handle complex rotations
7. *deleteNotExistException* : test whether the AVL tree can throw an exception when deleting a node that not exists
8. *deleteLeafTest* : test whether the AVL tree can delete a leaf node
9. *deleteSingleChildNodeTest* : test whether the AVL tree can correctly delete a node with one child
10. *deleteTwoChildrenNodeTest* : test whether the AVL tree can correctly delete a node with two children
11. *deleteRootTest* : test whether the AVL tree can correctly delete the root node
12. *searchByKeyTest* : test whether the AVL tree can correctly find the node based on the key

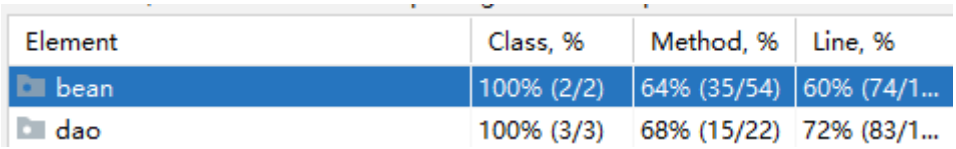
13. *searchByNotExistKeyTest* : test whether the AVL tree returns null when searching a key that not exists

Dao and Bean Testings

Number of test cases: 14



Code coverage:



Element	Class, %	Method, %	Line, %
bean	100% (2/2)	64% (35/54)	60% (74/1...
dao	100% (3/3)	68% (15/22)	72% (83/1...

Tests created for Bean:

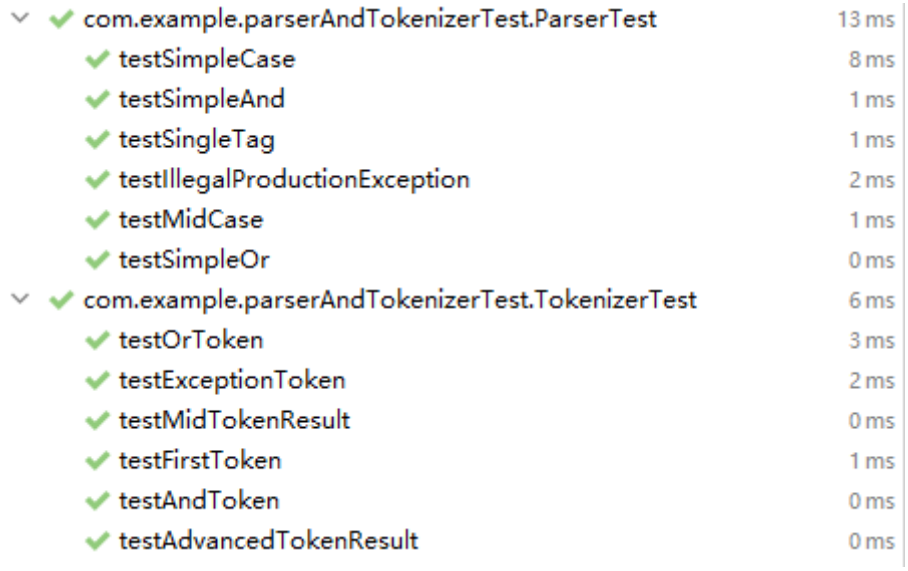
1. *userEmptyTest* : test whether all the setter and getter in User class are valid.
2. *constructorTest* : test whether the constructor in User class is valid.

Tests created for Dao:

1. *getTagListTest* : test whether the PostDao can get all the tags in posts.
2. *findInsertIndexTest* : test whether the PostDao can find the right place to insert post.
3. *addPostTest* : test whether the PostDao can add a post to database.
4. *getPostsTest* : test whether the PostDao can get all the posts in database.
5. *findUserByEmailTest* : test whether the UserDao can find the right user by it's email.
6. *findUserByIdTest* : test whether the UserDao can find the right user by it's user id.
7. *setFriendsTest* : test whether the UserDao can set friends for corresponding user.
8. *setFollowersTest* : test whether the UserDao can set followers for corresponding user.
9. *setSubscribersTest* test whether the UserDao can set subscribers for corresponding user.
10. *getPostsTest* : test whether the UserDao can get all the posts of each user.
11. *setPostInfoTest* : test whether the UserDao can instantiate each user's post.
12. *addUserTest* : test whether the UserDao can add user to database.

Parser and Tokenizer Tests

Number of test cases: 12



Code coverage:

100% classes, 93% lines covered in package 'com.example.wetok.parserAndTok.

Element	Class, %	Method, %	Line, %
AndExp	100% (1/1)	100% (4/4)	100% (13/13)
Exp	100% (1/1)	100% (0/0)	100% (1/1)
OrExp	100% (1/1)	100% (4/4)	100% (13/13)
Parser	100% (2/2)	100% (5/5)	93% (69/74)
TagExp	100% (1/1)	100% (3/3)	100% (8/8)
Token	100% (3/3)	85% (6/7)	70% (12/17)
Tokenizer	100% (1/1)	100% (4/4)	100% (30/30)

Tests created for Tokenizer:

1. *testAndToken* : test whether *AND* token can be recogize.
2. *testOrToken* : test whether *OR* token can be recogize.
3. *testFirstToken* : test whether first *LBRA* token can be recogize.
4. *testMidTokenResult* : test whether *TAG* and *OR* token at middle can be recogize.
5. *testAdvancedTokenResult* : test whether tokens from *(#weekend | #mood) & #time* can be tokenized.
6. *testExceptionToken* : test whether tokenizer throw exceptions as expected.

