

Code Listing for Early Rumour Detection in Social Media using Machine Learning

PSG 16 – DR ZHIWEI LIN – BENG SOFTWARE ENGINEERING

Thomas Franklin B00607399

April 2018

[1. Code directory listings: 3](#_Toc512264149)

[2. api/src/api/main/java/twitter/classification/api code listings 12](#_Toc512264150)

[application 12](#_Toc512264151)

[binder 12](#_Toc512264152)

[client 14](#_Toc512264153)

[persist/jdbc 16](#_Toc512264154)

[models 26](#_Toc512264155)

[queries 31](#_Toc512264156)

[resource 41](#_Toc512264157)

[service 46](#_Toc512264158)

[wordclouds 55](#_Toc512264159)

[3. classifier/src/main/java/twitter/classification/classifier code listings 55](#_Toc512264160)

[application 55](#_Toc512264161)

[binder 56](#_Toc512264162)

[factory 57](#_Toc512264163)

[classification 57](#_Toc512264164)

[helper 58](#_Toc512264165)

[mallet 59](#_Toc512264166)

[classifier 59](#_Toc512264167)

[pipes 62](#_Toc512264168)

[persist.jdbc 62](#_Toc512264169)

[queries 65](#_Toc512264170)

[resource 66](#_Toc512264171)

[service 67](#_Toc512264172)

[mallet 70](#_Toc512264173)

[weka 71](#_Toc512264174)

[weka 72](#_Toc512264175)

[classifier 72](#_Toc512264176)

[converter 73](#_Toc512264177)

[dataset 74](#_Toc512264178)

[filter 74](#_Toc512264179)

[stopwords 74](#_Toc512264180)

[4. common/src/main/java/twitter/classification/common code listings 75](#_Toc512264181)

[config 75](#_Toc512264182)

[exceptions 75](#_Toc512264183)

[models 76](#_Toc512264184)

[persist 87](#_Toc512264185)

[jdbc 91](#_Toc512264186)

[queries 91](#_Toc512264187)

[utils 92](#_Toc512264188)

[system 93](#_Toc512264189)

[binder 94](#_Toc512264190)

[factory 95](#_Toc512264191)

[helper 95](#_Toc512264192)

[tweetdetails 96](#_Toc512264193)

[model 96](#_Toc512264194)

[processing 99](#_Toc512264195)

[5. db schema code listings 100](#_Toc512264196)

[6. frontend/src/main/java/twitter/classification/web code listings 101](#_Toc512264197)

[application 101](#_Toc512264198)

[binder 102](#_Toc512264199)

[clients 103](#_Toc512264200)

[render 108](#_Toc512264201)

[resource 109](#_Toc512264202)

[exceptions 114](#_Toc512264203)

[7. pre-processor/src/main/java/twitter/classification/preprocessor code listings 115](#_Toc512264204)

[application 115](#_Toc512264205)

[binder 115](#_Toc512264206)

[factory 116](#_Toc512264207)

[client 116](#_Toc512264208)

[resource 117](#_Toc512264209)

[8. queue-reader/src/main/java/queuereader code listings 119](#_Toc512264210)

[application 119](#_Toc512264211)

[exceptions 119](#_Toc512264212)

[consumer 119](#_Toc512264213)

[module 120](#_Toc512264214)

[reader 121](#_Toc512264215)

[tweetdetails 122](#_Toc512264216)

[9. stream/src/main/java/twitter/classification/stream code listings 122](#_Toc512264217)

[application 122](#_Toc512264218)

[binder 123](#_Toc512264219)

[factory 123](#_Toc512264220)

[listener 125](#_Toc512264221)

[resource 126](#_Toc512264222)

[10. HTML and JS files 127](#_Toc512264223)

[master.hbs 127](#_Toc512264224)

[no-result.hbs 129](#_Toc512264225)

[search.hbs 129](#_Toc512264226)

[users.hbs 130](#_Toc512264227)

[dashboard.hbs 132](#_Toc512264228)

[hashtags.hbs 133](#_Toc512264229)

[partials/visualtisation-board.hbs 135](#_Toc512264230)

[exceptions/ 136](#_Toc512264231)

[exceptions.hbs 136](#_Toc512264232)

[not-found.hbs 136](#_Toc512264233)

[hashtags.js 136](#_Toc512264234)

[navigation.js 139](#_Toc512264235)

[search.js 140](#_Toc512264236)

[users.js 143](#_Toc512264237)

[11. Excluded files 146](#_Toc512264238)

# Code directory listings:

.

├── README.md

├── Tweet-Classification.iml

├── api

│   ├── api.iml

│   ├── build.gradle

│   ├── gradle

│   │   └── wrapper

│   │   ├── gradle-wrapper.jar

│   │   └── gradle-wrapper.properties

│   ├── gradlew

│   ├── gradlew.bat

│   └── src

│   ├── integration-test

│   │   └── java

│   │   └── twitter

│   │   └── classification

│   │   └── api

│   │   ├── integration

│   │   │   └── DbIntegrationHelper.java

│   │   ├── service

│   │   │   └── TimeLineTweetsServiceTest.java

│   │   └── util

│   │   └── RandomDataUtil.java

│   ├── main

│   │   ├── java

│   │   │   └── twitter

│   │   │   └── classification

│   │   │   └── api

│   │   │   ├── application

│   │   │   │   ├── WebApplication.java

│   │   │   │   └── binder

│   │   │   │   └── ServicesBinder.java

│   │   │   ├── client

│   │   │   │   ├── ClassifierStatusClient.java

│   │   │   │   ├── PreProcessorStatusClient.java

│   │   │   │   └── TwitterStreamClient.java

│   │   │   ├── persist

│   │   │   │   └── jdbc

│   │   │   │   ├── PaginatedHashtagTweetsDao.java

│   │   │   │   ├── PaginatedSearchTermTweetsDao.java

│   │   │   │   ├── PaginatedUserTweetsDao.java

│   │   │   │   ├── SelectDashBoardOverviewValuesDao.java

│   │   │   │   ├── SelectSearchTermClassificationCountDao.java

│   │   │   │   ├── SelectTopHashtagsClassificationCountDao.java

│   │   │   │   ├── SelectTopUsersClassificationCountDao.java

│   │   │   │   ├── SuggestedSearchResultsDao.java

│   │   │   │   ├── TestDatabaseConnectionDao.java

│   │   │   │   ├── TimeLineForHashtagsDao.java

│   │   │   │   ├── TimeLineForSearchTermDao.java

│   │   │   │   ├── TimeLineForUsersDao.java

│   │   │   │   ├── TweetsForHashtagsDao.java

│   │   │   │   ├── TweetsForSearchTermDao.java

│   │   │   │   ├── TweetsForUsersDao.java

│   │   │   │   ├── models

│   │   │   │   │   ├── ClassificationCountModel.java

│   │   │   │   │   ├── DashBoardOverviewModel.java

│   │   │   │   │   ├── PaginatedTweetsModel.java

│   │   │   │   │   ├── ProcessedHashtagTweetsForWordCloudModel.java

│   │   │   │   │   ├── ProcessedTweetsForWordCloudModel.java

│   │   │   │   │   ├── ProcessedUserTweetsForWordCloudModel.java

│   │   │   │   │   ├── SuggestedSearchResultsModel.java

│   │   │   │   │   ├── TimeLineForTweetsModel.java

│   │   │   │   │   ├── TopHashtagsClassificationModel.java

│   │   │   │   │   └── TopUsersClassificationModel.java

│   │   │   │   └── queries

│   │   │   │   ├── SelectDashBoardOverviewValuesDbQuery.java

│   │   │   │   ├── SelectHashtagTweetsDbQuery.java

│   │   │   │   ├── SelectSearchTermClassificationCountDbQuery.java

│   │   │   │   ├── SelectSearchTermTweetsDbQuery.java

│   │   │   │   ├── SelectTopHashtagsClassificationCountDbQuery.java

│   │   │   │   ├── SelectTopUsersClassificationCountDbQuery.java

│   │   │   │   ├── SelectUserTweetsDbQuery.java

│   │   │   │   ├── SuggestedSearchResultsDbQuery.java

│   │   │   │   ├── TimeLineForHashtagsDbQuery.java

│   │   │   │   ├── TimeLineForSearchTermDbQuery.java

│   │   │   │   ├── TimeLineForUsersDbQuery.java

│   │   │   │   ├── TweetsForHashtagWordCloudDbQuery.java

│   │   │   │   ├── TweetsForSearchTermWordCloudDbQuery.java

│   │   │   │   └── TweetsForUserWordCloudDbQuery.java

│   │   │   ├── resource

│   │   │   │   ├── DashBoardOverviewDataResource.java

│   │   │   │   ├── HashtagsResource.java

│   │   │   │   ├── SearchResource.java

│   │   │   │   ├── TopHashTagsResource.java

│   │   │   │   ├── TopUsersResource.java

│   │   │   │   └── UsersResource.java

│   │   │   ├── service

│   │   │   │   ├── DashBoardOverviewService.java

│   │   │   │   ├── DashBoardServicesStatusService.java

│   │   │   │   ├── HashtagResultsService.java

│   │   │   │   ├── PaginatedResultsService.java

│   │   │   │   ├── SearchTermResultService.java

│   │   │   │   ├── SuggestedSearchResultsService.java

│   │   │   │   ├── TopHashTagResultService.java

│   │   │   │   ├── TopUserResultService.java

│   │   │   │   └── UserResultsService.java

│   │   │   └── wordclouds

│   │   │   └── WordCloudCreationService.java

│   │   ├── resources

│   │   │   └── log4j.properties

│   │   └── webapp

│   │   └── WEB-INF

│   │   └── web.xml

│   └── test

│   └── java

│   └── twitter

│   └── classification

│   └── api

│   └── resource

│   └── HashtagsResourceTest.java

├── build.gradle

├── classifier

│   ├── build.gradle

│   ├── classifier.iml

│   ├── gradle

│   │   └── wrapper

│   │   ├── gradle-wrapper.jar

│   │   └── gradle-wrapper.properties

│   ├── gradlew

│   ├── gradlew.bat

│   └── src

│   ├── integration-test

│   │   └── java

│   │   └── twitter

│   │   └── classification

│   │   └── classifier

│   │   ├── integration

│   │   │   └── DbIntegrationHelper.java

│   │   ├── service

│   │   │   └── HandleProcessedTweetServiceTest.java

│   │   └── util

│   │   └── RandomDataUtil.java

│   ├── main

│   │   ├── java

│   │   │   └── twitter

│   │   │   └── classification

│   │   │   └── classifier

│   │   │   ├── application

│   │   │   │   ├── WebApplication.java

│   │   │   │   └── binder

│   │   │   │   ├── ServicesBinder.java

│   │   │   │   └── factory

│   │   │   │   ├── ClassifierFactory.java

│   │   │   │   └── VerificationClassifierFactory.java

│   │   │   ├── classification

│   │   │   │   └── LabelWeight.java

│   │   │   ├── helper

│   │   │   │   ├── ClassificationCodeFromValue.java

│   │   │   │   └── ClassificationFromVerificationCheck.java

│   │   │   ├── mallet

│   │   │   │   ├── classifier

│   │   │   │   │   └── TrainClassifier.java

│   │   │   │   └── pipes

│   │   │   │   └── FeaturePipes.java

│   │   │   ├── persist

│   │   │   │   └── jdbc

│   │   │   │   ├── InsertHashtagTweetClassificationDao.java

│   │   │   │   ├── InsertHashtagsDao.java

│   │   │   │   ├── InsertTweetsDao.java

│   │   │   │   ├── InsertUserTweetClassificationDao.java

│   │   │   │   ├── InsertUsersDao.java

│   │   │   │   └── queries

│   │   │   │   ├── InsertHashtagTweetClassificationDbQuery.java

│   │   │   │   ├── InsertHashtagsDbQuery.java

│   │   │   │   ├── InsertTweetsDbQuery.java

│   │   │   │   ├── InsertUserTweetClassificationDbQuery.java

│   │   │   │   └── InsertUsersDbQuery.java

│   │   │   ├── resource

│   │   │   │   └── ClassificationResource.java

│   │   │   ├── service

│   │   │   │   ├── HandleProcessedTweetService.java

│   │   │   │   ├── InsertHashtagEntitiesService.java

│   │   │   │   ├── InsertTweetsService.java

│   │   │   │   ├── InsertUserTweetClassificationService.java

│   │   │   │   ├── InsertUsersService.java

│   │   │   │   ├── TrainedClassifier.java

│   │   │   │   ├── VerificationClassifier.java

│   │   │   │   ├── mallet

│   │   │   │   │   └── NaiveBayesClassifier.java

│   │   │   │   └── weka

│   │   │   │   └── NaiveBayesClassifier.java

│   │   │   └── weka

│   │   │   ├── classifier

│   │   │   │   └── NaiveBayesClassifier.java

│   │   │   ├── converter

│   │   │   │   └── WekaInstanceFromString.java

│   │   │   ├── filter

│   │   │   │   └── StringToWordVectorFilter.java

│   │   │   └── stopwords

│   │   │   └── StopWordsHandler.java

│   │   ├── resources

│   │   │   ├── log4j.properties

│   │   │   └── stopwords

│   │   │   └── stopwords.txt

│   │   └── webapp

│   │   └── WEB-INF

│   │   └── web.xml

│   └── test

│   ├── java

│   │   └── twitter

│   │   └── classification

│   │   └── classifier

│   │   ├── helper

│   │   │   ├── ClassificationCodeFromValueTest.java

│   │   │   └── ClassificationFromVerificationCheckTest.java

│   │   ├── service

│   │   │   └── mallet

│   │   │   ├── MaxEntClassifierTest.java

│   │   │   └── NaiveBayesClassifierTest.java

│   │   └── weka

│   │   └── classifier

│   │   └── NaiveBayesClassifierTest.java

│   └── resources

├── common

│   ├── build.gradle

│   ├── common.iml

│   ├── gradle

│   │   └── wrapper

│   │   ├── gradle-wrapper.jar

│   │   └── gradle-wrapper.properties

│   ├── gradlew

│   ├── gradlew.bat

│   └── src

│   ├── main

│   │   └── java

│   │   └── twitter

│   │   └── classification

│   │   └── common

│   │   ├── config

│   │   │   └── ConfigurationKey.java

│   │   ├── exceptions

│   │   │   ├── ProcessingClientException.java

│   │   │   └── ProcessingResponseException.java

│   │   ├── models

│   │   │   ├── ClassificationValueForTweets.java

│   │   │   ├── ClassifierStatusResponse.java

│   │   │   ├── DashBoardOverviewResponse.java

│   │   │   ├── DashBoardServiceStatusResponse.java

│   │   │   ├── HashtagResults.java

│   │   │   ├── PreProcessorStatusResponse.java

│   │   │   ├── SearchResultsResponse.java

│   │   │   ├── ServiceItem.java

│   │   │   ├── SuggestedSearchResult.java

│   │   │   ├── SuggestedSearchTermsResponse.java

│   │   │   ├── TimeLineForTweets.java

│   │   │   ├── TopHashtagsResponse.java

│   │   │   ├── TopUsersResponse.java

│   │   │   ├── TwitterStreamResponse.java

│   │   │   └── UserResults.java

│   │   ├── persist

│   │   │   ├── Column.java

│   │   │   ├── ConnectionFactory.java

│   │   │   ├── ConnectionManager.java

│   │   │   ├── DbConnection.java

│   │   │   ├── DbConnectionResolver.java

│   │   │   ├── Entity.java

│   │   │   ├── ResultSetMapper.java

│   │   │   └── jdbc

│   │   │   ├── MySqlConnectionFactory.java

│   │   │   ├── queries

│   │   │   │   └── DbQuery.java

│   │   │   └── utils

│   │   │   └── DbQueryRunner.java

│   │   ├── system

│   │   │   ├── ConfigurationVariable.java

│   │   │   ├── binder

│   │   │   │   ├── ConfigurationVariableBinder.java

│   │   │   │   └── factory

│   │   │   │   └── BaseFactory.java

│   │   │   └── helper

│   │   │   ├── ConfigurationVariableParam.java

│   │   │   └── FileVariables.java

│   │   └── tweetdetails

│   │   ├── model

│   │   │   ├── ClassificationModel.java

│   │   │   ├── PreProcessedItem.java

│   │   │   ├── ProcessedStatusResponse.java

│   │   │   └── ProcessedTweetModel.java

│   │   └── processing

│   │   ├── ProcessResponse.java

│   │   └── TweetBodyProcessor.java

│   └── test

│   └── java

│   └── twitter

│   └── classification

│   └── common

│   └── tweetdetails

│   └── processing

│   └── TweetBodyProcessorTest.java

├── docker-compose.yml

├── frontend

│   ├── build.gradle

│   ├── frontend.iml

│   ├── gradle

│   │   └── wrapper

│   │   ├── gradle-wrapper.jar

│   │   └── gradle-wrapper.properties

│   ├── gradlew

│   ├── gradlew.bat

│   └── src

│   ├── main

│   │   ├── java

│   │   │   └── twitter

│   │   │   └── classification

│   │   │   └── web

│   │   │   ├── application

│   │   │   │   ├── WebApplication.java

│   │   │   │   └── binder

│   │   │   │   ├── ClientBinder.java

│   │   │   │   └── TemplateRenderBinder.java

│   │   │   ├── clients

│   │   │   │   ├── AlternativeSearchResultsClient.java

│   │   │   │   ├── DashBoardOverviewClient.java

│   │   │   │   ├── DashBoardServiceStatusClient.java

│   │   │   │   ├── SearchResultsClient.java

│   │   │   │   ├── TopHashTagsResultsClient.java

│   │   │   │   └── TopUsersResultsClient.java

│   │   │   ├── render

│   │   │   │   ├── HandleBarsTemplateRender.java

│   │   │   │   └── TemplateRender.java

│   │   │   └── resource

│   │   │   ├── DashBoardResource.java

│   │   │   ├── HashtagsResource.java

│   │   │   ├── SearchResource.java

│   │   │   ├── UsersResource.java

│   │   │   └── exceptions

│   │   │   ├── InternalServerExceptionMapper.java

│   │   │   └── NotFoundExceptionResourceMapper.java

│   │   ├── resources

│   │   │   ├── log4j.properties

│   │   │   └── templates

│   │   │   ├── dashboard.hbs

│   │   │   ├── exceptions

│   │   │   │   ├── exception.hbs

│   │   │   │   └── not-found.hbs

│   │   │   ├── hashtags.hbs

│   │   │   ├── master.hbs

│   │   │   ├── no-results.hbs

│   │   │   ├── partials

│   │   │   │   └── visualisation-board.hbs

│   │   │   ├── search.hbs

│   │   │   └── users.hbs

│   │   └── webapp

│   │   ├── WEB-INF

│   │   │   └── web.xml

│   │   └── assets

│   │   ├── css

│   │   └── js

│   │   ├── hashtags.js

│   │   ├── navigation.js

│   │   ├── search.js

│   │   └── users.js

│   └── test

├── gradle

│   └── wrapper

│   ├── gradle-wrapper.jar

│   └── gradle-wrapper.properties

├── gradlew

├── gradlew.bat

├── pre-processor

│   ├── build.gradle

│   ├── gradle

│   │   └── wrapper

│   │   ├── gradle-wrapper.jar

│   │   └── gradle-wrapper.properties

│   ├── gradlew

│   ├── gradlew.bat

│   └── src

│   ├── main

│   │   ├── java

│   │   │   └── twitter

│   │   │   └── classification

│   │   │   └── preprocessor

│   │   │   ├── application

│   │   │   │   ├── WebApplication.java

│   │   │   │   └── binder

│   │   │   │   ├── ConfigurationBinder.java

│   │   │   │   └── factory

│   │   │   │   └── ClassificationClientFactory.java

│   │   │   ├── client

│   │   │   │   └── ClassificationClient.java

│   │   │   └── resource

│   │   │   └── ReceiveTweetStatusResource.java

│   │   ├── resources

│   │   │   └── log4j.properties

│   │   └── webapp

│   │   └── WEB-INF

│   │   └── web.xml

│   └── test

│   └── java

├── queue

│   └── DockerFile

├── queue-reader

│   ├── build.gradle

│   ├── gradle

│   │   └── wrapper

│   │   ├── gradle-wrapper.jar

│   │   └── gradle-wrapper.properties

│   ├── gradlew

│   ├── gradlew.bat

│   └── src

│   ├── main

│   │   ├── java

│   │   │   └── twitter

│   │   │   └── classification

│   │   │   └── queuereader

│   │   │   ├── application

│   │   │   │   ├── QueueReaderApplication.java

│   │   │   │   └── exceptions

│   │   │   │   └── IgnoredHashtagEntity.java

│   │   │   ├── consumer

│   │   │   │   └── TweetConsumer.java

│   │   │   ├── module

│   │   │   │   └── ConfigurationModule.java

│   │   │   ├── reader

│   │   │   │   └── QueueReader.java

│   │   │   └── tweetdetails

│   │   │   └── TweetDetailsClient.java

│   │   └── resources

│   │   ├── configuration.txt

│   │   └── log4j.properties

│   └── test

│   └── java

├── settings.gradle

├── stream

│   ├── build.gradle

│   ├── gradle

│   │   └── wrapper

│   │   ├── gradle-wrapper.jar

│   │   └── gradle-wrapper.properties

│   ├── gradlew

│   ├── gradlew.bat

│   ├── src

│   │   ├── main

│   │   │   ├── java

│   │   │   │   └── twitter

│   │   │   │   └── classification

│   │   │   │   └── stream

│   │   │   │   ├── application

│   │   │   │   │   ├── TwitterStream.java

│   │   │   │   │   ├── WebApplication.java

│   │   │   │   │   └── binder

│   │   │   │   │   ├── MessageQueueBinder.java

│   │   │   │   │   ├── TwitterStreamBinder.java

│   │   │   │   │   └── factory

│   │   │   │   │   ├── MessageQueueFactory.java

│   │   │   │   │   └── TwitterStreamFactory.java

│   │   │   │   ├── listener

│   │   │   │   │   └── NewTweetListener.java

│   │   │   │   └── resource

│   │   │   │   └── StreamTweetsResource.java

│   │   │   ├── resources

│   │   │   │   └── log4j.properties

│   │   │   └── webapp

│   │   │   └── WEB-INF

│   │   │   └── web.xml

│   │   └── test

│   │   └── java

│   └── stream.iml

└── tomcat

└── DockerFile

203 directories, 258 files

# api/src/api/main/java/twitter/classification/api code listings

## application

**package** twitter.classification.api.application;  
  
**import** org.glassfish.jersey.server.ResourceConfig;  
  
**import** twitter.classification.api.application.binder.ServicesBinder;  
**import** twitter.classification.common.system.binder.ConfigurationVariableBinder;  
**import** twitter.classification.common.system.helper.FileVariables;  
  
**public class** WebApplication **extends** ResourceConfig {  
  
 */\*\*  
 \* Entry point of the jersey application for the api where it will load the configuration  
 \* values from the text file and register the services which will be required  
 \*/* **public** WebApplication() {  
  
 packages(**"twitter.classification.api.application"**);  
  
 loadConfigurationValues();  
  
 register(**new** ConfigurationVariableBinder());  
 register(**new** ServicesBinder());  
 }  
  
 **private void** loadConfigurationValues() {  
  
 **new** FileVariables().setValuesFromConfigurationFile();  
 }  
}

### binder

**package** twitter.classification.api.application.binder;  
  
**import** javax.inject.Singleton;  
  
**import** org.glassfish.hk2.utilities.binding.AbstractBinder;  
  
**import** twitter.classification.api.client.ClassifierStatusClient;  
**import** twitter.classification.api.client.PreProcessorStatusClient;  
**import** twitter.classification.api.client.TwitterStreamClient;  
**import** twitter.classification.api.persist.jdbc.PaginatedHashtagTweetsDao;  
**import** twitter.classification.api.persist.jdbc.PaginatedSearchTermTweetsDao;  
**import** twitter.classification.api.persist.jdbc.PaginatedUserTweetsDao;  
**import** twitter.classification.api.persist.jdbc.SelectDashBoardOverviewValuesDao;  
**import** twitter.classification.api.persist.jdbc.SelectSearchTermClassificationCountDao;  
**import** twitter.classification.api.persist.jdbc.SelectTopHashtagsClassificationCountDao;  
**import** twitter.classification.api.persist.jdbc.SelectTopUsersClassificationCountDao;  
**import** twitter.classification.api.persist.jdbc.SuggestedSearchResultsDao;  
**import** twitter.classification.api.persist.jdbc.TestDatabaseConnectionDao;  
**import** twitter.classification.api.persist.jdbc.TimeLineForHashtagsDao;  
**import** twitter.classification.api.persist.jdbc.TimeLineForSearchTermDao;  
**import** twitter.classification.api.persist.jdbc.TimeLineForUsersDao;  
**import** twitter.classification.api.persist.jdbc.TweetsForHashtagsDao;  
**import** twitter.classification.api.persist.jdbc.TweetsForSearchTermDao;  
**import** twitter.classification.api.persist.jdbc.TweetsForUsersDao;  
**import** twitter.classification.api.service.DashBoardOverviewService;  
**import** twitter.classification.api.service.DashBoardServicesStatusService;  
**import** twitter.classification.api.service.HashtagResultsService;  
**import** twitter.classification.api.service.SuggestedSearchResultsService;  
**import** twitter.classification.api.service.UserResultsService;  
**import** twitter.classification.api.service.SearchTermResultService;  
**import** twitter.classification.api.service.TopHashTagResultService;  
**import** twitter.classification.api.service.TopUserResultService;  
**import** twitter.classification.common.persist.ConnectionManager;  
**import** twitter.classification.common.persist.DbConnectionResolver;  
  
**public class** ServicesBinder **extends** AbstractBinder {  
  
 */\*\*  
 \* Services binder for the api, which binds the various  
 \* DAOs, Services and clients which are required by the APIs resources  
 \*/* @Override  
 **protected void** configure() {  
  
 bind(ConnectionManager.**class**).to(ConnectionManager.**class**).in(Singleton.**class**);  
  
 bind(TwitterStreamClient.**class**).to(TwitterStreamClient.**class**);  
 bind(ClassifierStatusClient.**class**).to(ClassifierStatusClient.**class**);  
 bind(PreProcessorStatusClient.**class**).to(PreProcessorStatusClient.**class**);  
  
 bind(TestDatabaseConnectionDao.**class**).to(TestDatabaseConnectionDao.**class**);  
 bind(SelectDashBoardOverviewValuesDao.**class**).to(SelectDashBoardOverviewValuesDao.**class**);  
 bind(SelectTopHashtagsClassificationCountDao.**class**).to(SelectTopHashtagsClassificationCountDao.**class**);  
 bind(SelectTopUsersClassificationCountDao.**class**).to(SelectTopUsersClassificationCountDao.**class**);  
 bind(TweetsForHashtagsDao.**class**).to(TweetsForHashtagsDao.**class**);  
 bind(TweetsForUsersDao.**class**).to(TweetsForUsersDao.**class**);  
 bind(PaginatedHashtagTweetsDao.**class**).to(PaginatedHashtagTweetsDao.**class**);  
 bind(PaginatedUserTweetsDao.**class**).to(PaginatedUserTweetsDao.**class**);  
 bind(SelectSearchTermClassificationCountDao.**class**).to(SelectSearchTermClassificationCountDao.**class**);  
 bind(TweetsForSearchTermDao.**class**).to(TweetsForSearchTermDao.**class**);  
 bind(PaginatedSearchTermTweetsDao.**class**).to(PaginatedSearchTermTweetsDao.**class**);  
 bind(TimeLineForHashtagsDao.**class**).to(TimeLineForHashtagsDao.**class**);  
 bind(TimeLineForUsersDao.**class**).to(TimeLineForUsersDao.**class**);  
 bind(TimeLineForSearchTermDao.**class**).to(TimeLineForSearchTermDao.**class**);  
 bind(SuggestedSearchResultsDao.**class**).to(SuggestedSearchResultsDao.**class**);  
  
 bind(DashBoardOverviewService.**class**).to(DashBoardOverviewService.**class**);  
 bind(DashBoardServicesStatusService.**class**).to(DashBoardServicesStatusService.**class**);  
 bind(TopHashTagResultService.**class**).to(TopHashTagResultService.**class**);  
 bind(TopUserResultService.**class**).to(TopUserResultService.**class**);  
 bind(HashtagResultsService.**class**).to(HashtagResultsService.**class**);  
 bind(UserResultsService.**class**).to(UserResultsService.**class**);  
 bind(SearchTermResultService.**class**).to(SearchTermResultService.**class**);  
 bind(SuggestedSearchResultsService.**class**).to(SuggestedSearchResultsService.**class**);  
  
 bind(DbConnectionResolver.**class**).to(DbConnectionResolver.**class**).in(Singleton.**class**);  
 }  
}

## client

**package** twitter.classification.api.client;  
  
**import** java.util.Optional;  
  
**import** javax.inject.Inject;  
**import** javax.ws.rs.client.Client;  
**import** javax.ws.rs.client.ClientBuilder;  
**import** javax.ws.rs.client.WebTarget;  
**import** javax.ws.rs.core.Response;  
  
**import** org.glassfish.jersey.client.ClientConfig;  
  
**import** com.fasterxml.jackson.jaxrs.json.JacksonJsonProvider;  
**import** twitter.classification.common.exceptions.ProcessingClientException;  
**import** twitter.classification.common.models.ClassifierStatusResponse;  
**import** twitter.classification.common.system.ConfigurationVariable;  
**import** twitter.classification.common.system.helper.ConfigurationVariableParam;  
**import** twitter.classification.common.tweetdetails.processing.ProcessResponse;  
  
**public class** ClassifierStatusClient {  
  
 **private** Client **client**;  
 **private** String **uri**;  
  
 @Inject  
 **public** ClassifierStatusClient(  
 @ConfigurationVariableParam(variable = ConfigurationVariable.***CLASSIFIER\_STATUS\_URI***) String uri  
 ) {  
  
 **this**.**client** = ClientBuilder.*newClient*(**new** ClientConfig(JacksonJsonProvider.**class**));  
 **this**.**uri** = uri;  
 }  
  
 */\*\*  
 \* Checks to see if the classifier is running in order for the status  
 \* to be displayed to the user  
 \*  
 \** ***@return*** *classifiers status  
 \** ***@throws*** *ProcessingClientException  
 \*/* **public** ClassifierStatusResponse isRunning() **throws** ProcessingClientException {  
  
 Response response;  
  
 **try** {  
  
 WebTarget target = **client**.target(**uri**);  
  
 response = **client**.target(target.getUri())  
 .request()  
 .get();  
  
 } **catch** (Exception exception) {  
  
 **return new** ClassifierStatusResponse().setRunning(**false**);  
 }  
  
 Optional<ClassifierStatusResponse> classifierStatusResponseOptional = ProcessResponse.*processResponse*(response, ClassifierStatusResponse.**class**);  
  
 **return** classifierStatusResponseOptional.orElseGet(() -> **new** ClassifierStatusResponse().setRunning(**false**));  
 }  
}

**package** twitter.classification.api.client;  
  
**import** java.util.Optional;  
  
**import** javax.inject.Inject;  
**import** javax.ws.rs.client.Client;  
**import** javax.ws.rs.client.ClientBuilder;  
**import** javax.ws.rs.client.WebTarget;  
**import** javax.ws.rs.core.Response;  
  
**import** org.glassfish.jersey.client.ClientConfig;  
  
**import** com.fasterxml.jackson.jaxrs.json.JacksonJsonProvider;  
**import** twitter.classification.common.exceptions.ProcessingClientException;  
**import** twitter.classification.common.models.PreProcessorStatusResponse;  
**import** twitter.classification.common.system.ConfigurationVariable;  
**import** twitter.classification.common.system.helper.ConfigurationVariableParam;  
**import** twitter.classification.common.tweetdetails.processing.ProcessResponse;  
  
**public class** PreProcessorStatusClient {  
  
 **private** Client **client**;  
 **private** String **uri**;  
  
 @Inject  
 **public** PreProcessorStatusClient(  
 @ConfigurationVariableParam(variable = ConfigurationVariable.***PRE\_PROCESSOR\_STATUS\_URI***) String uri  
 ) {  
  
 **this**.**client** = ClientBuilder.*newClient*(**new** ClientConfig(JacksonJsonProvider.**class**));  
 **this**.**uri** = uri;  
 }  
  
 */\*\*  
 \* Check to see if the preprocessor is running to display back to the user in status report  
 \*  
 \** ***@return*** *pre processors status  
 \** ***@throws*** *ProcessingClientException  
 \*/* **public** PreProcessorStatusResponse isRunning() **throws** ProcessingClientException {  
  
 Response response;  
  
 **try** {  
  
 WebTarget target = **client**.target(**uri**);  
  
 response = **client**.target(target.getUri())  
 .request()  
 .get();  
  
 } **catch** (Exception exception) {  
  
 **return new** PreProcessorStatusResponse().setRunning(**false**);  
 }  
  
 Optional<PreProcessorStatusResponse> preProcessorStatusResponseOptional = ProcessResponse.*processResponse*(response, PreProcessorStatusResponse.**class**);  
  
 **return** preProcessorStatusResponseOptional.orElseGet(() -> **new** PreProcessorStatusResponse().setRunning(**false**));  
 }  
}

**package** twitter.classification.api.client;  
  
**import** java.util.Optional;  
  
**import** javax.inject.Inject;  
**import** javax.ws.rs.client.Client;  
**import** javax.ws.rs.client.ClientBuilder;  
**import** javax.ws.rs.client.WebTarget;  
**import** javax.ws.rs.core.Response;  
  
**import** org.glassfish.jersey.client.ClientConfig;  
  
**import** com.fasterxml.jackson.jaxrs.json.JacksonJsonProvider;  
**import** twitter.classification.common.exceptions.ProcessingClientException;  
**import** twitter.classification.common.models.TwitterStreamResponse;  
**import** twitter.classification.common.system.ConfigurationVariable;  
**import** twitter.classification.common.system.helper.ConfigurationVariableParam;  
**import** twitter.classification.common.tweetdetails.processing.ProcessResponse;  
  
**public class** TwitterStreamClient {  
  
 **private** Client **client**;  
 **private** String **uri**;  
  
 @Inject  
 **public** TwitterStreamClient(  
 @ConfigurationVariableParam(variable = ConfigurationVariable.***TWITTER\_STREAM\_STATUS\_URI***) String uri  
 ) {  
  
 **this**.**client** = ClientBuilder.*newClient*(**new** ClientConfig(JacksonJsonProvider.**class**));  
 **this**.**uri** = uri;  
 }  
  
 */\*\*  
 \* Returns the status of the twitter stream  
 \*  
 \** ***@return*** *status of twitter stream  
 \** ***@throws*** *ProcessingClientException  
 \*/* **public** TwitterStreamResponse isRunning() **throws** ProcessingClientException {  
  
 Response response;  
  
 **try** {  
  
 WebTarget target = **client**.target(**uri**);  
  
 response = **client**.target(target.getUri())  
 .request()  
 .get();  
  
 } **catch** (Exception exception) {  
  
 **return new** TwitterStreamResponse().setRunning(**false**);  
 }  
  
 Optional<TwitterStreamResponse> twitterStreamOptional = ProcessResponse.*processResponse*(response, TwitterStreamResponse.**class**);  
  
 **return** twitterStreamOptional.orElseGet(() -> **new** TwitterStreamResponse().setRunning(**false**).setFilterList(**null**));  
 }  
}

## persist/jdbc

**package** twitter.classification.api.persist.jdbc;  
  
**import** java.util.List;  
  
**import** javax.inject.Inject;  
  
**import** twitter.classification.api.persist.jdbc.models.PaginatedTweetsModel;  
**import** twitter.classification.api.persist.jdbc.queries.SelectHashtagTweetsDbQuery;  
**import** twitter.classification.api.persist.jdbc.queries.SelectUserTweetsDbQuery;  
**import** twitter.classification.common.persist.ConnectionManager;  
**import** twitter.classification.common.persist.jdbc.utils.DbQueryRunner;  
  
**public class** PaginatedHashtagTweetsDao {  
  
 **private** ConnectionManager **connectionManager**;  
  
 @Inject  
 **public** PaginatedHashtagTweetsDao(ConnectionManager connectionManager) {  
  
 **this**.**connectionManager** = connectionManager;  
 }  
  
 */\*\*  
 \* Returns the paginated tweet results for a hashtag term  
 \*  
 \** ***@param hashtag*** *\** ***@param offset*** *\** ***@param limit*** *\** ***@return*** *paginated results  
 \*/* **public** List<PaginatedTweetsModel> get(String hashtag, **int** offset, **int** limit) {  
  
 DbQueryRunner dbQueryRunner = **new** DbQueryRunner(**connectionManager**.getConnection());  
  
 **try** {  
  
 **return** dbQueryRunner.executeQuery(**new** SelectHashtagTweetsDbQuery().buildQuery(), PaginatedTweetsModel.**class**, hashtag, offset, limit);  
 } **catch** (Exception e) {  
  
 e.printStackTrace();  
 }  
  
 **return null**;  
 }  
}

**package** twitter.classification.api.persist.jdbc;  
  
**import** java.util.List;  
  
**import** javax.inject.Inject;  
  
**import** twitter.classification.api.persist.jdbc.models.PaginatedTweetsModel;  
**import** twitter.classification.api.persist.jdbc.queries.SelectSearchTermTweetsDbQuery;  
**import** twitter.classification.common.persist.ConnectionManager;  
**import** twitter.classification.common.persist.jdbc.utils.DbQueryRunner;  
  
**public class** PaginatedSearchTermTweetsDao {  
  
 **private** ConnectionManager **connectionManager**;  
  
 @Inject  
 **public** PaginatedSearchTermTweetsDao(ConnectionManager connectionManager) {  
  
 **this**.**connectionManager** = connectionManager;  
 }  
  
 */\*\*  
 \* Returns the paginated tweet results for a search term  
 \*  
 \** ***@param searchTerm*** *\** ***@param offset*** *\** ***@param limit*** *\** ***@return*** *paginated results  
 \*/* **public** List<PaginatedTweetsModel> get(String searchTerm, **int** offset, **int** limit) {  
  
 DbQueryRunner dbQueryRunner = **new** DbQueryRunner(**connectionManager**.getConnection());  
  
 **try** {  
  
 **return** dbQueryRunner.executeQuery(**new** SelectSearchTermTweetsDbQuery().buildQuery(), PaginatedTweetsModel.**class**, searchTerm, searchTerm, offset, limit);  
 } **catch** (Exception e) {  
  
 e.printStackTrace();  
 }  
  
 **return null**;  
 }  
}

**package** twitter.classification.api.persist.jdbc;  
  
**import** java.util.List;  
  
**import** javax.inject.Inject;  
  
**import** twitter.classification.api.persist.jdbc.models.PaginatedTweetsModel;  
**import** twitter.classification.api.persist.jdbc.queries.SelectUserTweetsDbQuery;  
**import** twitter.classification.common.persist.ConnectionManager;  
**import** twitter.classification.common.persist.jdbc.utils.DbQueryRunner;  
  
**public class** PaginatedUserTweetsDao {  
  
 **private** ConnectionManager **connectionManager**;  
  
 @Inject  
 **public** PaginatedUserTweetsDao(ConnectionManager connectionManager) {  
  
 **this**.**connectionManager** = connectionManager;  
 }  
  
 */\*\*  
 \* Returns the paginated tweet results for a username term  
 \*  
 \** ***@param username*** *\** ***@param offset*** *\** ***@param limit*** *\** ***@return*** *paginated results  
 \*/* **public** List<PaginatedTweetsModel> get(String username, **int** offset, **int** limit) {  
  
 DbQueryRunner dbQueryRunner = **new** DbQueryRunner(**connectionManager**.getConnection());  
  
 **try** {  
  
 **return** dbQueryRunner.executeQuery(**new** SelectUserTweetsDbQuery().buildQuery(), PaginatedTweetsModel.**class**, username, offset, limit);  
 } **catch** (Exception e) {  
  
 e.printStackTrace();  
 }  
  
 **return null**;  
 }  
}

**package** twitter.classification.api.persist.jdbc;  
  
**import** java.util.List;  
  
**import** javax.inject.Inject;  
  
**import** twitter.classification.api.persist.jdbc.models.DashBoardOverviewModel;  
**import** twitter.classification.api.persist.jdbc.queries.SelectDashBoardOverviewValuesDbQuery;  
**import** twitter.classification.common.persist.ConnectionManager;  
**import** twitter.classification.common.persist.jdbc.utils.DbQueryRunner;  
  
**public class** SelectDashBoardOverviewValuesDao {  
  
 **private** ConnectionManager **connectionManager**;  
  
 @Inject  
 **public** SelectDashBoardOverviewValuesDao(ConnectionManager connectionManager) {  
  
 **this**.**connectionManager** = connectionManager;  
 }  
  
 */\*\*  
 \* Returns the overview results for the dashboard  
 \*  
 \** ***@return*** *overview results  
 \*/* **public** List<DashBoardOverviewModel> select() {  
  
 DbQueryRunner dbQueryRunner = **new** DbQueryRunner(**connectionManager**.getConnection());  
  