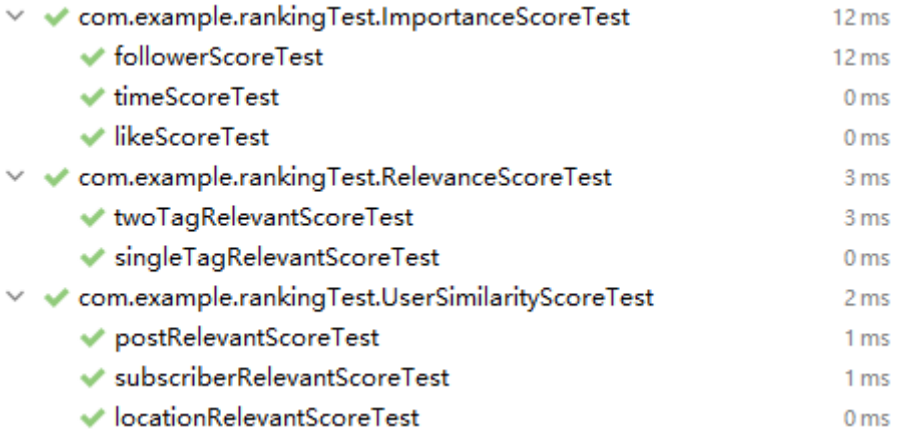
Tests created for Parser:

1. *testSingleTag* : test whether single-tag-search is correct.
2. *testSimpleAnd* : test whether search tag with *AND* operation is correct.
3. *testSimpleOr* : test whether search tag with *OR* operation is correct.

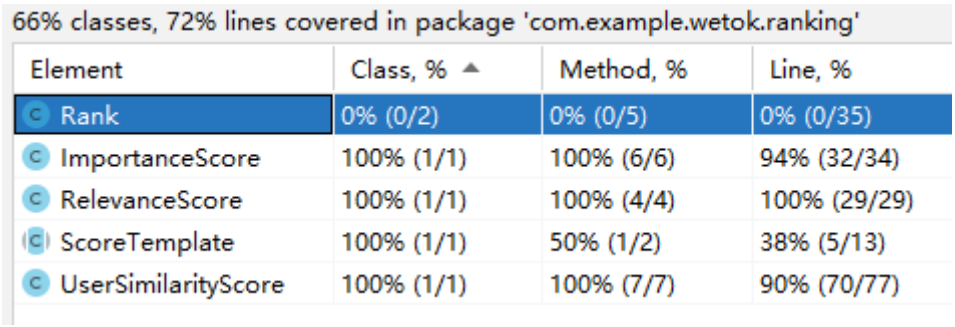
4. *testSimpleCase* : test whether search tag with multiple *AND* operations is correct.
5. *testMidCase* : test whether search tag with *AND* and *OR* operations is correct.
6. *testIllegalProductionException* : test whether parser throw exceptions as expected.

Ranking Tests

Number of test cases: 8



Code coverage:



66% classes, 72% lines covered in package 'com.example.wetok.ranking'

Element	Class, %	Method, %	Line, %
Rank	0% (0/2)	0% (0/5)	0% (0/35)
ImportanceScore	100% (1/1)	100% (6/6)	94% (32/34)
RelevanceScore	100% (1/1)	100% (4/4)	100% (29/29)
ScoreTemplate	100% (1/1)	50% (1/2)	38% (5/13)
UserSimilarityScore	100% (1/1)	100% (7/7)	90% (70/77)

Tests created for Ranking Tests:

1. ImportanceScoreTest: 3 tests for 3 component of calculating importance score: time, like, follower.
2. RelevanceScoreTest: 2 tests for 1 tag relevent and 2 tags relevant.
3. UserSimilarityScoreTest: 3 tests for 3 component of calculating user similarity score: location, subscriber, post.

Implemented Features

Improved Search

1. *Search functionality can handle partially valid and invalid search queries. (medium)*

UI Design and Testing

1. *UI must have portrait and landscape layout variants as well as support for different screen sizes. Simply using Android studio's automated support for orientation and screen sizes and or creating support without effort to make them look reasonable will net you zero marks. (easy)*

Greater Data Usage, Handling and Sophistication

1. *User profile activity containing a media file (image, animation (e.g. gif), video). (easy)*
2. *Deletion method of either a Red-Black Tree and or AVL tree data structure. The deletion of nodes must serve a purpose within your application (e.g. deleting posts). (hard)*

User Interactivity

1. *The ability to micro-interact with 'posts' (e.g. like, report, etc.) [stored in-memory]. (easy)*
2. *The ability for users to 'follow' other users. There must be an adjustment to either the user's timeline in relation to their following users or a section specifically dedicated to posts by followed users. [stored in-memory] (medium)*

User Interactivity

1. *Use Firebase to implement user Authentication/Authorisation. (easy)*

Suprise Features

1. *how to sort items returned for a given search and/or in your timeline (ranking algorithm)*
2. *logging user activity to improve search results and/or timeline creation (simple personalisation)*

Team Meetings

- *Team Meeting 1*
- *Team Meeting 2*
- *Team Meeting 3*
- *Team Meeting 4*