 **try** {  
  
 **return** dbQueryRunner.executeQuery(**new** SelectDashBoardOverviewValuesDbQuery().buildQuery(), DashBoardOverviewModel.**class**);  
 } **catch** (Exception e) {  
  
 e.printStackTrace();  
 }  
  
 **return null**;  
 }  
}

**package** twitter.classification.api.persist.jdbc;  
  
**import** java.sql.SQLException;  
**import** java.util.List;  
  
**import** javax.inject.Inject;  
  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** twitter.classification.api.persist.jdbc.models.ClassificationCountModel;  
**import** twitter.classification.api.persist.jdbc.queries.SelectSearchTermClassificationCountDbQuery;  
**import** twitter.classification.common.persist.ConnectionManager;  
**import** twitter.classification.common.persist.jdbc.utils.DbQueryRunner;  
  
**public class** SelectSearchTermClassificationCountDao {  
  
 **private static final** Logger ***logger*** = LoggerFactory.*getLogger*(SelectSearchTermClassificationCountDao.**class**);  
  
 **private** ConnectionManager **connectionManager**;  
  
 @Inject  
 **public** SelectSearchTermClassificationCountDao(ConnectionManager connectionManager) {  
  
 **this**.**connectionManager** = connectionManager;  
 }  
  
 */\*\*  
 \* Classification count results for a search term  
 \*  
 \** ***@param searchTerm*** *\** ***@return*** *classification count results  
 \*/* **public** List<ClassificationCountModel> select(String searchTerm) {  
  
 DbQueryRunner dbQueryRunner = **new** DbQueryRunner(**connectionManager**.getConnection());  
  
 **try** {  
 **return** dbQueryRunner.executeQuery(**new** SelectSearchTermClassificationCountDbQuery().buildQuery(), ClassificationCountModel.**class**, searchTerm, searchTerm);  
 } **catch** (SQLException e) {  
 e.printStackTrace();  
 }  
 **return null**;  
 }  
}

**package** twitter.classification.api.persist.jdbc;  
  
**import** java.sql.SQLException;  
**import** java.util.List;  
  
**import** javax.inject.Inject;  
  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** twitter.classification.api.persist.jdbc.models.TopHashtagsClassificationModel;  
**import** twitter.classification.api.persist.jdbc.queries.SelectTopHashtagsClassificationCountDbQuery;  
**import** twitter.classification.common.persist.ConnectionManager;  
**import** twitter.classification.common.persist.jdbc.utils.DbQueryRunner;  
  
**public class** SelectTopHashtagsClassificationCountDao {  
  
 **private static final** Logger ***logger*** = LoggerFactory.*getLogger*(SelectTopHashtagsClassificationCountDao.**class**);  
  
 **private** ConnectionManager **connectionManager**;  
  
 @Inject  
 **public** SelectTopHashtagsClassificationCountDao(ConnectionManager connectionManager) {  
  
 **this**.**connectionManager** = connectionManager;  
 }  
  
 */\*\*  
 \* Classification count results for top hashtags  
 \*  
 \** ***@return*** *classification count results  
 \*/* **public** List<TopHashtagsClassificationModel> select() {  
  
 DbQueryRunner dbQueryRunner = **new** DbQueryRunner(**connectionManager**.getConnection());  
  
 **try** {  
 **return** dbQueryRunner.executeQuery(**new** SelectTopHashtagsClassificationCountDbQuery().buildQuery(), TopHashtagsClassificationModel.**class**);  
 } **catch** (SQLException e) {  
 e.printStackTrace();  
 }  
 **return null**;  
 }  
}

**package** twitter.classification.api.persist.jdbc;  
  
**import** java.sql.SQLException;  
**import** java.util.List;  
  
**import** javax.inject.Inject;  
  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** twitter.classification.api.persist.jdbc.models.TopUsersClassificationModel;  
**import** twitter.classification.api.persist.jdbc.queries.SelectTopUsersClassificationCountDbQuery;  
**import** twitter.classification.common.persist.ConnectionManager;  
**import** twitter.classification.common.persist.jdbc.utils.DbQueryRunner;  
  
**public class** SelectTopUsersClassificationCountDao {  
  
 **private static final** Logger ***logger*** = LoggerFactory.*getLogger*(SelectTopUsersClassificationCountDao.**class**);  
  
 **private** ConnectionManager **connectionManager**;  
  
 @Inject  
 **public** SelectTopUsersClassificationCountDao(ConnectionManager connectionManager) {  
  
 **this**.**connectionManager** = connectionManager;  
 }  
  
 */\*\*  
 \* Classification count results for top users  
 \*  
 \** ***@return*** *classification count results  
 \*/* **public** List<TopUsersClassificationModel> select() {  
  
 DbQueryRunner dbQueryRunner = **new** DbQueryRunner(**connectionManager**.getConnection());  
  
 **try** {  
 **return** dbQueryRunner.executeQuery(**new** SelectTopUsersClassificationCountDbQuery().buildQuery(), TopUsersClassificationModel.**class**);  
 } **catch** (SQLException e) {  
 e.printStackTrace();  
 }  
 **return null**;  
 }  
}

**package** twitter.classification.api.persist.jdbc;  
  
**import** java.util.List;  
  
**import** javax.inject.Inject;  
  
**import** twitter.classification.api.persist.jdbc.models.SuggestedSearchResultsModel;  
**import** twitter.classification.api.persist.jdbc.queries.SuggestedSearchResultsDbQuery;  
**import** twitter.classification.common.persist.ConnectionManager;  
**import** twitter.classification.common.persist.jdbc.utils.DbQueryRunner;  
  
**public class** SuggestedSearchResultsDao {  
  
 **private** ConnectionManager **connectionManager**;  
  
 @Inject  
 **public** SuggestedSearchResultsDao(ConnectionManager connectionManager) {  
  
 **this**.**connectionManager** = connectionManager;  
 }  
  
 */\*\*  
 \* Return a list of suggested search terms which the user can perform  
 \*  
 \** ***@return*** *suggested search terms  
 \*/* **public** List<SuggestedSearchResultsModel> get() {  
  
 DbQueryRunner dbQueryRunner = **new** DbQueryRunner(**connectionManager**.getConnection());  
  
 **try** {  
  
 **return** dbQueryRunner.executeQuery(**new** SuggestedSearchResultsDbQuery().buildQuery(), SuggestedSearchResultsModel.**class**);  
 } **catch** (Exception e) {  
  
 e.printStackTrace();  
 }  
  
 **return null**;  
 }  
}

**package** twitter.classification.api.persist.jdbc;  
  
**import** javax.inject.Inject;  
  
**import** twitter.classification.common.persist.ConnectionManager;  
  
**public class** TestDatabaseConnectionDao {  
  
 **private** ConnectionManager **connectionManager**;  
  
 @Inject  
 **public** TestDatabaseConnectionDao(ConnectionManager connectionManager) {  
  
 **this**.**connectionManager** = connectionManager;  
 }  
  
 */\*\*  
 \* Test the database for the status report on the homepage/dashboard  
 \*  
 \** ***@return*** *results if the database is working  
 \*/* **public boolean** test() {  
  
 **try** {  
 **return connectionManager**.getConnection().isValid(10);  
 } **catch** (Exception e) {  
  
 **return false**;  
 }  
 }  
}

**package** twitter.classification.api.persist.jdbc;  
  
**import** java.sql.SQLException;  
**import** java.util.List;  
  
**import** javax.inject.Inject;  
  
**import** com.fasterxml.jackson.core.JsonProcessingException;  
**import** com.fasterxml.jackson.databind.ObjectMapper;  
**import** twitter.classification.api.persist.jdbc.models.ProcessedHashtagTweetsForWordCloudModel;  
**import** twitter.classification.api.persist.jdbc.models.TimeLineForTweetsModel;  
**import** twitter.classification.api.persist.jdbc.queries.TimeLineForHashtagsDbQuery;  
**import** twitter.classification.common.persist.ConnectionFactory;  
**import** twitter.classification.common.persist.ConnectionManager;  
**import** twitter.classification.common.persist.jdbc.MySqlConnectionFactory;  
**import** twitter.classification.common.persist.jdbc.utils.DbQueryRunner;  
  
**public class** TimeLineForHashtagsDao {  
  
 **private** ConnectionManager **connectionManager**;  
  
 @Inject  
 **public** TimeLineForHashtagsDao(ConnectionManager connectionManager) {  
  
 **this**.**connectionManager** = connectionManager;  
 }  
  
 */\*\*  
 \* Return the timeline for particular hashtag term  
 \*  
 \** ***@param hashtag*** *\** ***@return*** *timeline results  
 \*/* **public** List<TimeLineForTweetsModel> get(String hashtag) {  
  
 DbQueryRunner dbQueryRunner = **new** DbQueryRunner(**connectionManager**.getConnection());  
  
 **try** {  
  
 **return** dbQueryRunner.executeQuery(**new** TimeLineForHashtagsDbQuery().buildQuery(), TimeLineForTweetsModel.**class**, hashtag, hashtag, hashtag, hashtag, hashtag, hashtag, hashtag, hashtag, hashtag, hashtag);  
 } **catch** (Exception e) {  
  
 e.printStackTrace();  
 }  
  
 **return null**;  
 }  
}

**package** twitter.classification.api.persist.jdbc;  
  
**import** java.sql.SQLException;  
**import** java.util.List;  
  
**import** javax.inject.Inject;  
  
**import** com.fasterxml.jackson.core.JsonProcessingException;  
**import** com.fasterxml.jackson.databind.ObjectMapper;  
**import** twitter.classification.api.persist.jdbc.models.TimeLineForTweetsModel;  
**import** twitter.classification.api.persist.jdbc.queries.TimeLineForSearchTermDbQuery;  
**import** twitter.classification.common.persist.ConnectionFactory;  
**import** twitter.classification.common.persist.ConnectionManager;  
**import** twitter.classification.common.persist.jdbc.MySqlConnectionFactory;  
**import** twitter.classification.common.persist.jdbc.utils.DbQueryRunner;  
  
**public class** TimeLineForSearchTermDao {  
  
 **private** ConnectionManager **connectionManager**;  
  
 @Inject  
 **public** TimeLineForSearchTermDao(ConnectionManager connectionManager) {  
  
 **this**.**connectionManager** = connectionManager;  
 }  
  
 */\*\*  
 \* Return the timeline for particular search term  
 \*  
 \** ***@param searchTerm*** *\** ***@return*** *timeline results  
 \*/* **public** List<TimeLineForTweetsModel> get(String searchTerm) {  
  
 DbQueryRunner dbQueryRunner = **new** DbQueryRunner(**connectionManager**.getConnection());  
  
 **try** {  
  
 **return** dbQueryRunner.executeQuery(**new** TimeLineForSearchTermDbQuery().buildQuery(), TimeLineForTweetsModel.**class**, searchTerm, searchTerm, searchTerm, searchTerm, searchTerm, searchTerm, searchTerm, searchTerm, searchTerm, searchTerm, searchTerm, searchTerm, searchTerm, searchTerm, searchTerm, searchTerm, searchTerm, searchTerm, searchTerm, searchTerm);  
 } **catch** (Exception e) {  
  
 e.printStackTrace();  
 }  
  
 **return null**;  
 }  
}

**package** twitter.classification.api.persist.jdbc;  
  
**import** java.util.List;  
  
**import** javax.inject.Inject;  
  
**import** twitter.classification.api.persist.jdbc.models.TimeLineForTweetsModel;  
**import** twitter.classification.api.persist.jdbc.queries.TimeLineForHashtagsDbQuery;  
**import** twitter.classification.api.persist.jdbc.queries.TimeLineForUsersDbQuery;  
**import** twitter.classification.common.persist.ConnectionManager;  
**import** twitter.classification.common.persist.jdbc.utils.DbQueryRunner;  
  
**public class** TimeLineForUsersDao {  
  
 **private** ConnectionManager **connectionManager**;  
  
 @Inject  
 **public** TimeLineForUsersDao(ConnectionManager connectionManager) {  
  
 **this**.**connectionManager** = connectionManager;  
 }  
  
 */\*\*  
 \* Return the timeline for particular username term  
 \*  
 \** ***@param username*** *\** ***@return*** *timeline results  
 \*/* **public** List<TimeLineForTweetsModel> get(String username) {  
  
 DbQueryRunner dbQueryRunner = **new** DbQueryRunner(**connectionManager**.getConnection());  
  
 **try** {  
  
 **return** dbQueryRunner.executeQuery(**new** TimeLineForUsersDbQuery().buildQuery(), TimeLineForTweetsModel.**class**, username, username, username, username, username, username, username, username, username, username);  
 } **catch** (Exception e) {  
  
 e.printStackTrace();  
 }  
  
 **return null**;  
 }  
}

**package** twitter.classification.api.persist.jdbc;  
  
**import** java.util.List;  
  
**import** javax.inject.Inject;  
  
**import** twitter.classification.api.persist.jdbc.models.ProcessedHashtagTweetsForWordCloudModel;  
**import** twitter.classification.api.persist.jdbc.queries.TweetsForHashtagWordCloudDbQuery;  
**import** twitter.classification.common.persist.ConnectionManager;  
**import** twitter.classification.common.persist.jdbc.utils.DbQueryRunner;  
  
**public class** TweetsForHashtagsDao {  
  
 **private** ConnectionManager **connectionManager**;  
  
 @Inject  
 **public** TweetsForHashtagsDao(ConnectionManager connectionManager) {  
  
 **this**.**connectionManager** = connectionManager;  
 }  
  
 */\*\*  
 \* Return the word cloud results for a hashtag  
 \*  
 \** ***@param hashtag*** *\** ***@return*** *wordcloud results  
 \*/* **public** List<ProcessedHashtagTweetsForWordCloudModel> get(String hashtag) {  
  
 DbQueryRunner dbQueryRunner = **new** DbQueryRunner(**connectionManager**.getConnection());  
  
 **try** {  
  
 **return** dbQueryRunner.executeQuery(**new** TweetsForHashtagWordCloudDbQuery().buildQuery(), ProcessedHashtagTweetsForWordCloudModel.**class**, hashtag);  
 } **catch** (Exception e) {  
  
 e.printStackTrace();  
 }  
  
 **return null**;  
 }  
}

**package** twitter.classification.api.persist.jdbc;  
  
**import** java.util.List;  
  
**import** javax.inject.Inject;  
  
**import** twitter.classification.api.persist.jdbc.models.ProcessedTweetsForWordCloudModel;  
**import** twitter.classification.api.persist.jdbc.queries.TweetsForSearchTermWordCloudDbQuery;  
**import** twitter.classification.common.persist.ConnectionManager;  
**import** twitter.classification.common.persist.jdbc.utils.DbQueryRunner;  
  
**public class** TweetsForSearchTermDao {  
  
 **private** ConnectionManager **connectionManager**;  
  
 @Inject  
 **public** TweetsForSearchTermDao(ConnectionManager connectionManager) {  
  
 **this**.**connectionManager** = connectionManager;  
 }  
  
 */\*\*  
 \* Return the word cloud results for a search term  
 \*  
 \** ***@param searchTerm*** *\** ***@return*** *wordcloud results  
 \*/* **public** List<ProcessedTweetsForWordCloudModel> get(String searchTerm) {  
  
 DbQueryRunner dbQueryRunner = **new** DbQueryRunner(**connectionManager**.getConnection());  
  
 **try** {  
  
 **return** dbQueryRunner.executeQuery(**new** TweetsForSearchTermWordCloudDbQuery().buildQuery(), ProcessedTweetsForWordCloudModel.**class**, searchTerm, searchTerm);  
 } **catch** (Exception e) {  
  
 e.printStackTrace();  
 }  
  
 **return null**;  
 }  
}

**package** twitter.classification.api.persist.jdbc;  
  
**import** java.util.List;  
  
**import** javax.inject.Inject;  
  
**import** twitter.classification.api.persist.jdbc.models.ProcessedHashtagTweetsForWordCloudModel;  
**import** twitter.classification.api.persist.jdbc.models.ProcessedUserTweetsForWordCloudModel;  
**import** twitter.classification.api.persist.jdbc.queries.TweetsForHashtagWordCloudDbQuery;  
**import** twitter.classification.api.persist.jdbc.queries.TweetsForUserWordCloudDbQuery;  
**import** twitter.classification.common.persist.ConnectionManager;  
**import** twitter.classification.common.persist.jdbc.utils.DbQueryRunner;  
  
**public class** TweetsForUsersDao {  
  
 **private** ConnectionManager **connectionManager**;  
  
 @Inject  
 **public** TweetsForUsersDao(ConnectionManager connectionManager) {  
  
 **this**.**connectionManager** = connectionManager;  
 }  
  
 */\*\*  
 \* Return the word cloud results for a username  
 \*  
 \** ***@param username*** *\** ***@return*** *wordcloud results  
 \*/* **public** List<ProcessedUserTweetsForWordCloudModel> get(String username) {  
  
 DbQueryRunner dbQueryRunner = **new** DbQueryRunner(**connectionManager**.getConnection());  
  
 **try** {  
  
 **return** dbQueryRunner.executeQuery(**new** TweetsForUserWordCloudDbQuery().buildQuery(), ProcessedUserTweetsForWordCloudModel.**class**, username);  
 } **catch** (Exception e) {  
  
 e.printStackTrace();  
 }  
  
 **return null**;  
 }  
}

### models

**package** twitter.classification.api.persist.jdbc.models;  
  
**import** java.math.BigDecimal;  
  
**import** twitter.classification.common.persist.Column;  
**import** twitter.classification.common.persist.Entity;  
  
@Entity  
**public class** ClassificationCountModel {  
  
 @Column(name = **"count\_of\_rumours"**)  
 **private** BigDecimal **countOfRumours**;  
  
 @Column(name = **"count\_of\_non\_rumours"**)  
 **private** BigDecimal **countOfNonRumours**;  
  
 @Column(name = **"total\_classification\_count"**)  
 **private** BigDecimal **totalClassificationCount**;  
  
 **public** BigDecimal getCountOfRumours() {  
  
 **return countOfRumours**;  
 }  
  
 **public** BigDecimal getCountOfNonRumours() {  
  
 **return countOfNonRumours**;  
 }  
  
 **public** BigDecimal getTotalClassificationCount() {  
  
 **return totalClassificationCount**;  
 }  
}

**package** twitter.classification.api.persist.jdbc.models;  
  
**import** twitter.classification.common.persist.Column;  
**import** twitter.classification.common.persist.Entity;  
  
@Entity  
**public class** DashBoardOverviewModel {  
  
 @Column(name = **"count\_of\_rumours"**)  
 **private** Long **countOfRumours**;  
 @Column(name = **"count\_of\_non\_rumours"**)  
 **private** Long **countOfNonRumours**;  
 @Column(name = **"total\_count\_of\_classifications"**)  
 **private** Long **totalCountOfClassifications**;  
 @Column(name = **"count\_of\_users"**)  
 **private** Long **countOfUsers**;  
 @Column(name = **"count\_of\_hashtags"**)  
 **private** Long **countOfHashtags**;  
 @Column(name = **"count\_of\_tweets"**)  
 **private** Long **countOfTweets**;  
  
 **public** Long getCountOfRumours() {  
  
 **return countOfRumours**;  
 }  
  
 **public** Long getCountOfNonRumours() {  
  
 **return countOfNonRumours**;  
 }  
  
 **public** Long getTotalCountOfClassifications() {  
  
 **return totalCountOfClassifications**;  
 }  
  
 **public** Long getCountOfUsers() {  
  
 **return countOfUsers**;  
 }  
  
 **public** Long getCountOfHashtags() {  
  
 **return countOfHashtags**;  
 }  
  
 **public** Long getCountOfTweets() {  
  
 **return countOfTweets**;  
 }  
}

**package** twitter.classification.api.persist.jdbc.models;  
  
**import** twitter.classification.common.persist.Column;  
  
**public class** PaginatedTweetsModel {  
  
 @Column(name = **"classification\_value"**)  
 **private** String **classificationValue**;  
 @Column(name = **"processed\_tweet\_text"**)  
 **private** String **tweetText**;  
 @Column(name = **"id"**)  
 **private** Long **id**;  
  
 **public** PaginatedTweetsModel() {  
 }  
  
 **public** String getClassificationValue() {  
  
 **return classificationValue**;  
 }  
  
 **public void** setClassificationValue(String classificationValue) {  
  
 **this**.**classificationValue** = classificationValue;  
 }  
  
 **public** String getTweetText() {  
  
 **return tweetText**;  
 }  
  
 **public void** setTweetText(String tweetText) {  
  
 **this**.**tweetText** = tweetText;  
 }  
  
 **public** Long getId() {  
  
 **return id**;  
 }  
  
 **public void** setId(Long id) {  
  
 **this**.**id** = id;  
 }  
}

**package** twitter.classification.api.persist.jdbc.models;  
  
**import** java.io.Serializable;  
  
**import** twitter.classification.common.persist.Column;  
**import** twitter.classification.common.persist.Entity;  
  
@Entity  
**public class** ProcessedHashtagTweetsForWordCloudModel **extends** ProcessedTweetsForWordCloudModel {  
  
 @Column(name = **"hashtag\_value"**)  
 **private** String **hashtagValue**;  
  
 **public** ProcessedHashtagTweetsForWordCloudModel() {  
 }  
  
 **public** String getHashtagValue() {  
  
 **return hashtagValue**;  
 }  
  
 **public void** setHashtagValue(String hashtagValue) {  
  
 **this**.**hashtagValue** = hashtagValue;  
 }  
}

**package** twitter.classification.api.persist.jdbc.models;  
  
**import** twitter.classification.common.persist.Column;  
**import** twitter.classification.common.persist.Entity;  
  
@Entity  
**public class** ProcessedTweetsForWordCloudModel {  
  
 @Column(name = **"processed\_tweet\_text"**)  
 **private** String **originalTextList**;  
  
 **public** ProcessedTweetsForWordCloudModel() {  
 }  
  
 **public** String getOriginalTextList() {  
  
 **return originalTextList**;  
 }  
  
 **public void** setOriginalTextList(String originalTextList) {  
  
 **this**.**originalTextList** = originalTextList;  
 }  
}

**package** twitter.classification.api.persist.jdbc.models;  
  
**import** twitter.classification.common.persist.Column;  
**import** twitter.classification.common.persist.Entity;  
  
@Entity  
**public class** ProcessedUserTweetsForWordCloudModel **extends** ProcessedTweetsForWordCloudModel {  
  
 @Column(name = **"username"**)  
 **private** String **username**;  
  
 **public** ProcessedUserTweetsForWordCloudModel() {  
 }  
  
 **public** String getUsername() {  
  
 **return username**;  
 }  
  
 **public void** setUsername(String username) {  
  
 **this**.**username** = username;  
 }  
}

**package** twitter.classification.api.persist.jdbc.models;  
  
**import** java.math.BigDecimal;  
  
**import** twitter.classification.common.persist.Column;  
  
**public class** SuggestedSearchResultsModel {  
  
 @Column(name = **"value"**)  
 **private** String **value**;  
 @Column(name = **"total\_classification\_count"**)  
 **private** BigDecimal **totalClassificationCount**;  
  
 **public** String getValue() {  
  
 **return value**;  
 }  
  
 **public** SuggestedSearchResultsModel setValue(String value) {  
  
 **this**.**value** = value;  
  
 **return this**;  
 }  
  
 **public** BigDecimal getTotalClassificationCount() {  
  
 **return totalClassificationCount**;  
 }  
  
 **public** SuggestedSearchResultsModel setTotalClassificationCount(BigDecimal totalClassificationCount) {  
  
 **this**.**totalClassificationCount** = totalClassificationCount;  
  
 **return this**;  
 }  
}

**package** twitter.classification.api.persist.jdbc.models;  
  
**import** twitter.classification.common.persist.Column;  
**import** twitter.classification.common.persist.Entity;  
  
@Entity  
**public class** TimeLineForTweetsModel {  
  
 @Column(name = **"count\_of\_rumours\_in\_last\_hour"**)  
 **private** Long **countOfRumoursLastHour**;  
 @Column(name = **"count\_of\_non\_rumours\_in\_last\_hour"**)  
 **private** Long **countOfNonRumoursLastHour**;  
 @Column(name = **"count\_of\_rumours\_over\_an\_hour"**)  
 **private** Long **countOfRumoursOverAnHour**;  
 @Column(name = **"count\_of\_non\_rumours\_over\_an\_hour"**)  
 **private** Long **countOfNonRumoursOverAnHour**;  
 @Column(name = **"count\_of\_rumours\_over\_two\_hours"**)  
 **private** Long **countOfRumoursOverTwoHours**;  
 @Column(name = **"count\_of\_non\_rumours\_over\_two\_hours"**)  
 **private** Long **countOfNonRumoursOverTwoHours**;  
 @Column(name = **"count\_of\_rumours\_over\_three\_hours"**)  
 **private** Long **countOfRumoursOverThreeHours**;  
 @Column(name = **"count\_of\_non\_rumours\_over\_three\_hours"**)  
 **private** Long **countOfNonRumoursOverThreeHours**;  
 @Column(name = **"count\_of\_rumours\_over\_four\_hours"**)  
 **private** Long **countOfRumoursOverFourHours**;  
 @Column(name = **"count\_of\_non\_rumours\_over\_four\_hours"**)  
 **private** Long **countOfNonRumoursOverFourHours**;  
  
 **public** Long getCountOfRumoursLastHour() {  
  
 **return countOfRumoursLastHour**;  
 }  
  
 **public** Long getCountOfNonRumoursLastHour() {  
  
 **return countOfNonRumoursLastHour**;  
 }  
  
 **public** Long getCountOfRumoursOverAnHour() {  
  
 **return countOfRumoursOverAnHour**;  
 }  
  
 **public** Long getCountOfNonRumoursOverAnHour() {  
  
 **return countOfNonRumoursOverAnHour**;  
 }  
  
 **public** Long getCountOfRumoursOverTwoHours() {  
  
 **return countOfRumoursOverTwoHours**;  
 }  
  
 **public** Long getCountOfNonRumoursOverTwoHours() {  
  
 **return countOfNonRumoursOverTwoHours**;  
 }  
  
 **public** Long getCountOfRumoursOverThreeHours() {  
  
 **return countOfRumoursOverThreeHours**;  
 }  
  
 **public** Long getCountOfNonRumoursOverThreeHours() {  
  
 **return countOfNonRumoursOverThreeHours**;  
 }  
  
 **public** Long getCountOfRumoursOverFourHours() {  
  
 **return countOfRumoursOverFourHours**;  
 }  
  
 **public** Long getCountOfNonRumoursOverFourHours() {  
  
 **return countOfNonRumoursOverFourHours**;  
 }  
}

**package** twitter.classification.api.persist.jdbc.models;  
  
**import** java.math.BigDecimal;  
  
**import** twitter.classification.common.persist.Column;  
**import** twitter.classification.common.persist.Entity;  
  
@Entity  
**public class** TopHashtagsClassificationModel **extends** ClassificationCountModel {  
  
 @Column(name = **"hashtag\_value"**)  
 **private** String **hashtagValue**;  
  
 **public** String getHashtagValue() {  
  
 **return hashtagValue**;  
 }  
}

**package** twitter.classification.api.persist.jdbc.models;  
  
**import** java.math.BigDecimal;  
  
**import** twitter.classification.common.persist.Column;  
**import** twitter.classification.common.persist.Entity;  
  
@Entity  
**public class** TopUsersClassificationModel **extends** ClassificationCountModel {  
  
 @Column(name = **"username"**)  
 **private** String **username**;  
  
 **public** String getUsername() {  
  
 **return username**;  
 }  
}

### queries

**package** twitter.classification.api.persist.jdbc.queries;  
  
**import** twitter.classification.common.persist.jdbc.queries.DbQuery;  
  
**public class** SelectDashBoardOverviewValuesDbQuery **implements** DbQuery {  
  
 */\*\*  
 \* DB query to return the count of rumours, non-rumours, total classifications etc.  
 \* for the dashboard results page  
 \*  
 \** ***@return*** *sql for dashboard overview results  
 \*/* @Override  
 **public** String buildQuery() {  
  
 **return "SELECT DISTINCT "** +  
 **" (SELECT count(\*) FROM tweets JOIN classification\_types ON classification\_id = classification\_types.id WHERE classification\_value = 'rumour') count\_of\_rumours,"** +  
 **" (SELECT count(\*) FROM tweets JOIN classification\_types ON classification\_id = classification\_types.id WHERE classification\_value = 'non-rumour') count\_of\_non\_rumours,"** +  
 **" (SELECT count(\*) FROM tweets JOIN classification\_types ON classification\_id = classification\_types.id WHERE classification\_value = 'rumour' OR classification\_value = 'non-rumour') total\_count\_of\_classifications,"** +  
 **" (SELECT count(\*) FROM users) as count\_of\_users,"** +  
 **" (SELECT count(\*) FROM tweets) as count\_of\_tweets,"** +  
 **" (SELECT count(\*) FROM hashtags) as count\_of\_hashtags;"**;  
 }  
}

**package** twitter.classification.api.persist.jdbc.queries;  
  
**import** twitter.classification.common.persist.jdbc.queries.DbQuery;  
  
**public class** SelectHashtagTweetsDbQuery **implements** DbQuery {  
  
 */\*\*  
 \* Sql to return the classification value with the text for a particular hashtag  
 \*  
 \** ***@return*** *sql for a particular hashtag result  
 \*/* @Override  
 **public** String buildQuery() {  
  
 **return "SELECT classification\_types.classification\_value, tweets.processed\_tweet\_text, tweets.id "** +  
 **"FROM hashtags "** +  
 **" JOIN hashtag\_tweet\_classifications ON hashtags.id = hashtag\_tweet\_classifications.hashtag\_id "** +  
 **" JOIN tweets ON hashtag\_tweet\_classifications.tweet\_id = tweets.id "** +  
 **" JOIN classification\_types ON tweets.classification\_id = classification\_types.id "** +  
 **"WHERE hashtags.hashtag\_value = ? "** +  
 **"GROUP BY tweets.id "** +  
 **"LIMIT ?, ?;"**;  
 }  
}

**package** twitter.classification.api.persist.jdbc.queries;  
  
**import** twitter.classification.common.persist.jdbc.queries.DbQuery;  
  
**public class** SelectSearchTermClassificationCountDbQuery **implements** DbQuery {  
  
 */\*\*  
 \* Sql for the count results for a particular hashtag/user name search result  
 \*  
 \** ***@return*** *sql  
 \*/* @Override  
 **public** String buildQuery() {  
  
 **return "SELECT"** +  
 **" sum(CASE WHEN classification\_types.classification\_value = 'rumour' THEN 1 ELSE 0 END) count\_of\_rumours,"** +  
 **" sum(CASE WHEN classification\_types.classification\_value = 'non-rumour' THEN 1 ELSE 0 END) count\_of\_non\_rumours,"** +  
 **" sum(CASE WHEN classification\_types.classification\_value = 'non-rumour' OR classification\_types.classification\_value = 'rumour' THEN 1 ELSE 0 END) total\_classification\_count "** +  
 **"FROM users\_tweet\_classifications"** +  
 **" JOIN users"** +  
 **" ON users\_tweet\_classifications.user\_id = users.id"** +  
 **" JOIN tweets"** +  
 **" ON users\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN hashtag\_tweet\_classifications"** +  
 **" ON hashtag\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN hashtags"** +  
 **" ON hashtags.id = hashtag\_tweet\_classifications.hashtag\_id"** +  
 **" JOIN classification\_types"** +  
 **" ON tweets.classification\_id = classification\_types.id "** +  
 **"WHERE hashtags.hashtag\_value = ?"** +  
 **" OR users.username = ? "** +  
 **"ORDER BY total\_classification\_count DESC;"**;  
 }  
}

**package** twitter.classification.api.persist.jdbc.queries;  
  
**import** twitter.classification.common.persist.jdbc.queries.DbQuery;  
  
**public class** SelectSearchTermTweetsDbQuery **implements** DbQuery {  
  
 */\*\*  
 \* Sql for the tweets for a particular search term  
 \*  
 \** ***@return*** *sql for search term results  
 \*/* @Override  
 **public** String buildQuery() {  
  
 **return "SELECT classification\_types.classification\_value, tweets.processed\_tweet\_text, tweets.id "** +  
 **"FROM users "** +  
 **" JOIN users\_tweet\_classifications ON users.id = users\_tweet\_classifications.user\_id "** +  
 **" JOIN tweets ON users\_tweet\_classifications.tweet\_id = tweets.id "** +  
 **" JOIN hashtag\_tweet\_classifications ON hashtag\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN hashtags ON hashtags.id = hashtag\_tweet\_classifications.hashtag\_id"** +  
 **" JOIN classification\_types ON tweets.classification\_id = classification\_types.id "** +  
 **"WHERE users.username = ? OR hashtags.hashtag\_value = ? "** +  
 **"GROUP BY tweets.id "** +  
 **"LIMIT ?, ?;"**;  
 }  
}

**package** twitter.classification.api.persist.jdbc.queries;  
  
**import** twitter.classification.common.persist.jdbc.queries.DbQuery;  
  
**public class** SelectTopHashtagsClassificationCountDbQuery **implements** DbQuery {  
  
 */\*\*  
 \* The count of top hashtags results and the classification stats  
 \*  
 \** ***@return*** *sql for hashtag results  
 \*/* @Override  
 **public** String buildQuery() {  
  
 **return "SELECT hashtags.hashtag\_value,"** +  
 **" sum(CASE WHEN classification\_types.classification\_value = 'rumour' THEN 1 ELSE 0 END) count\_of\_rumours,"** +  
 **" sum(CASE WHEN classification\_types.classification\_value = 'non-rumour' THEN 1 ELSE 0 END) count\_of\_non\_rumours,"** +  
 **" sum(CASE WHEN classification\_types.classification\_value = 'non-rumour' OR classification\_types.classification\_value = 'rumour' THEN 1 ELSE 0 END) total\_classification\_count "** +  
 **"FROM hashtag\_tweet\_classifications"** +  
 **" JOIN hashtags"** +  
 **" ON hashtag\_tweet\_classifications.hashtag\_id = hashtags.id"** +  
 **" JOIN tweets"** +  
 **" ON hashtag\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN classification\_types"** +  
 **" ON tweets.classification\_id = classification\_types.id "** +  
 **"GROUP BY hashtags.hashtag\_value "** +  
 **"ORDER BY total\_classification\_count DESC "** +  
 **"LIMIT 10;"**;  
 }  
}

**package** twitter.classification.api.persist.jdbc.queries;  
  
**import** twitter.classification.common.persist.jdbc.queries.DbQuery;  
  
**public class** SelectTopUsersClassificationCountDbQuery **implements** DbQuery {  
  
 */\*\*  
 \* The count of top users results and the classification stats  
 \*  
 \** ***@return*** *sql for users results  
 \*/* @Override  
 **public** String buildQuery() {  
  
 **return "SELECT users.username,"** +  
 **" sum(CASE WHEN classification\_types.classification\_value = 'rumour' THEN 1 ELSE 0 END) count\_of\_rumours,"** +  
 **" sum(CASE WHEN classification\_types.classification\_value = 'non-rumour' THEN 1 ELSE 0 END) count\_of\_non\_rumours,"** +  
 **" sum(CASE WHEN classification\_types.classification\_value = 'non-rumour' OR classification\_types.classification\_value = 'rumour' THEN 1 ELSE 0 END) total\_classification\_count "** +  
 **"FROM users\_tweet\_classifications"** +  
 **" JOIN users"** +  
 **" ON users\_tweet\_classifications.user\_id = users.id"** +  
 **" JOIN tweets"** +  
 **" ON users\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN classification\_types"** +  
 **" ON tweets.classification\_id = classification\_types.id "** +  
 **"GROUP BY users.username "** +  
 **"ORDER BY total\_classification\_count DESC "** +  
 **"LIMIT 10;"**;  
 }  
}

**package** twitter.classification.api.persist.jdbc.queries;  
  
**import** twitter.classification.common.persist.jdbc.queries.DbQuery;  
  
**public class** SelectUserTweetsDbQuery **implements** DbQuery {  
  
 */\*\*  
 \* Sql for a particular users results including the classification values  
 \*  
 \** ***@return*** *sql table results for a user  
 \*/* @Override  
 **public** String buildQuery() {  
  
 **return "SELECT classification\_types.classification\_value, tweets.processed\_tweet\_text, tweets.id "** +  
 **"FROM users "** +  
 **" JOIN users\_tweet\_classifications ON users.id = users\_tweet\_classifications.user\_id "** +  
 **" JOIN tweets ON users\_tweet\_classifications.tweet\_id = tweets.id "** +  
 **" JOIN classification\_types ON tweets.classification\_id = classification\_types.id "** +  
 **"WHERE users.username = ? "** +  
 **"GROUP BY tweets.id "** +  
 **"LIMIT ?, ?;"**;  
 }  
}

**package** twitter.classification.api.persist.jdbc.queries;  
  
**import** twitter.classification.common.persist.jdbc.queries.DbQuery;  
  
**public class** SuggestedSearchResultsDbQuery **implements** DbQuery {  
  
 */\*\*  
 \* DB Query to fetch the hashtag/username value based on the total classifications for the term,  
 \* in order to suggest them to the user when their search returns 0 results  
 \*  
 \** ***@return*** *sql  
 \*/* @Override  
 **public** String buildQuery() {  
  
 **return "SELECT"** +  
 **" (CASE WHEN users.username IS NOT NULL THEN users.username ELSE hashtags.hashtag\_value END) as 'value',"** +  
 **" sum(CASE WHEN classification\_types.classification\_value = 'non-rumour' OR classification\_types.classification\_value = 'rumour' THEN 1 ELSE 0 END) total\_classification\_count"** +  
 **" FROM tweets"** +  
 **" JOIN hashtag\_tweet\_classifications ON tweets.id = hashtag\_tweet\_classifications.tweet\_id"** +  
 **" JOIN hashtags ON hashtag\_tweet\_classifications.hashtag\_id = hashtags.id"** +  
 **" JOIN users\_tweet\_classifications ON tweets.id = users\_tweet\_classifications.tweet\_id"** +  
 **" JOIN users ON users\_tweet\_classifications.user\_id = users.id"** +  
 **" JOIN classification\_types ON tweets.classification\_id = classification\_types.id"** +  
 **" GROUP BY value"** +  
 **" ORDER BY total\_classification\_count DESC"** +  
 **" LIMIT 10;"**;  
 }  
}

**package** twitter.classification.api.persist.jdbc.queries;  
  
**import** twitter.classification.common.persist.jdbc.queries.DbQuery;  
  
**public class** TimeLineForHashtagsDbQuery **implements** DbQuery {  
  
 */\*\*  
 \* Timeline DB query for a particular hashtag  
 \*  
 \** ***@return*** *sql to fetch the results of classifications in last 5 hours  
 \*/* @Override  
 **public** String buildQuery() {  
   
 **return "SELECT"** +  
 **" (SELECT count(\*)"** +  
 **" FROM tweets"** +  
 **" JOIN hashtag\_tweet\_classifications"** +  
 **" ON hashtag\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN hashtags"** +  
 **" ON hashtags.id = hashtag\_tweet\_classifications.hashtag\_id"** +  
 **" JOIN classification\_types ON classification\_id = classification\_types.id"** +  
 **" WHERE classification\_value = 'rumour'"** +  
 **" AND hashtags.hashtag\_value = ?"** +  
 **" AND tweets.created\_on >= DATE\_SUB(NOW(), INTERVAL 1 HOUR)) count\_of\_rumours\_in\_last\_hour,"** +  
 **" (SELECT count(\*)"** +  
 **" FROM tweets"** +  
 **" JOIN hashtag\_tweet\_classifications ON hashtag\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN hashtags ON hashtags.id = hashtag\_tweet\_classifications.hashtag\_id"** +  
 **" JOIN classification\_types ON classification\_id = classification\_types.id"** +  
 **" WHERE classification\_value = 'non-rumour' AND hashtags.hashtag\_value = ? AND"** +  
 **" tweets.created\_on >= DATE\_SUB(NOW(), INTERVAL 1 HOUR)) count\_of\_non\_rumours\_in\_last\_hour,"** +  
 **" (SELECT count(\*)"** +  
 **" FROM tweets"** +  
 **" JOIN hashtag\_tweet\_classifications ON hashtag\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN hashtags ON hashtags.id = hashtag\_tweet\_classifications.hashtag\_id"** +  
 **" JOIN classification\_types ON classification\_id = classification\_types.id"** +  
 **" WHERE classification\_value = 'rumour' AND hashtags.hashtag\_value = ? AND"** +  
 **" tweets.created\_on >= DATE\_SUB(NOW(), INTERVAL 2 HOUR) AND"** +  
 **" tweets.created\_on <= DATE\_SUB(NOW(), INTERVAL 1 HOUR)) count\_of\_rumours\_over\_an\_hour,"** +  
 **" (SELECT count(\*)"** +  
 **" FROM tweets"** +  
 **" JOIN hashtag\_tweet\_classifications ON hashtag\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN hashtags ON hashtags.id = hashtag\_tweet\_classifications.hashtag\_id"** +  
 **" JOIN classification\_types ON classification\_id = classification\_types.id"** +  
 **" WHERE classification\_value = 'non-rumour' AND hashtags.hashtag\_value = ? AND"** +  
 **" tweets.created\_on >= DATE\_SUB(NOW(), INTERVAL 2 HOUR) AND"** +  
 **" tweets.created\_on <= DATE\_SUB(NOW(), INTERVAL 1 HOUR)) count\_of\_non\_rumours\_over\_an\_hour,"** +  
 **" (SELECT count(\*)"** +  
 **" FROM tweets"** +  
 **" JOIN hashtag\_tweet\_classifications ON hashtag\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN hashtags ON hashtags.id = hashtag\_tweet\_classifications.hashtag\_id"** +  
 **" JOIN classification\_types ON classification\_id = classification\_types.id"** +  
 **" WHERE classification\_value = 'rumour' AND hashtags.hashtag\_value = ? AND"** +  
 **" tweets.created\_on >= DATE\_SUB(NOW(), INTERVAL 3 HOUR) AND"** +  
 **" tweets.created\_on <= DATE\_SUB(NOW(), INTERVAL 2 HOUR)) count\_of\_rumours\_over\_two\_hours,"** +  
 **" (SELECT count(\*)"** +  
 **" FROM tweets"** +  
 **" JOIN hashtag\_tweet\_classifications ON hashtag\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN hashtags ON hashtags.id = hashtag\_tweet\_classifications.hashtag\_id"** +  
 **" JOIN classification\_types ON classification\_id = classification\_types.id"** +  
 **" WHERE classification\_value = 'non-rumour' AND hashtags.hashtag\_value = ? AND"** +  
 **" tweets.created\_on >= DATE\_SUB(NOW(), INTERVAL 3 HOUR) AND"** +  
 **" tweets.created\_on <= DATE\_SUB(NOW(), INTERVAL 2 HOUR)) count\_of\_non\_rumours\_over\_two\_hours,"** +  
 **" (SELECT count(\*)"** +  
 **" FROM tweets"** +  
 **" JOIN hashtag\_tweet\_classifications ON hashtag\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN hashtags ON hashtags.id = hashtag\_tweet\_classifications.hashtag\_id"** +  
 **" JOIN classification\_types ON classification\_id = classification\_types.id"** +  
 **" WHERE classification\_value = 'rumour' AND hashtags.hashtag\_value = ? AND"** +  
 **" tweets.created\_on >= DATE\_SUB(NOW(), INTERVAL 4 HOUR) AND"** +  
 **" tweets.created\_on <= DATE\_SUB(NOW(), INTERVAL 3 HOUR)) count\_of\_rumours\_over\_three\_hours,"** +  
 **" (SELECT count(\*)"** +  
 **" FROM tweets"** +  
 **" JOIN hashtag\_tweet\_classifications ON hashtag\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN hashtags ON hashtags.id = hashtag\_tweet\_classifications.hashtag\_id"** +  
 **" JOIN classification\_types ON classification\_id = classification\_types.id"** +  
 **" WHERE classification\_value = 'non-rumour' AND hashtags.hashtag\_value = ? AND"** +  
 **" tweets.created\_on >= DATE\_SUB(NOW(), INTERVAL 4 HOUR) AND"** +  
 **" tweets.created\_on <= DATE\_SUB(NOW(), INTERVAL 3 HOUR)) count\_of\_non\_rumours\_over\_three\_hours,"** +  
 **" (SELECT count(\*)"** +  
 **" FROM tweets"** +  
 **" JOIN hashtag\_tweet\_classifications ON hashtag\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN hashtags ON hashtags.id = hashtag\_tweet\_classifications.hashtag\_id"** +  
 **" JOIN classification\_types ON classification\_id = classification\_types.id"** +  
 **" WHERE classification\_value = 'rumour' AND hashtags.hashtag\_value = ? AND"** +  
 **" tweets.created\_on >= DATE\_SUB(NOW(), INTERVAL 5 HOUR) AND"** +  
 **" tweets.created\_on <= DATE\_SUB(NOW(), INTERVAL 4 HOUR)) count\_of\_rumours\_over\_four\_hours,"** +  
 **" (SELECT count(\*)"** +  
 **" FROM tweets"** +  
 **" JOIN hashtag\_tweet\_classifications ON hashtag\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN hashtags ON hashtags.id = hashtag\_tweet\_classifications.hashtag\_id"** +  
 **" JOIN classification\_types ON classification\_id = classification\_types.id"** +  
 **" WHERE classification\_value = 'non-rumour' AND hashtags.hashtag\_value = ? AND"** +  
 **" tweets.created\_on >= DATE\_SUB(NOW(), INTERVAL 5 HOUR) AND"** +  
 **" tweets.created\_on <= DATE\_SUB(NOW(), INTERVAL 4 HOUR)) count\_of\_non\_rumours\_over\_four\_hours;"**;  
 }  
}

**package** twitter.classification.api.persist.jdbc.queries;  
  
**import** twitter.classification.common.persist.jdbc.queries.DbQuery;  
  
**public class** TimeLineForSearchTermDbQuery **implements** DbQuery {  
  
 */\*\*  
 \* Timeline DB query for a particular hashtag/user  
 \*  
 \** ***@return*** *sql to fetch the results of classifications in last 5 hours  
 \*/* @Override  
 **public** String buildQuery() {  
   
 **return "SELECT"** +  
 **" (SELECT count(\*)"** +  
 **" FROM tweets"** +  
 **" JOIN hashtag\_tweet\_classifications ON hashtag\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN hashtags ON hashtags.id = hashtag\_tweet\_classifications.hashtag\_id"** +  
 **" JOIN users\_tweet\_classifications ON users\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN users ON users.id = users\_tweet\_classifications.user\_id"** +  
 **" JOIN classification\_types ON classification\_id = classification\_types.id"** +  
 **" WHERE classification\_value = 'rumour' AND tweets.created\_on >= DATE\_SUB(NOW(), INTERVAL 1 HOUR) AND hashtags.hashtag\_value = ? OR users.username = ?) count\_of\_rumours\_in\_last\_hour,"** +  
 **" (SELECT count(\*)"** +  
 **" FROM tweets"** +  
 **" JOIN hashtag\_tweet\_classifications ON hashtag\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN hashtags ON hashtags.id = hashtag\_tweet\_classifications.hashtag\_id"** +  
 **" JOIN users\_tweet\_classifications ON users\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN users ON users.id = users\_tweet\_classifications.user\_id"** +  
 **" JOIN classification\_types ON classification\_id = classification\_types.id"** +  
 **" WHERE classification\_value = 'non-rumour' AND"** +  
 **" tweets.created\_on >= DATE\_SUB(NOW(), INTERVAL 1 HOUR) AND hashtags.hashtag\_value = ? OR users.username = ?) count\_of\_non\_rumours\_in\_last\_hour,"** +  
 **" (SELECT count(\*)"** +  
 **" FROM tweets"** +  
 **" JOIN hashtag\_tweet\_classifications ON hashtag\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN hashtags ON hashtags.id = hashtag\_tweet\_classifications.hashtag\_id"** +  
 **" JOIN users\_tweet\_classifications ON users\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN users ON users.id = users\_tweet\_classifications.user\_id"** +  
 **" JOIN classification\_types ON classification\_id = classification\_types.id"** +  
 **" WHERE classification\_value = 'rumour' AND"** +  
 **" tweets.created\_on >= DATE\_SUB(NOW(), INTERVAL 2 HOUR) AND"** +  
 **" tweets.created\_on <= DATE\_SUB(NOW(), INTERVAL 1 HOUR) AND hashtags.hashtag\_value = ? OR users.username = ?) count\_of\_rumours\_over\_an\_hour,"** +  
 **" (SELECT count(\*)"** +  
 **" FROM tweets"** +  
 **" JOIN hashtag\_tweet\_classifications ON hashtag\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN hashtags ON hashtags.id = hashtag\_tweet\_classifications.hashtag\_id"** +  
 **" JOIN users\_tweet\_classifications ON users\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN users ON users.id = users\_tweet\_classifications.user\_id"** +  
 **" JOIN classification\_types ON classification\_id = classification\_types.id"** +  
 **" WHERE classification\_value = 'non-rumour' AND"** +  
 **" tweets.created\_on >= DATE\_SUB(NOW(), INTERVAL 2 HOUR) AND"** +  
 **" tweets.created\_on <= DATE\_SUB(NOW(), INTERVAL 1 HOUR) AND hashtags.hashtag\_value = ? OR users.username = ?) count\_of\_non\_rumours\_over\_an\_hour,"** +  
 **" (SELECT count(\*)"** +  
 **" FROM tweets"** +  
 **" JOIN hashtag\_tweet\_classifications ON hashtag\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN hashtags ON hashtags.id = hashtag\_tweet\_classifications.hashtag\_id"** +  
 **" JOIN users\_tweet\_classifications ON users\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN users ON users.id = users\_tweet\_classifications.user\_id"** +  
 **" JOIN classification\_types ON classification\_id = classification\_types.id"** +  
 **" WHERE classification\_value = 'rumour' AND"** +  
 **" tweets.created\_on >= DATE\_SUB(NOW(), INTERVAL 3 HOUR) AND"** +  
 **" tweets.created\_on <= DATE\_SUB(NOW(), INTERVAL 2 HOUR) AND hashtags.hashtag\_value = ? OR users.username = ?) count\_of\_rumours\_over\_two\_hours,"** +  
 **" (SELECT count(\*)"** +  
 **" FROM tweets"** +  
 **" JOIN hashtag\_tweet\_classifications ON hashtag\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN hashtags ON hashtags.id = hashtag\_tweet\_classifications.hashtag\_id"** +  
 **" JOIN users\_tweet\_classifications ON users\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN users ON users.id = users\_tweet\_classifications.user\_id"** +  
 **" JOIN classification\_types ON classification\_id = classification\_types.id"** +  
 **" WHERE classification\_value = 'non-rumour' AND"** +  
 **" tweets.created\_on >= DATE\_SUB(NOW(), INTERVAL 3 HOUR) AND"** +  
 **" tweets.created\_on <= DATE\_SUB(NOW(), INTERVAL 2 HOUR) AND hashtags.hashtag\_value = ? OR users.username = ?) count\_of\_non\_rumours\_over\_two\_hours,"** +  
 **" (SELECT count(\*)"** +  
 **" FROM tweets"** +  
 **" JOIN hashtag\_tweet\_classifications ON hashtag\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN hashtags ON hashtags.id = hashtag\_tweet\_classifications.hashtag\_id"** +  
 **" JOIN users\_tweet\_classifications ON users\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN users ON users.id = users\_tweet\_classifications.user\_id"** +  
 **" JOIN classification\_types ON classification\_id = classification\_types.id"** +  
 **" WHERE classification\_value = 'rumour' AND"** +  
 **" tweets.created\_on >= DATE\_SUB(NOW(), INTERVAL 4 HOUR) AND"** +  
 **" tweets.created\_on <= DATE\_SUB(NOW(), INTERVAL 3 HOUR) AND hashtags.hashtag\_value = ? OR users.username = ?) count\_of\_rumours\_over\_three\_hours,"** +  
 **" (SELECT count(\*)"** +  
 **" FROM tweets"** +  
 **" JOIN hashtag\_tweet\_classifications ON hashtag\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN hashtags ON hashtags.id = hashtag\_tweet\_classifications.hashtag\_id"** +  
 **" JOIN users\_tweet\_classifications ON users\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN users ON users.id = users\_tweet\_classifications.user\_id"** +  
 **" JOIN classification\_types ON classification\_id = classification\_types.id"** +  
 **" WHERE classification\_value = 'non-rumour' AND"** +  
 **" tweets.created\_on >= DATE\_SUB(NOW(), INTERVAL 4 HOUR) AND"** +  
 **" tweets.created\_on <= DATE\_SUB(NOW(), INTERVAL 3 HOUR) AND hashtags.hashtag\_value = ? OR users.username = ?) count\_of\_non\_rumours\_over\_three\_hours,"** +  
 **" (SELECT count(\*)"** +  
 **" FROM tweets"** +  
 **" JOIN hashtag\_tweet\_classifications ON hashtag\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN hashtags ON hashtags.id = hashtag\_tweet\_classifications.hashtag\_id"** +  
 **" JOIN users\_tweet\_classifications ON users\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN users ON users.id = users\_tweet\_classifications.user\_id"** +  
 **" JOIN classification\_types ON classification\_id = classification\_types.id"** +  
 **" WHERE classification\_value = 'rumour' AND"** +  
 **" tweets.created\_on >= DATE\_SUB(NOW(), INTERVAL 5 HOUR) AND"** +  
 **" tweets.created\_on <= DATE\_SUB(NOW(), INTERVAL 4 HOUR) AND hashtags.hashtag\_value = ? OR users.username = ?) count\_of\_rumours\_over\_four\_hours,"** +  
 **" (SELECT count(\*)"** +  
 **" FROM tweets"** +  
 **" JOIN hashtag\_tweet\_classifications ON hashtag\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN hashtags ON hashtags.id = hashtag\_tweet\_classifications.hashtag\_id"** +  
 **" JOIN users\_tweet\_classifications ON users\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN users ON users.id = users\_tweet\_classifications.user\_id"** +  
 **" JOIN classification\_types ON classification\_id = classification\_types.id"** +  
 **" WHERE classification\_value = 'non-rumour' AND"** +  
 **" tweets.created\_on >= DATE\_SUB(NOW(), INTERVAL 5 HOUR) AND"** +  
 **" tweets.created\_on <= DATE\_SUB(NOW(), INTERVAL 4 HOUR) AND hashtags.hashtag\_value = ? OR users.username = ?) count\_of\_non\_rumours\_over\_four\_hours;"**;  
 }  
}

**package** twitter.classification.api.persist.jdbc.queries;  
  
**import** twitter.classification.common.persist.jdbc.queries.DbQuery;  
  
**public class** TimeLineForUsersDbQuery **implements** DbQuery {  
  
 */\*\*  
 \* Timeline DB query for a particular user  
 \*  
 \** ***@return*** *sql to fetch the results of classifications in last 5 hours  
 \*/* @Override  
 **public** String buildQuery() {  
   
 **return "SELECT"** +  
 **" (SELECT count(\*)"** +  
 **" FROM tweets"** +  
 **" JOIN users\_tweet\_classifications"** +  
 **" ON users\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN users"** +  
 **" ON users.id = users\_tweet\_classifications.user\_id"** +  
 **" JOIN classification\_types ON classification\_id = classification\_types.id"** +  
 **" WHERE classification\_value = 'rumour'"** +  
 **" AND users.username = ?"** +  
 **" AND tweets.created\_on >= DATE\_SUB(NOW(), INTERVAL 1 HOUR)) count\_of\_rumours\_in\_last\_hour,"** +  
 **" (SELECT count(\*)"** +  
 **" FROM tweets"** +  
 **" JOIN users\_tweet\_classifications ON users\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN users ON users.id = users\_tweet\_classifications.user\_id"** +  
 **" JOIN classification\_types ON classification\_id = classification\_types.id"** +  
 **" WHERE classification\_value = 'non-rumour' AND users.username = ? AND"** +  
 **" tweets.created\_on >= DATE\_SUB(NOW(), INTERVAL 1 HOUR)) count\_of\_non\_rumours\_in\_last\_hour,"** +  
 **" (SELECT count(\*)"** +  
 **" FROM tweets"** +  
 **" JOIN users\_tweet\_classifications ON users\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN users ON users.id = users\_tweet\_classifications.user\_id"** +  
 **" JOIN classification\_types ON classification\_id = classification\_types.id"** +  
 **" WHERE classification\_value = 'rumour' AND users.username = ? AND"** +  
 **" tweets.created\_on >= DATE\_SUB(NOW(), INTERVAL 2 HOUR) AND"** +  
 **" tweets.created\_on <= DATE\_SUB(NOW(), INTERVAL 1 HOUR)) count\_of\_rumours\_over\_an\_hour,"** +  
 **" (SELECT count(\*)"** +  
 **" FROM tweets"** +  
 **" JOIN users\_tweet\_classifications ON users\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN users ON users.id = users\_tweet\_classifications.user\_id"** +  
 **" JOIN classification\_types ON classification\_id = classification\_types.id"** +  
 **" WHERE classification\_value = 'non-rumour' AND users.username = ? AND"** +  
 **" tweets.created\_on >= DATE\_SUB(NOW(), INTERVAL 2 HOUR) AND"** +  
 **" tweets.created\_on <= DATE\_SUB(NOW(), INTERVAL 1 HOUR)) count\_of\_non\_rumours\_over\_an\_hour,"** +  
 **" (SELECT count(\*)"** +  
 **" FROM tweets"** +  
 **" JOIN users\_tweet\_classifications ON users\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN users ON users.id = users\_tweet\_classifications.user\_id"** +  
 **" JOIN classification\_types ON classification\_id = classification\_types.id"** +  
 **" WHERE classification\_value = 'rumour' AND users.username = ? AND"** +  
 **" tweets.created\_on >= DATE\_SUB(NOW(), INTERVAL 3 HOUR) AND"** +  
 **" tweets.created\_on <= DATE\_SUB(NOW(), INTERVAL 2 HOUR)) count\_of\_rumours\_over\_two\_hours,"** +  
 **" (SELECT count(\*)"** +  
 **" FROM tweets"** +  
 **" JOIN users\_tweet\_classifications ON users\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN users ON users.id = users\_tweet\_classifications.user\_id"** +  
 **" JOIN classification\_types ON classification\_id = classification\_types.id"** +  
 **" WHERE classification\_value = 'non-rumour' AND users.username = ? AND"** +  
 **" tweets.created\_on >= DATE\_SUB(NOW(), INTERVAL 3 HOUR) AND"** +  
 **" tweets.created\_on <= DATE\_SUB(NOW(), INTERVAL 2 HOUR)) count\_of\_non\_rumours\_over\_two\_hours,"** +  
 **" (SELECT count(\*)"** +  
 **" FROM tweets"** +  
 **" JOIN users\_tweet\_classifications ON users\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN users ON users.id = users\_tweet\_classifications.user\_id"** +  
 **" JOIN classification\_types ON classification\_id = classification\_types.id"** +  
 **" WHERE classification\_value = 'rumour' AND users.username = ? AND"** +  
 **" tweets.created\_on >= DATE\_SUB(NOW(), INTERVAL 4 HOUR) AND"** +  
 **" tweets.created\_on <= DATE\_SUB(NOW(), INTERVAL 3 HOUR)) count\_of\_rumours\_over\_three\_hours,"** +  
 **" (SELECT count(\*)"** +  
 **" FROM tweets"** +  
 **" JOIN users\_tweet\_classifications ON users\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN users ON users.id = users\_tweet\_classifications.user\_id"** +  
 **" JOIN classification\_types ON classification\_id = classification\_types.id"** +  
 **" WHERE classification\_value = 'non-rumour' AND users.username = ? AND"** +  
 **" tweets.created\_on >= DATE\_SUB(NOW(), INTERVAL 4 HOUR) AND"** +  
 **" tweets.created\_on <= DATE\_SUB(NOW(), INTERVAL 3 HOUR)) count\_of\_non\_rumours\_over\_three\_hours,"** +  
 **" (SELECT count(\*)"** +  
 **" FROM tweets"** +  
 **" JOIN users\_tweet\_classifications ON users\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN users ON users.id = users\_tweet\_classifications.user\_id"** +  
 **" JOIN classification\_types ON classification\_id = classification\_types.id"** +  
 **" WHERE classification\_value = 'rumour' AND users.username = ? AND"** +  
 **" tweets.created\_on >= DATE\_SUB(NOW(), INTERVAL 5 HOUR) AND"** +  
 **" tweets.created\_on <= DATE\_SUB(NOW(), INTERVAL 4 HOUR)) count\_of\_rumours\_over\_four\_hours,"** +  
 **" (SELECT count(\*)"** +  
 **" FROM tweets"** +  
 **" JOIN users\_tweet\_classifications ON users\_tweet\_classifications.tweet\_id = tweets.id"** +  
 **" JOIN users ON users.id = users\_tweet\_classifications.user\_id"** +  
 **" JOIN classification\_types ON classification\_id = classification\_types.id"** +  
 **" WHERE classification\_value = 'non-rumour' AND users.username = ? AND"** +  
 **" tweets.created\_on >= DATE\_SUB(NOW(), INTERVAL 5 HOUR) AND"** +  
 **" tweets.created\_on <= DATE\_SUB(NOW(), INTERVAL 4 HOUR)) count\_of\_non\_rumours\_over\_four\_hours;"**;  
 }  
}

**package** twitter.classification.api.persist.jdbc.queries;  
  
**import** twitter.classification.common.persist.jdbc.queries.DbQuery;  
  
**public class** TweetsForHashtagWordCloudDbQuery **implements** DbQuery {  
  
 */\*\*  
 \* Fetch the tweet text for a hashtag to present in a word cloud  
 \*  
 \** ***@return*** *sql for wordcloud results  
 \*/* @Override  
 **public** String buildQuery() {  
  
 **return "SELECT hashtags.hashtag\_value, tweets.processed\_tweet\_text "** +  
 **"FROM hashtags "** +  
 **" JOIN hashtag\_tweet\_classifications ON hashtags.id = hashtag\_tweet\_classifications.hashtag\_id "** +  
 **" JOIN tweets ON hashtag\_tweet\_classifications.tweet\_id = tweets.id "** +  
 **"WHERE hashtags.hashtag\_value = ?;"**;  
 }  
}

**package** twitter.classification.api.persist.jdbc.queries;  
  
**import** twitter.classification.common.persist.jdbc.queries.DbQuery;  
  
**public class** TweetsForSearchTermWordCloudDbQuery **implements** DbQuery {  
  
 */\*\*  
 \* Fetch the tweet text for a user/hashtag to present in a word cloud  
 \*  
 \** ***@return*** *sql for wordcloud results  
 \*/* @Override  
 **public** String buildQuery() {  
  
 **return "SELECT tweets.processed\_tweet\_text "** +  
 **"FROM hashtags "** +  
 **" JOIN hashtag\_tweet\_classifications ON hashtags.id = hashtag\_tweet\_classifications.hashtag\_id "** +  
 **" JOIN tweets ON hashtag\_tweet\_classifications.tweet\_id = tweets.id "** +  
 **" JOIN users\_tweet\_classifications ON users\_tweet\_classifications.tweet\_id = tweets.id "** +  
 **" JOIN users ON users\_tweet\_classifications.user\_id = users.id "** +  
 **"WHERE hashtags.hashtag\_value = ? OR users.username = ?;"**;  
 }  
}

**package** twitter.classification.api.persist.jdbc.queries;  
  
**import** twitter.classification.common.persist.jdbc.queries.DbQuery;  
  
**public class** TweetsForUserWordCloudDbQuery **implements** DbQuery {  
  
 */\*\*  
 \* Fetch the tweet text for a user to present in a word cloud  
 \*  
 \** ***@return*** *sql for wordcloud results  
 \*/* @Override  
 **public** String buildQuery() {  
  
 **return "SELECT users.username, tweets.processed\_tweet\_text "** +  
 **"FROM users"** +  
 **" JOIN users\_tweet\_classifications ON users.id = users\_tweet\_classifications.user\_id"** +  
 **" JOIN tweets ON users\_tweet\_classifications.tweet\_id = tweets.id "** +  
 **"WHERE users.username = ?;"**;  
 }  
}

## resource

**package** twitter.classification.api.resource;  
  
**import** javax.inject.Inject;  
**import** javax.inject.Singleton;  
**import** javax.ws.rs.GET;  
**import** javax.ws.rs.Path;  
**import** javax.ws.rs.Produces;  
**import** javax.ws.rs.core.MediaType;  
  
**import** twitter.classification.api.service.DashBoardOverviewService;  
**import** twitter.classification.api.service.DashBoardServicesStatusService;  
**import** twitter.classification.common.exceptions.ProcessingClientException;  
**import** twitter.classification.common.models.DashBoardOverviewResponse;  
**import** twitter.classification.common.models.DashBoardServiceStatusResponse;  
  
@Singleton  
@Path(**"/dashboard"**)  
**public class** DashBoardOverviewDataResource {  
  
 **private** DashBoardOverviewService **dashBoardOverviewService**;  
 **private** DashBoardServicesStatusService **dashBoardServicesStatusService**;  
  
 @Inject  
 **public** DashBoardOverviewDataResource(  
 DashBoardOverviewService dashBoardOverviewService,  
 DashBoardServicesStatusService dashBoardServicesStatusService  
 ) {  
  
 **this**.**dashBoardOverviewService** = dashBoardOverviewService;  
 **this**.**dashBoardServicesStatusService** = dashBoardServicesStatusService;  
 }  
  
 */\*\*  
 \* For retrieving the results for the dashboard overview  
 \*  
 \** ***@return*** *json of the dashboard overview results  
 \*/* @GET  
 @Path(**"/overview"**)  
 @Produces(MediaType.APPLICATION\_JSON)  
 **public** DashBoardOverviewResponse getDashBoardOverview() {  
  
 **return** dashBoardOverviewService.retrieve();  
 }  
  
 */\*\*  
 \* For retrieving the status of the running services  
 \*  
 \** ***@return*** *status of the running services  
 \** ***@throws*** *ProcessingClientException  
 \*/* @GET  
 @Path(**"/services/status"**)  
 @Produces(MediaType.APPLICATION\_JSON)  
 **public** DashBoardServiceStatusResponse getDashBoardServicesStatus() **throws** ProcessingClientException {  
  
 **return** dashBoardServicesStatusService.status();  
 }  
}

**package** twitter.classification.api.resource;  
  
**import** java.util.List;  
  
**import** javax.inject.Inject;  
**import** javax.inject.Singleton;  
**import** javax.ws.rs.GET;  
**import** javax.ws.rs.Path;  
**import** javax.ws.rs.PathParam;  
  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** twitter.classification.api.service.HashtagResultsService;  
**import** twitter.classification.common.models.ClassificationValueForTweets;  
**import** twitter.classification.common.models.TimeLineForTweets;  
  
@Singleton  
@Path(**"/hashtags/{value}"**)  
**public class** HashtagsResource {  
  
 **private static final** Logger ***logger*** = LoggerFactory.*getLogger*(HashtagsResource.**class**);  
  
 **private** HashtagResultsService **hashtagResultsService**;  
  
 @Inject  
 **public** HashtagsResource(HashtagResultsService hashtagResultsService) {  
  
 **this**.**hashtagResultsService** = hashtagResultsService;  
 }  
  
 */\*\*  
 \* Paginated results for a hashtag  
 \*  
 \** ***@param value*** *\** ***@param limit*** *\** ***@param offset*** *\** ***@return*** *paginated results  
 \*/* @GET  
 @Path(**"/{offset:[0-9]+}/{limit:[0-9]+}"**)  
 **public** List<ClassificationValueForTweets> getPaginatedResults(  
 @PathParam(**"value"**) String value,  
 @PathParam(**"limit"**) **int** limit,  
 @PathParam(**"offset"**) **int** offset  
 ) {  
  
 ***logger***.debug(**"Path params for value is {}, limit is {}, offset is {}"**, value, limit, offset);  
  
 **return hashtagResultsService**.getPaginatedResultsHashtag(value, offset, limit);  
 }  
  
 */\*\*  
 \* Get the time line for a hashtag  
 \*  
 \** ***@param value*** *\** ***@return*** *timeline results  
 \*/* @GET  
 @Path(**"/timeline"**)  
 **public** TimeLineForTweets getTimeLineForHashtag(  
 @PathParam(**"value"**) String value  
 ) {  
  
 **return hashtagResultsService**.getTimeLineForHashtag(value);  
 }  
}

**package** twitter.classification.api.resource;  
  
**import** java.io.IOException;  
**import** java.util.List;  
  
**import** javax.inject.Inject;  
**import** javax.inject.Singleton;  
**import** javax.ws.rs.GET;  
**import** javax.ws.rs.Path;  
**import** javax.ws.rs.PathParam;  
  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** twitter.classification.api.service.SearchTermResultService;  
**import** twitter.classification.api.service.SuggestedSearchResultsService;  
**import** twitter.classification.common.models.SuggestedSearchResult;  
**import** twitter.classification.common.models.ClassificationValueForTweets;  
**import** twitter.classification.common.models.SearchResultsResponse;  
**import** twitter.classification.common.models.SuggestedSearchTermsResponse;  
**import** twitter.classification.common.models.TimeLineForTweets;  
  
@Singleton  
@Path(**"/search"**)  
**public class** SearchResource {  
  
 **private static final** Logger ***logger*** = LoggerFactory.*getLogger*(SearchResource.**class**);  
  
 **private** SearchTermResultService **searchTermResultService**;  
 **private** SuggestedSearchResultsService **suggestedSearchResultsService**;  
  
 @Inject  
 **public** SearchResource(  
 SearchTermResultService searchTermResultService,  
 SuggestedSearchResultsService suggestedSearchResultsService  
 ) {  
  
 **this**.**searchTermResultService** = searchTermResultService;  
 **this**.**suggestedSearchResultsService** = suggestedSearchResultsService;  
 }  
  
 */\*\*  
 \* Search results for a certain search term  
 \*  
 \** ***@param searchTerm*** *\** ***@return*** *search results  
 \** ***@throws*** *IOException  
 \*/* @GET  
 @Path(**"/{value}"**)  
 **public** SearchResultsResponse get(  
 @PathParam(**"value"**) String searchTerm  
 ) **throws** IOException {  
  
 **return searchTermResultService**.get(searchTerm);  
 }  
  
 */\*\*  
 \* Paginated table results for a particular search term  
 \*  
 \** ***@param searchValue*** *\** ***@param limit*** *\** ***@param offset*** *\** ***@return*** *paginated results  
 \*/* @GET  
 @Path(**"/{value}/{offset:[0-9]+}/{limit:[0-9]+}"**)  
 **public** List<ClassificationValueForTweets> getPaginatedResults(  
 @PathParam(**"value"**) String searchValue,  
 @PathParam(**"limit"**) **int** limit,  
 @PathParam(**"offset"**) **int** offset  
 ) {  
  
 ***logger***.debug(**"Path params for value is {}, limit is {}, offset is {}"**, searchValue, limit, offset);  
  
 **return searchTermResultService**.getPaginatedResults(searchValue, offset, limit);  
 }  
  
 */\*\*  
 \* Return the suggestions for search terms  
 \*  
 \** ***@return*** *search term suggestions  
 \*/* @GET  
 @Path(**"/suggestions"**)  
 **public** SuggestedSearchTermsResponse getSuggestedSearchResults() {  
  
 **return suggestedSearchResultsService**.get();  
 }  
  
 */\*\*  
 \* Timeline for a particular search term  
 \*  
 \** ***@param value*** *\** ***@return*** *timeline results  
 \*/* @GET  
 @Path(**"/{value}/timeline"**)  
 **public** TimeLineForTweets getTimeLineForSearchTerm(  
 @PathParam(**"value"**) String value  
 ) {  
  
 **return searchTermResultService**.getTimeLineForSearchTerm(value);  
 }  
}

**package** twitter.classification.api.resource;  
  
**import** java.io.IOException;  
  
**import** javax.inject.Inject;  
**import** javax.inject.Singleton;  
**import** javax.ws.rs.GET;  
**import** javax.ws.rs.Path;  
**import** javax.ws.rs.Produces;  
**import** javax.ws.rs.core.MediaType;  
  
**import** twitter.classification.api.service.TopHashTagResultService;  
**import** twitter.classification.common.models.TopHashtagsResponse;  
  
@Singleton  
@Path(**"/top/hashtags"**)  
**public class** TopHashTagsResource {  
  
 **private** TopHashTagResultService **topHashTagResultService**;  
  
 @Inject  
 **public** TopHashTagsResource(  
 TopHashTagResultService topHashTagResultService  
 ) {  
  
 **this**.**topHashTagResultService** = topHashTagResultService;  
 }  
  
 */\*\*  
 \* Top hashtag results for the hashtag page  
 \*  
 \** ***@return*** *top hashtags  
 \** ***@throws*** *IOException  
 \*/* @GET  
 @Produces(MediaType.***APPLICATION\_JSON***)  
 **public** TopHashtagsResponse getTopHashtagResults() **throws** IOException {  
  
 **return topHashTagResultService**.get();  
 }  
}

**package** twitter.classification.api.resource;  
  
**import** java.io.IOException;  
  
**import** javax.inject.Inject;  
**import** javax.inject.Singleton;  
**import** javax.ws.rs.GET;  
**import** javax.ws.rs.Path;  
**import** javax.ws.rs.Produces;  
**import** javax.ws.rs.core.MediaType;  
  
**import** twitter.classification.api.persist.jdbc.models.TopUsersClassificationModel;  
**import** twitter.classification.api.service.TopUserResultService;  
**import** twitter.classification.common.models.TopUsersResponse;  
  
@Singleton  
@Path(**"/top/users"**)  
**public class** TopUsersResource {  
  
 **private** TopUserResultService **topUserResultService**;  
  
 @Inject  
 **public** TopUsersResource(  
 TopUserResultService topUserResultService  
 ) {  
  
 **this**.**topUserResultService** = topUserResultService;  
 }  
  
 */\*\*  
 \* Top users results  
 \*  
 \** ***@return*** *top users results  
 \** ***@throws*** *IOException  
 \*/* @GET  
 @Produces(MediaType.***APPLICATION\_JSON***)  
 **public** TopUsersResponse getTopHashtagResults() **throws** IOException {  
  
 **return topUserResultService**.get();  
 }  
}

**package** twitter.classification.api.resource;  
  
**import** java.util.List;  
  
**import** javax.inject.Inject;  
**import** javax.inject.Singleton;  
**import** javax.ws.rs.GET;  
**import** javax.ws.rs.Path;  
**import** javax.ws.rs.PathParam;  
  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** twitter.classification.api.service.UserResultsService;  
**import** twitter.classification.common.models.ClassificationValueForTweets;  
**import** twitter.classification.common.models.TimeLineForTweets;  
  
@Singleton  
@Path(**"/users/{value}"**)  
**public class** UsersResource {  
  
 **private static final** Logger ***logger*** = LoggerFactory.*getLogger*(UsersResource.**class**);  
  
 **private** UserResultsService **userResultsService**;  
  
 @Inject  
 **public** UsersResource(UserResultsService userResultsService) {  
  
 **this**.**userResultsService** = userResultsService;  
 }  
  
 */\*\*  
 \* Paginated results for the users table  
 \*  
 \** ***@param value*** *\** ***@param limit*** *\** ***@param offset*** *\** ***@return*** *paginated results  
 \*/* @GET  
 @Path(**"/{offset:[0-9]+}/{limit:[0-9]+}"**)  
 **public** List<ClassificationValueForTweets> getPaginatedResults(  
 @PathParam(**"value"**) String value,  
 @PathParam(**"limit"**) **int** limit,  
 @PathParam(**"offset"**) **int** offset  
 ) {  
  
 ***logger***.debug(**"Path params for value is {}, limit is {}, offset is {}"**, value, limit, offset);  
  
 **return userResultsService**.getPaginatedUserResults(value, offset, limit);  
 }  
  
 */\*\*  
 \* Timeline for a username  
 \*  
 \** ***@param value*** *\** ***@return*** *timeline results  
 \*/* @GET  
 @Path(**"/timeline"**)  
 **public** TimeLineForTweets getTimeLineForUsername(  
 @PathParam(**"value"**) String value  
 ) {  
  
 **return userResultsService**.getTimeLineForUsername(value);  
 }  
}

## service

**package** twitter.classification.api.service;  
  
**import** java.util.List;  
  
**import** javax.inject.Inject;  
  
**import** twitter.classification.api.persist.jdbc.SelectDashBoardOverviewValuesDao;  
**import** twitter.classification.api.persist.jdbc.models.DashBoardOverviewModel;  
**import** twitter.classification.common.models.DashBoardOverviewResponse;  
**import** twitter.classification.common.persist.DbConnection;  
  
**public class** DashBoardOverviewService {  
  
 **private** SelectDashBoardOverviewValuesDao **selectDashBoardOverviewValuesDao**;  
  
 @Inject  
 **public** DashBoardOverviewService(SelectDashBoardOverviewValuesDao selectDashBoardOverviewValuesDao) {  
  
 **this**.**selectDashBoardOverviewValuesDao** = selectDashBoardOverviewValuesDao;  
 }  
  
 */\*\*  
 \* Retrieve the results for the dashboard overview information, such as total count of rumours, non-rumours  
 \* etc.  
 \*  
 \** ***@return*** *overview results  
 \*/* @DbConnection  
 **public** DashBoardOverviewResponse retrieve() {  
  
 List<DashBoardOverviewModel> dashBoardOverviewModels = **selectDashBoardOverviewValuesDao**.select();  
  
 **if** (dashBoardOverviewModels != **null** && !dashBoardOverviewModels.isEmpty()) {  
  
 DashBoardOverviewModel dashBoardOverviewModel = dashBoardOverviewModels.get(0);  
  
 DashBoardOverviewResponse dashBoardOverviewResponse = **new** DashBoardOverviewResponse();  
  
 dashBoardOverviewResponse.setTotalClassifications(dashBoardOverviewModel.getTotalCountOfClassifications().intValue());  
 dashBoardOverviewResponse.setTotalHashtags(dashBoardOverviewModel.getCountOfHashtags().intValue());  
 dashBoardOverviewResponse.setTotalNonRumours(dashBoardOverviewModel.getCountOfNonRumours().intValue());  
 dashBoardOverviewResponse.setTotalRumours(dashBoardOverviewModel.getCountOfRumours().intValue());  
 dashBoardOverviewResponse.setTotalUsernames(dashBoardOverviewModel.getCountOfUsers().intValue());  
 dashBoardOverviewResponse.setTotalTweets(dashBoardOverviewModel.getCountOfTweets().intValue());  
  
 **return** dashBoardOverviewResponse;  
 }  
  
 **return new** DashBoardOverviewResponse().setAllToZero();  
 }  
}

**package** twitter.classification.api.service;  
  
**import** javax.inject.Inject;  
  
**import** twitter.classification.api.client.ClassifierStatusClient;  
**import** twitter.classification.api.client.PreProcessorStatusClient;  
**import** twitter.classification.api.client.TwitterStreamClient;  
**import** twitter.classification.api.persist.jdbc.TestDatabaseConnectionDao;  
**import** twitter.classification.common.exceptions.ProcessingClientException;  
**import** twitter.classification.common.models.ClassifierStatusResponse;  
**import** twitter.classification.common.models.DashBoardServiceStatusResponse;  
**import** twitter.classification.common.models.PreProcessorStatusResponse;  
**import** twitter.classification.common.models.ServiceItem;  
**import** twitter.classification.common.models.TwitterStreamResponse;  
**import** twitter.classification.common.persist.DbConnection;  
  
**public class** DashBoardServicesStatusService {  
  
 **private** TwitterStreamClient **twitterStreamClient**;  
 **private** TestDatabaseConnectionDao **testDatabaseConnectionDao**;  
 **private** ClassifierStatusClient **classifierStatusClient**;  
 **private** PreProcessorStatusClient **preProcessorStatusClient**;  
  
 @Inject  
 **public** DashBoardServicesStatusService(  
 TwitterStreamClient twitterStreamClient,  
 TestDatabaseConnectionDao testDatabaseConnectionDao,  
 ClassifierStatusClient classifierStatusClient,  
 PreProcessorStatusClient preProcessorStatusClient  
 ) {  
  
 **this**.**twitterStreamClient** = twitterStreamClient;  
 **this**.**testDatabaseConnectionDao** = testDatabaseConnectionDao;  
 **this**.**classifierStatusClient** = classifierStatusClient;  
 **this**.**preProcessorStatusClient** = preProcessorStatusClient;  
 }  
  
 */\*\*  
 \* Status results of the running services  
 \*  
 \** ***@return*** *status results  
 \** ***@throws*** *ProcessingClientException  
 \*/* @DbConnection  
 **public** DashBoardServiceStatusResponse status() **throws** ProcessingClientException {  
  
 TwitterStreamResponse twitterStreamResponse = **twitterStreamClient**.isRunning();  
 **boolean** isDatabaseRunning = **testDatabaseConnectionDao**.test();  
 ClassifierStatusResponse classifierStatusResponse = **classifierStatusClient**.isRunning();  
 PreProcessorStatusResponse preProcessorStatusResponse = **preProcessorStatusClient**.isRunning();  
  
 ServiceItem twitterService = **new** ServiceItem(**"Stream"**, twitterStreamResponse.getRunning(), twitterStreamResponse.getFilterList());  
 ServiceItem databaseService = **new** ServiceItem(**"Database"**, isDatabaseRunning);  
 *// queue performs a healthcheck which if it fails, no service can start - so it always will be running* ServiceItem queueService = **new** ServiceItem(**"Queue"**, **true**);  
 ServiceItem classifierService = **new** ServiceItem(**"Classifier"**, classifierStatusResponse.getRunning());  
 ServiceItem preProcessorService = **new** ServiceItem(**"Pre-Processor"**, preProcessorStatusResponse.getRunning());  
  
 DashBoardServiceStatusResponse dashBoardServiceStatusResponse = **new** DashBoardServiceStatusResponse();  
 dashBoardServiceStatusResponse.addServiceItem(twitterService);  
 dashBoardServiceStatusResponse.addServiceItem(databaseService);  
 dashBoardServiceStatusResponse.addServiceItem(queueService);  
 dashBoardServiceStatusResponse.addServiceItem(classifierService);  
 dashBoardServiceStatusResponse.addServiceItem(preProcessorService);  
  
 **return** dashBoardServiceStatusResponse;  
 }  
}

**package** twitter.classification.api.service;  
  
**import** java.util.ArrayList;  
**import** java.util.List;  
  
**import** javax.inject.Inject;  
  
**import** twitter.classification.api.persist.jdbc.PaginatedHashtagTweetsDao;  
**import** twitter.classification.api.persist.jdbc.TimeLineForHashtagsDao;  
**import** twitter.classification.api.persist.jdbc.models.TimeLineForTweetsModel;  
**import** twitter.classification.common.models.ClassificationValueForTweets;  
**import** twitter.classification.common.models.TimeLineForTweets;  
**import** twitter.classification.common.persist.DbConnection;  
  
**public class** HashtagResultsService {  
  
 **private** PaginatedHashtagTweetsDao **paginatedHashtagTweetsDao**;  
 **private** TimeLineForHashtagsDao **timeLineForHashtagsDao**;  
  
 @Inject  
 **public** HashtagResultsService(  
 PaginatedHashtagTweetsDao paginatedHashtagTweetsDao,  
 TimeLineForHashtagsDao timeLineForHashtagsDao  
 ) {  
  
 **this**.**paginatedHashtagTweetsDao** = paginatedHashtagTweetsDao;  
 **this**.**timeLineForHashtagsDao** = timeLineForHashtagsDao;  
 }  
  
 */\*\*  
 \* Retrieve the paginated table results for a hashtag  
 \*  
 \** ***@param hashtag*** *\** ***@param offset*** *\** ***@param limit*** *\** ***@return*** *paginated results  
 \*/* @DbConnection  
 **public** List<ClassificationValueForTweets> getPaginatedResultsHashtag(String hashtag, **int** offset, **int** limit) {  
  
 **return new** PaginatedResultsService().paginatedResults(**new** ArrayList<>(), **paginatedHashtagTweetsDao**.get(hashtag, offset, limit));  
 }  
  
 */\*\*  
 \* Retrieve the timeline for a hashtag  
 \*  
 \** ***@param hashtag*** *\** ***@return*** *timeline  
 \*/* @DbConnection  
 **public** TimeLineForTweets getTimeLineForHashtag(String hashtag) {  
  
 TimeLineForTweetsModel timeLineForTweetsModel = **timeLineForHashtagsDao**.get(hashtag).get(0);  
  
 **return new** TimeLineForTweets()  
 .setNonRumoursLastHour(timeLineForTweetsModel.getCountOfNonRumoursLastHour())  
 .setRumoursLastHour(timeLineForTweetsModel.getCountOfRumoursLastHour())  
 .setNonRumoursOverOneHour(timeLineForTweetsModel.getCountOfNonRumoursOverAnHour())  
 .setRumoursOverOneHour(timeLineForTweetsModel.getCountOfRumoursOverAnHour())  
 .setNonRumoursOverTwoHour(timeLineForTweetsModel.getCountOfNonRumoursOverTwoHours())  
 .setRumoursOverTwoHour(timeLineForTweetsModel.getCountOfRumoursOverTwoHours())  
 .setNonRumoursOverThreeHour(timeLineForTweetsModel.getCountOfNonRumoursOverThreeHours())  
 .setRumoursOverThreeHour(timeLineForTweetsModel.getCountOfRumoursOverThreeHours())  
 .setNonRumoursOverFourHour(timeLineForTweetsModel.getCountOfNonRumoursOverFourHours())  
 .setRumoursOverFourHour(timeLineForTweetsModel.getCountOfRumoursOverFourHours());  
 }  
}

**package** twitter.classification.api.service;  
  
**import** java.util.List;  
  
**import** twitter.classification.api.persist.jdbc.models.PaginatedTweetsModel;  
**import** twitter.classification.common.models.ClassificationValueForTweets;  
  
**public class** PaginatedResultsService {  
  
 */\*\*  
 \* Reusable method to return a paginated results list for the paginated tweets  
 \*  
 \** ***@param classificationValueForTweetsList*** *\** ***@param paginatedTweetsModels*** *\** ***@return*** *paginated results  
 \*/* **public** List<ClassificationValueForTweets> paginatedResults(List<ClassificationValueForTweets> classificationValueForTweetsList, List<PaginatedTweetsModel> paginatedTweetsModels) {  
  
 **for** (PaginatedTweetsModel paginatedTweetsModel : paginatedTweetsModels) {  
  
 ClassificationValueForTweets classificationValueForTweets = **new** ClassificationValueForTweets();  
  
 classificationValueForTweets.setId(paginatedTweetsModel.getId().intValue());  
 classificationValueForTweets.setTweetText(paginatedTweetsModel.getTweetText());  
 classificationValueForTweets.setClassificationValue(paginatedTweetsModel.getClassificationValue());  
  
 classificationValueForTweetsList.add(classificationValueForTweets);  
 }  
  
 **return** classificationValueForTweetsList;  
 }  
}

**package** twitter.classification.api.service;  
  
**import** java.io.IOException;  
**import** java.util.ArrayList;  
**import** java.util.List;  
  
**import** javax.inject.Inject;  
  
**import** twitter.classification.api.persist.jdbc.PaginatedSearchTermTweetsDao;  
**import** twitter.classification.api.persist.jdbc.SelectSearchTermClassificationCountDao;  
**import** twitter.classification.api.persist.jdbc.TimeLineForSearchTermDao;  
**import** twitter.classification.api.persist.jdbc.models.ClassificationCountModel;  
**import** twitter.classification.api.persist.jdbc.models.TimeLineForTweetsModel;  
**import** twitter.classification.common.models.ClassificationValueForTweets;  
**import** twitter.classification.common.models.SearchResultsResponse;  
**import** twitter.classification.common.models.TimeLineForTweets;  
**import** twitter.classification.common.persist.DbConnection;  
  
**public class** SearchTermResultService {  
  
 **private** SelectSearchTermClassificationCountDao **selectSearchTermClassificationCountDao**;  
 **private** PaginatedSearchTermTweetsDao **paginatedSearchTermTweetsDao**;  
 **private** TimeLineForSearchTermDao **timeLineForSearchTermDao**;  
  
 @Inject  
 **public** SearchTermResultService(  
 SelectSearchTermClassificationCountDao selectSearchTermClassificationCountDao,  
 PaginatedSearchTermTweetsDao paginatedSearchTermTweetsDao,  
 TimeLineForSearchTermDao timeLineForSearchTermDao  
 ) {  
  
  
 **this**.**selectSearchTermClassificationCountDao** = selectSearchTermClassificationCountDao;  
 **this**.**paginatedSearchTermTweetsDao** = paginatedSearchTermTweetsDao;  
 **this**.**timeLineForSearchTermDao** = timeLineForSearchTermDao;  
 }  
  
 */\*\*  
 \* Will return the results for the search term where it will  
 \* contain the Word Cloud image, Chart etc. as Base64 strings  
 \* which will be rendered in the HTML.  
 \* <p>  
 \* Will also contain data about the count of rumours etc.  
 \*  
 \** ***@return*** *{****@link*** *SearchResultsResponse}  
 \** ***@throws*** *IOException From the encoding of the Base64 String  
 \*/* @DbConnection  
 **public** SearchResultsResponse get(String searchTerm) **throws** IOException {  
  
 SearchResultsResponse searchResultsResponse = **new** SearchResultsResponse();  
  
 List<ClassificationCountModel> classificationCountModelList = **selectSearchTermClassificationCountDao**.select(searchTerm);  
  
 ClassificationCountModel classificationCountModel = classificationCountModelList.get(0);  
  
 searchResultsResponse.setCountOfRumours(classificationCountModel.getCountOfRumours() != **null** ? classificationCountModel.getCountOfRumours().intValue() : **null**);  
 searchResultsResponse.setCountOfNonRumours(classificationCountModel.getCountOfNonRumours() != **null** ? classificationCountModel.getCountOfNonRumours().intValue() : **null**);  
 searchResultsResponse.setTotalCountOfClassifications(classificationCountModel.getTotalClassificationCount() != **null** ? classificationCountModel.getTotalClassificationCount().intValue() : **null**);  
  
  
 **return** searchResultsResponse;  
 }  
  
 */\*\*  
 \* Will return the paginated results for the search term  
 \*  
 \** ***@param searchTerm*** *\** ***@return*** *\*/* @DbConnection  
 **public** List<ClassificationValueForTweets> getPaginatedResults(String searchTerm, **int** offset, **int** limit) {  
  
 **return new** PaginatedResultsService().paginatedResults(**new** ArrayList<>(), **paginatedSearchTermTweetsDao**.get(searchTerm, offset, limit));  
 }  
  
 */\*\*  
 \* Will return the timeline for a particular search term  
 \*  
 \** ***@param searchTerm*** *\** ***@return*** *\*/* @DbConnection  
 **public** TimeLineForTweets getTimeLineForSearchTerm(String searchTerm) {  
  
 TimeLineForTweetsModel timeLineForTweetsModel = **timeLineForSearchTermDao**.get(searchTerm).get(0);  
  
 **return new** TimeLineForTweets()  
 .setNonRumoursLastHour(timeLineForTweetsModel.getCountOfNonRumoursLastHour())  
 .setRumoursLastHour(timeLineForTweetsModel.getCountOfRumoursLastHour())  
 .setNonRumoursOverOneHour(timeLineForTweetsModel.getCountOfNonRumoursOverAnHour())  
 .setRumoursOverOneHour(timeLineForTweetsModel.getCountOfRumoursOverAnHour())  
 .setNonRumoursOverTwoHour(timeLineForTweetsModel.getCountOfNonRumoursOverTwoHours())  
 .setRumoursOverTwoHour(timeLineForTweetsModel.getCountOfRumoursOverTwoHours())  
 .setNonRumoursOverThreeHour(timeLineForTweetsModel.getCountOfNonRumoursOverThreeHours())  
 .setRumoursOverThreeHour(timeLineForTweetsModel.getCountOfRumoursOverThreeHours())  
 .setNonRumoursOverFourHour(timeLineForTweetsModel.getCountOfNonRumoursOverFourHours())  
 .setRumoursOverFourHour(timeLineForTweetsModel.getCountOfRumoursOverFourHours());  
 }  
}

**package** twitter.classification.api.service;  
  
**import** java.util.ArrayList;  
**import** java.util.List;  
  
**import** javax.inject.Inject;  
  
**import** twitter.classification.api.persist.jdbc.SuggestedSearchResultsDao;  
**import** twitter.classification.api.persist.jdbc.models.SuggestedSearchResultsModel;  
**import** twitter.classification.common.models.SuggestedSearchResult;  
**import** twitter.classification.common.models.SuggestedSearchTermsResponse;  
**import** twitter.classification.common.persist.DbConnection;  
  
**public class** SuggestedSearchResultsService {  
  
 **private** SuggestedSearchResultsDao **suggestedSearchResultsDao**;  
  
 @Inject  
 **public** SuggestedSearchResultsService(SuggestedSearchResultsDao suggestedSearchResultsDao) {  
  
 **this**.**suggestedSearchResultsDao** = suggestedSearchResultsDao;  
 }  
  
 */\*\*  
 \* Suggested search terms when there are no results for a user search  
 \*  
 \** ***@return*** *\*/* @DbConnection  
 **public** SuggestedSearchTermsResponse get() {  
  
 SuggestedSearchTermsResponse suggestedSearchResultResponse = **new** SuggestedSearchTermsResponse();  
  
 List<SuggestedSearchResultsModel> suggestedSearchResultsModelList = **suggestedSearchResultsDao**.get();  
  
 **for** (SuggestedSearchResultsModel suggestedSearchResultsModel : suggestedSearchResultsModelList) {  
  
 SuggestedSearchResult suggestedSearchResult = **new** SuggestedSearchResult();  
  
 suggestedSearchResult.setValue(suggestedSearchResultsModel.getValue());  
  
 suggestedSearchResultResponse.addSuggestedSearchResult(suggestedSearchResult);  
 }  
  
 **return** suggestedSearchResultResponse;  
 }  
}

**package** twitter.classification.api.service;  
  
**import** java.io.IOException;  
**import** java.util.List;  
  
**import** javax.inject.Inject;  
  
**import** twitter.classification.api.persist.jdbc.SelectTopHashtagsClassificationCountDao;  
**import** twitter.classification.api.persist.jdbc.models.TopHashtagsClassificationModel;  
**import** twitter.classification.common.models.HashtagResults;  
**import** twitter.classification.common.models.TopHashtagsResponse;  
**import** twitter.classification.common.persist.DbConnection;  
  
**public class** TopHashTagResultService {  
  
 **private** SelectTopHashtagsClassificationCountDao **selectTopHashtagsClassificationCountDao**;  
  
 @Inject  
 **public** TopHashTagResultService(  
 SelectTopHashtagsClassificationCountDao selectTopHashtagsClassificationCountDao  
 ) {  
  
  
 **this**.**selectTopHashtagsClassificationCountDao** = selectTopHashtagsClassificationCountDao;  
 }  
  
 */\*\*  
 \* Will return the top hashtags results where it will  
 \* contain the Word Cloud image, Chart etc. as Base64 strings  
 \* which will be rendered in the HTML.  
 \* <p>  
 \* Will also contain data about the count of rumours etc.  
 \*  
 \** ***@return*** *{****@link*** *TopHashtagsResponse}  
 \** ***@throws*** *IOException From the encoding of the Base64 String  
 \*/* @DbConnection  
 **public** TopHashtagsResponse get() **throws** IOException {  
  
 List<TopHashtagsClassificationModel> topHashtagsClassificationModelList = **selectTopHashtagsClassificationCountDao**.select();  
 TopHashtagsResponse topHashtagsResponse = **new** TopHashtagsResponse();  
  
  
 **for** (TopHashtagsClassificationModel topHashtagsClassificationModel : topHashtagsClassificationModelList) {  
 HashtagResults hashtagResults = **new** HashtagResults();  
  
 hashtagResults.setHashtagValue(topHashtagsClassificationModel.getHashtagValue());  
 hashtagResults.setCountOfNonRumours(topHashtagsClassificationModel.getCountOfNonRumours().intValue());  
 hashtagResults.setCountOfRumours(topHashtagsClassificationModel.getCountOfRumours().intValue());  
 hashtagResults.setTotalCountOfClassifications(topHashtagsClassificationModel.getTotalClassificationCount().intValue());  
  
 topHashtagsResponse.addHashtagResult(hashtagResults);  
 }  
  
 **return** topHashtagsResponse;  
 }  
}

**package** twitter.classification.api.service;  
  
**import** java.io.IOException;  
**import** java.util.List;  
  
**import** javax.inject.Inject;  
  
**import** twitter.classification.api.persist.jdbc.SelectTopUsersClassificationCountDao;  
**import** twitter.classification.api.persist.jdbc.models.TopUsersClassificationModel;  
**import** twitter.classification.common.models.TopUsersResponse;  
**import** twitter.classification.common.models.UserResults;  
**import** twitter.classification.common.persist.DbConnection;  
  
**public class** TopUserResultService {  
  
 **private** SelectTopUsersClassificationCountDao **selectTopUsersClassificationCountDao**;  
  
 @Inject  
 **public** TopUserResultService(  
 SelectTopUsersClassificationCountDao selectTopUsersClassificationCountDao  
 ) {  
  
  
 **this**.**selectTopUsersClassificationCountDao** = selectTopUsersClassificationCountDao;  
 }  
  
 */\*\*  
 \* Will return the top users results where it will  
 \* contain the Word Cloud image, Chart etc. as Base64 strings  
 \* which will be rendered in the HTML.  
 \* <p>  
 \* Will also contain data about the count of rumours etc.  
 \*  
 \** ***@return*** *{****@link*** *TopUsersResponse}  
 \** ***@throws*** *IOException From the encoding of the Base64 String  
 \*/* @DbConnection  
 **public** TopUsersResponse get() **throws** IOException {  
  
 List<TopUsersClassificationModel> topUsersClassificationModelList = **selectTopUsersClassificationCountDao**.select();  
 TopUsersResponse topUsersResponse = **new** TopUsersResponse();  
  
  
 **for** (TopUsersClassificationModel topUsersClassificationModel : topUsersClassificationModelList) {  
 UserResults userResults = **new** UserResults();  
  
 userResults.setUsername(topUsersClassificationModel.getUsername());  
 userResults.setCountOfNonRumours(topUsersClassificationModel.getCountOfNonRumours().intValue());  
 userResults.setCountOfRumours(topUsersClassificationModel.getCountOfRumours().intValue());  
 userResults.setTotalCountOfClassifications(topUsersClassificationModel.getTotalClassificationCount().intValue());  
  
 topUsersResponse.addUserResult(userResults);  
 }  
  
 **return** topUsersResponse;  
 }  
}

**package** twitter.classification.api.service;  
  
**import** java.util.ArrayList;  
**import** java.util.List;  
  
**import** javax.inject.Inject;  
  
**import** twitter.classification.api.persist.jdbc.PaginatedUserTweetsDao;  
**import** twitter.classification.api.persist.jdbc.TimeLineForUsersDao;  
**import** twitter.classification.api.persist.jdbc.models.TimeLineForTweetsModel;  
**import** twitter.classification.common.models.ClassificationValueForTweets;  
**import** twitter.classification.common.models.TimeLineForTweets;  
**import** twitter.classification.common.persist.DbConnection;  
  
**public class** UserResultsService {  
  
 **private** PaginatedUserTweetsDao **paginatedUserTweetsDao**;  
 **private** TimeLineForUsersDao **timeLineForUsersDao**;  
  
 @Inject  
 **public** UserResultsService(  
 PaginatedUserTweetsDao paginatedUserTweetsDao,  
 TimeLineForUsersDao timeLineForUsersDao  
 ) {  
  
 **this**.**paginatedUserTweetsDao** = paginatedUserTweetsDao;  
 **this**.**timeLineForUsersDao** = timeLineForUsersDao;  
 }  
  
 */\*\*  
 \* Paginated table results for a particular user  
 \*  
 \** ***@param username*** *\** ***@param offset*** *\** ***@param limit*** *\** ***@return*** *\*/* @DbConnection  
 **public** List<ClassificationValueForTweets> getPaginatedUserResults(String username, **int** offset, **int** limit) {  
  
 **return new** PaginatedResultsService().paginatedResults(**new** ArrayList<>(), **paginatedUserTweetsDao**.get(username, offset, limit));  
 }  
  
 */\*\*  
 \* Timeline of classification results over 5 hours for a username  
 \*  
 \** ***@param username*** *\** ***@return*** *\*/* @DbConnection  
 **public** TimeLineForTweets getTimeLineForUsername(String username) {  
  
 TimeLineForTweetsModel timeLineForTweetsModel = **timeLineForUsersDao**.get(username).get(0);  
  
 **return new** TimeLineForTweets()  
 .setNonRumoursLastHour(timeLineForTweetsModel.getCountOfNonRumoursLastHour())  
 .setRumoursLastHour(timeLineForTweetsModel.getCountOfRumoursLastHour())  
 .setNonRumoursOverOneHour(timeLineForTweetsModel.getCountOfNonRumoursOverAnHour())  
 .setRumoursOverOneHour(timeLineForTweetsModel.getCountOfRumoursOverAnHour())  
 .setNonRumoursOverTwoHour(timeLineForTweetsModel.getCountOfNonRumoursOverTwoHours())  
 .setRumoursOverTwoHour(timeLineForTweetsModel.getCountOfRumoursOverTwoHours())  
 .setNonRumoursOverThreeHour(timeLineForTweetsModel.getCountOfNonRumoursOverThreeHours())  
 .setRumoursOverThreeHour(timeLineForTweetsModel.getCountOfRumoursOverThreeHours())  
 .setNonRumoursOverFourHour(timeLineForTweetsModel.getCountOfNonRumoursOverFourHours())  
 .setRumoursOverFourHour(timeLineForTweetsModel.getCountOfRumoursOverFourHours());  
 }  
}

## wordclouds

**package** twitter.classification.api.wordclouds;  
  
**import** java.awt.Color;  
**import** java.awt.Dimension;  
**import** java.io.ByteArrayOutputStream;  
**import** java.io.IOException;  
**import** java.nio.charset.StandardCharsets;  
**import** java.util.Base64;  
**import** java.util.List;  
  
**import** javax.imageio.ImageIO;  
  
**import** com.kennycason.kumo.CollisionMode;  
**import** com.kennycason.kumo.WordCloud;  
**import** com.kennycason.kumo.WordFrequency;  
**import** com.kennycason.kumo.bg.CircleBackground;  
**import** com.kennycason.kumo.font.scale.SqrtFontScalar;  
**import** com.kennycason.kumo.nlp.FrequencyAnalyzer;  
**import** com.kennycason.kumo.palette.ColorPalette;  
  
**public class** WordCloudCreationService {  
  
 **private** FrequencyAnalyzer **frequencyAnalyzer**;  
 **private** Dimension **dimension**;  
 **private** WordCloud **wordCloud**;  
  
 **public** WordCloudCreationService() {  
  
 **dimension** = **new** Dimension(200, 200);  
 **wordCloud** = **new** WordCloud(**dimension**, CollisionMode.***PIXEL\_PERFECT***);  
 **wordCloud**.setPadding(1);  
 **wordCloud**.setBackground(**new** CircleBackground(100));  
 **wordCloud**.setColorPalette(**new** ColorPalette(**new** Color(0x86F177), **new** Color(0x69F534), **new** Color(0x40AAF1), **new** Color(0x40C5F1), **new** Color(0x40D3F1), **new** Color(0xFFFFFF)));  
 **wordCloud**.setFontScalar(**new** SqrtFontScalar(10, 40));  
 }  
  
 */\*\*  
 \* Returns a base64 string that can be rendered as an image in the frontend for a list of tweets  
 \*  
 \** ***@param tweets*** *\** ***@return*** *\** ***@throws*** *IOException  
 \*/* **public** String base64String(List<String> tweets) **throws** IOException {  
  
 **frequencyAnalyzer** = **new** FrequencyAnalyzer();  
 List<WordFrequency> wordFrequencies = **frequencyAnalyzer**.load(tweets);  
 **wordCloud**.build(wordFrequencies);  
  
 ByteArrayOutputStream byteArrayOutputStream = **new** ByteArrayOutputStream();  
  
 ImageIO.*write*(**wordCloud**.getBufferedImage(), **"png"**, Base64.*getEncoder*().wrap(byteArrayOutputStream));  
 **return** byteArrayOutputStream.toString(StandardCharsets.***ISO\_8859\_1***.name());  
 }  
}

# classifier/src/main/java/twitter/classification/classifier code listings

## application

**package** twitter.classification.classifier.application;  
  
**import** org.glassfish.jersey.server.ResourceConfig;  
  
**import** twitter.classification.classifier.application.binder.ServicesBinder;  
**import** twitter.classification.common.system.binder.ConfigurationVariableBinder;  
**import** twitter.classification.common.system.helper.FileVariables;  
  
**import static** twitter.classification.common.system.helper.FileVariables.*setLogLevel*;  
  
**public class** WebApplication **extends** ResourceConfig {  
  
 **public** WebApplication() {  
  
 packages(**"twitter.classification.classifier.application"**);  
  
 loadConfigurationValues();  
 *setLogLevel*();  
 register(**new** ConfigurationVariableBinder());  
 register(**new** ServicesBinder());  
 }  
  
 **private void** loadConfigurationValues() {  
  
 **new** FileVariables().setValuesFromConfigurationFile();  
 }  
}

### binder

**package** twitter.classification.classifier.application.binder;  
  
**import** javax.inject.Singleton;  
  
**import** org.glassfish.hk2.utilities.binding.AbstractBinder;  
  
**import** twitter.classification.classifier.application.binder.factory.ClassifierFactory;  
**import** twitter.classification.classifier.application.binder.factory.VerificationClassifierFactory;  
**import** twitter.classification.classifier.helper.ClassificationFromVerificationCheck;  
**import** twitter.classification.classifier.persist.jdbc.InsertHashtagTweetClassificationDao;  
**import** twitter.classification.classifier.persist.jdbc.InsertHashtagsDao;  
**import** twitter.classification.classifier.persist.jdbc.InsertTweetsDao;  
**import** twitter.classification.classifier.persist.jdbc.InsertUserTweetClassificationDao;  
**import** twitter.classification.classifier.persist.jdbc.InsertUsersDao;  
**import** twitter.classification.classifier.service.HandleProcessedTweetService;  
**import** twitter.classification.classifier.service.InsertHashtagEntitiesService;  
**import** twitter.classification.classifier.service.InsertTweetsService;  
**import** twitter.classification.classifier.service.InsertUserTweetClassificationService;  
**import** twitter.classification.classifier.service.InsertUsersService;  
**import** twitter.classification.classifier.service.TrainedClassifier;  
**import** twitter.classification.classifier.service.VerificationClassifier;  
**import** twitter.classification.common.persist.ConnectionManager;  
**import** twitter.classification.common.persist.DbConnectionResolver;  
  
**public class** ServicesBinder **extends** AbstractBinder {  
  
 @Override  
 **protected void** configure() {  
  
 bindFactory(ClassifierFactory.**class**).to(TrainedClassifier.**class**);  
 bindFactory(VerificationClassifierFactory.**class**).to(VerificationClassifier.**class**);  
  
 bind(ConnectionManager.**class**).to(ConnectionManager.**class**).in(Singleton.**class**);  
  
 bind(InsertTweetsDao.**class**).to(InsertTweetsDao.**class**);  
 bind(InsertUsersDao.**class**).to(InsertUsersDao.**class**);  
 bind(InsertUserTweetClassificationDao.**class**).to(InsertUserTweetClassificationDao.**class**);  
 bind(InsertHashtagsDao.**class**).to(InsertHashtagsDao.**class**);  
 bind(InsertHashtagTweetClassificationDao.**class**).to(InsertHashtagTweetClassificationDao.**class**);  
  
 bind(InsertTweetsService.**class**).to(InsertTweetsService.**class**);  
 bind(InsertUsersService.**class**).to(InsertUsersService.**class**);  
 bind(InsertUserTweetClassificationService.**class**).to(InsertUserTweetClassificationService.**class**);  
 bind(InsertHashtagEntitiesService.**class**).to(InsertHashtagEntitiesService.**class**);  
 bind(HandleProcessedTweetService.**class**).to(HandleProcessedTweetService.**class**);  
  
 bind(DbConnectionResolver.**class**).to(DbConnectionResolver.**class**).in(Singleton.**class**);  
  
 bind(ClassificationFromVerificationCheck.**class**).to(ClassificationFromVerificationCheck.**class**);  
 }  
}

#### factory

**package** twitter.classification.classifier.application.binder.factory;  
  
**import** javax.inject.Inject;  
  
**import** twitter.classification.classifier.service.TrainedClassifier;  
**import** twitter.classification.classifier.service.mallet.NaiveBayesClassifier;  
**import** twitter.classification.common.system.binder.factory.BaseFactory;  
  
**public class** ClassifierFactory **implements** BaseFactory<TrainedClassifier> {  
  
 **private final** NaiveBayesClassifier **classifier**;  
  
 @Inject  
 **public** ClassifierFactory() {  
  
 **classifier** = **new** NaiveBayesClassifier();  
 **classifier**.assignClassifierFromDisc();  
 }  
  
 @Override  
 **public** TrainedClassifier provide() {  
  
 **return classifier**;  
 }  
}

**package** twitter.classification.classifier.application.binder.factory;  
  
**import** javax.inject.Inject;  
  
**import** twitter.classification.classifier.service.VerificationClassifier;  
**import** twitter.classification.classifier.service.weka.NaiveBayesClassifier;  
**import** twitter.classification.common.system.binder.factory.BaseFactory;  
  
**public class** VerificationClassifierFactory **implements** BaseFactory<VerificationClassifier> {  
  
 **private final** NaiveBayesClassifier **classifier**;  
  
 @Inject  
 **public** VerificationClassifierFactory() {  
  
 **classifier** = **new** NaiveBayesClassifier();  
 **classifier**.assignClassifierFromDisc();  
 }  
  
 @Override  
 **public** VerificationClassifier provide() {  
  
 **return classifier**;  
 }  
}

## classification

**package** twitter.classification.classifier.classification;  
  
**public class** LabelWeight {  
  
 **private final** String **label**;  
 **private final double weight**;  
  
 **public** LabelWeight(String label, **double** weight) {  
  
  
 **this**.**label** = label;  
 **this**.**weight** = weight;  
 }  
  
 **public** String getLabel() {  
  
 **return label**;  
 }  
  
 **public double** getWeight() {  
  
 **return weight**;  
 }  
}

## helper

**package** twitter.classification.classifier.helper;  
  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**public class** ClassificationCodeFromValue {  
  
 **public static final** Logger ***logger*** = LoggerFactory.*getLogger*(ClassificationCodeFromValue.**class**);  
  
 **private static final** String ***RUMOUR\_CODE*** = **"RMR"**;  
 **private static final** String ***NON\_RUMOUR\_CODE*** = **"NOR"**;  
 **private static final** String ***UNDEFINED\_CODE*** = **"UDF"**;  
  
 **public static** String getClassificationCodeFromValue(String classificationValue) {  
  
 **if** (classificationValue != **null**) {  
 **switch** (classificationValue) {  
  
 **case "non-rumour"**:  
 **return *NON\_RUMOUR\_CODE***;  
 **case "rumour"**:  
 **return *RUMOUR\_CODE***;  
 **default**:  
 ***logger***.info(**"Logging an undefined code for classification value of: {}"**, classificationValue);  
 **return *UNDEFINED\_CODE***;  
 }  
 } **else** {  
 **return *UNDEFINED\_CODE***;  
 }  
 }  
}

**package** twitter.classification.classifier.helper;  
  
**import** javax.inject.Inject;  
  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** twitter.classification.classifier.classification.LabelWeight;  
**import** twitter.classification.common.system.ConfigurationVariable;  
**import** twitter.classification.common.system.helper.ConfigurationVariableParam;  
  
**public class** ClassificationFromVerificationCheck {  
  
 **private static final** Logger ***logger*** = LoggerFactory.*getLogger*(ClassificationFromVerificationCheck.**class**);  
  
 **private double classificationWeightThreshold**;  
  
 @Inject  
 **public** ClassificationFromVerificationCheck(  
 @ConfigurationVariableParam(variable = ConfigurationVariable.***CLASSIFICATION\_WEIGHT\_THRESHOLD***) String classificationWeightThreshold  
 ) {  
  
 **this**.**classificationWeightThreshold** = Double.*valueOf*(classificationWeightThreshold);  
 }  
  
 */\*\*  
 \* As classification is done by both a trained classifier from the Mallet and Weka library the following method  
 \* will consolidate the label based on the two, i.e. if they are the same then it is fine, if they are different  
 \* then if the classification weight from Mallet is lower than the configuration value then it will take  
 \* the label from Wekas' classifier.  
 \*  
 \** ***@param originalClassification*** *\** ***@param verificationClassification*** *\** ***@return*** *{****@link*** *String} the label of the classification  
 \*/* **public** String consolidateClassificationWithVerification(LabelWeight originalClassification, String verificationClassification) {  
  
 ***logger***.debug(  
 **"Original classification is {}, verification classification {}, is same: {}"**,  
 originalClassification.getLabel(),  
 verificationClassification,  
 originalClassification.getLabel().equals(verificationClassification)  
 );  
  
 **if** (originalClassification.getLabel().equals(verificationClassification)) {  
  
 **return** originalClassification.getLabel();  
 } **else** {  
  
 **if** (originalClassification.getWeight() < **classificationWeightThreshold**) {  
  
 ***logger***.debug(  
 **"Storing the verification classification of {}, as original classification weight was {} which is below the threshold of {}"**,  
 verificationClassification,  
 originalClassification.getWeight(),  
 **classificationWeightThreshold** );  
  
 **return** verificationClassification;  
 } **else** {  
  
 **return** originalClassification.getLabel();  
 }  
 }  
 }  
}

## mallet

### classifier

**package** twitter.classification.classifier.mallet.classifier;  
  
**import** java.io.File;  
**import** java.io.FileFilter;  
**import** java.io.FileNotFoundException;  
**import** java.io.FileOutputStream;  
**import** java.io.FileReader;  
**import** java.io.IOException;  
**import** java.io.ObjectOutputStream;  
**import** java.net.URISyntaxException;  
**import** java.nio.file.Paths;  
**import** java.util.ArrayList;  
**import** java.util.Random;  
  
**import** cc.mallet.classify.Classifier;  
**import** cc.mallet.classify.ClassifierTrainer;  
**import** cc.mallet.classify.MaxEntTrainer;  
**import** cc.mallet.classify.NaiveBayesTrainer;  
**import** cc.mallet.pipe.CharSequence2TokenSequence;  
**import** cc.mallet.pipe.FeatureSequence2FeatureVector;  
**import** cc.mallet.pipe.Input2CharSequence;  
**import** cc.mallet.pipe.Pipe;  
**import** cc.mallet.pipe.SerialPipes;  
**import** cc.mallet.pipe.Target2Label;  
**import** cc.mallet.pipe.TokenSequence2FeatureSequence;  
**import** cc.mallet.pipe.TokenSequenceRemoveStopwords;  
**import** cc.mallet.pipe.iterator.CsvIterator;  
**import** cc.mallet.pipe.iterator.FileIterator;  
**import** cc.mallet.types.InstanceList;  
**import** twitter.classification.classifier.mallet.pipes.FeaturePipes;  
  
*/\*\*  
 \* For training a new classifier and serialising to disk using Mallet  
 \*/***public class** TrainClassifier {  
  
 **private static boolean** *isTestingMode* = **false**;  
  
 */\*\*  
 \* Main method to manually train the classifiers  
 \*  
 \** ***@param args*** *\** ***@throws*** *Exception  
 \*/* **public static void** main(String[] args) **throws** Exception {  
  
 TrainClassifier testClassifier = **new** TrainClassifier(**false**);  
  
 testClassifier.trainNaiveBayesClassifier();  
 testClassifier.trainMaxEntClassifier();  
 }  
  
 */\*\*  
 \** ***@param testing*** *param to signify if the class is used for testing mode, as no need to serialise a new object  
 \* to disc if testing the classifier  
 \*/* **public** TrainClassifier(**boolean** testing) {  
  
 *isTestingMode* = testing;  
 }  
  
 */\*\*  
 \* To train a max ent classifier, as both a naive bayes  
 \* and max ent was evaluated in early stages  
 \*  
 \** ***@return*** *{****@link*** *Classifier}  
 \** ***@throws*** *IOException  
 \** ***@throws*** *URISyntaxException  
 \*/* **public** Classifier trainMaxEntClassifier() **throws** IOException, URISyntaxException {  
  
 FileReader fileReader = getFileReader();  
  
 ArrayList<Pipe> pipes = **new** ArrayList<>();  
  
 pipes.add(**new** Target2Label());  
 pipes.add(**new** CharSequence2TokenSequence());  
 pipes.add(**new** TokenSequence2FeatureSequence());  
 pipes.add(**new** FeatureSequence2FeatureVector());  
 SerialPipes pipe = **new** SerialPipes(pipes);  
  
 InstanceList trainingInstanceList = **new** InstanceList(pipe);  
  
 *// file is format of non-rumour|rumour, data* trainingInstanceList.addThruPipe(**new** CsvIterator(fileReader, **"(non-rumour|rumour), (.\*)"**, 2, 1, -1));  
  
 ClassifierTrainer trainer = **new** MaxEntTrainer();  
 Classifier classifier = trainer.train(trainingInstanceList);  
  
 **if** (*isTestingMode*) {  
 **return** classifier;  
 }  
  
 File classifierFile = **new** File(Paths.*get*(**"classifier/src/main/webapp/WEB-INF/classes/trained-classifier/max-ent-classifier.txt"**).toUri());  
  
 *// safety to only have one file* **if** (!classifierFile.exists()) {  
 classifierFile.createNewFile();  
 } **else** {  
 classifierFile.delete();  
 classifierFile.createNewFile();  
 }  
  
 ObjectOutputStream objectOutputStream = **new** ObjectOutputStream(**new** FileOutputStream(classifierFile));  
  
 objectOutputStream.writeObject(classifier);  
 objectOutputStream.close();  
  
 **return** classifier;  
 }  
  
 */\*\*  
 \* To train a naive bayes classifier, as both a naive bayes  
 \* and max ent was evaluated in early stages  
 \*  
 \** ***@return*** *{****@link*** *Classifier}  
 \** ***@throws*** *IOException  
 \** ***@throws*** *URISyntaxException  
 \*/* **public** Classifier trainNaiveBayesClassifier() **throws** IOException, URISyntaxException {  
  
 FileReader fileReader = getFileReader();  
  
 SerialPipes pipe = **new** FeaturePipes(*isTestingMode*).getFeaturePipes();  
  
 InstanceList trainingInstanceList = **new** InstanceList(pipe);  
  
 *// file is format of non-rumour|rumour, data* trainingInstanceList.addThruPipe(**new** CsvIterator(fileReader, **"(non-rumour|rumour), (.\*)"**, 2, 1, -1));  
  
 trainingInstanceList.shuffle(**new** Random(**new** ThreadLocal().hashCode()));  
  
 ClassifierTrainer trainer = **new** NaiveBayesTrainer();  
 Classifier classifier = trainer.train(trainingInstanceList);  
  
 **if** (*isTestingMode*) {  
 **return** classifier;  
 }  
  
 File classifierFile = **new** File(Paths.*get*(**"classifier/src/main/webapp/WEB-INF/classes/trained-classifier/classifier.txt"**).toUri());  
  
 *// safety to only have one file* **if** (!classifierFile.exists()) {  
 classifierFile.createNewFile();  
 } **else** {  
 classifierFile.delete();  
 classifierFile.createNewFile();  
 }  
  
 ObjectOutputStream objectOutputStream = **new** ObjectOutputStream(**new** FileOutputStream(classifierFile));  
  
 objectOutputStream.writeObject(classifier);  
 objectOutputStream.close();  
  
 **return** classifier;  
 }  
  
 **private** FileReader getFileReader() **throws** FileNotFoundException {  
  
 **return new** FileReader(getClass().getClassLoader().getResource(**"datasets/rumours-non-rumours-dataset.csv"**).getFile());  
 }  
}

### pipes

**package** twitter.classification.classifier.mallet.pipes;  
  
**import** java.nio.file.Paths;  
**import** java.util.ArrayList;  
  
**import** cc.mallet.pipe.CharSequence2TokenSequence;  
**import** cc.mallet.pipe.FeatureSequence2FeatureVector;  
**import** cc.mallet.pipe.Pipe;  
**import** cc.mallet.pipe.SerialPipes;  
**import** cc.mallet.pipe.Target2Label;  
**import** cc.mallet.pipe.TokenSequence2FeatureSequence;  
**import** cc.mallet.pipe.TokenSequenceRemoveStopwords;  
  
**public class** FeaturePipes {  
  
 **private** String **stopWordsPath** = **"classifier/src/main/resources/stopwords/stopwords.txt"**;  
  
 **public** FeaturePipes(Boolean isTestingMode) {  
  
 **if** (isTestingMode) {  
  
 **stopWordsPath** = **"src/main/resources/stopwords/stopwords.txt"**;  
 }  
 }  
  
 **public** SerialPipes getFeaturePipes() {  
  
 ArrayList<Pipe> pipes = **new** ArrayList<>();  
  
 pipes.add(**new** Target2Label());  
 pipes.add(**new** CharSequence2TokenSequence());  
 pipes.add(**new** TokenSequenceRemoveStopwords().addStopWords(Paths.*get*(**stopWordsPath**).toFile()).setCaseSensitive(**false**));  
 pipes.add(**new** TokenSequence2FeatureSequence());  
 pipes.add(**new** FeatureSequence2FeatureVector());  
  
 **return new** SerialPipes(pipes);  
 }  
}

## persist.jdbc

**package** twitter.classification.classifier.persist.jdbc;  
  
**import** javax.inject.Inject;  
  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** twitter.classification.classifier.persist.jdbc.queries.InsertHashtagsDbQuery;  
**import** twitter.classification.common.persist.ConnectionManager;  
**import** twitter.classification.common.persist.jdbc.utils.DbQueryRunner;  
  
**public class** InsertHashtagsDao {  
  
 **public static final** Logger ***logger*** = LoggerFactory.*getLogger*(InsertHashtagsDao.**class**);  
  
 **private** ConnectionManager **connectionManager**;  
  
 @Inject  
 **public** InsertHashtagsDao(ConnectionManager connectionManager) {  
  
 **this**.**connectionManager** = connectionManager;  
 }  
  
 **public void** insert(String hashtagValue) {  
  
 DbQueryRunner runner = **new** DbQueryRunner(**connectionManager**.getConnection());  
 runner.executeUpdate(**new** InsertHashtagsDbQuery().buildQuery(), hashtagValue);  
 }  
}

**package** twitter.classification.classifier.persist.jdbc;  
  
**import** javax.inject.Inject;  
  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** twitter.classification.classifier.persist.jdbc.queries.InsertHashtagTweetClassificationDbQuery;  
**import** twitter.classification.common.persist.ConnectionManager;  
**import** twitter.classification.common.persist.jdbc.utils.DbQueryRunner;  
  
**public class** InsertHashtagTweetClassificationDao {  
  
 **public static final** Logger ***logger*** = LoggerFactory.*getLogger*(InsertHashtagTweetClassificationDao.**class**);  
  
 **private** ConnectionManager **connectionManager**;  
  
 @Inject  
 **public** InsertHashtagTweetClassificationDao(ConnectionManager connectionManager) {  
  
 **this**.**connectionManager** = connectionManager;  
 }  
  
 **public void** insert(String hashtagValue, Long twitterTweetId) {  
  
 DbQueryRunner runner = **new** DbQueryRunner(**connectionManager**.getConnection());  
 runner.executeUpdate(**new** InsertHashtagTweetClassificationDbQuery().buildQuery(), hashtagValue, twitterTweetId);  
 }  
}

**package** twitter.classification.classifier.persist.jdbc;  
  
**import** javax.inject.Inject;  
  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** twitter.classification.classifier.persist.jdbc.queries.InsertTweetsDbQuery;  
**import** twitter.classification.common.persist.ConnectionManager;  
**import** twitter.classification.common.persist.jdbc.utils.DbQueryRunner;  
  
**public class** InsertTweetsDao {  
  
 **public static final** Logger ***logger*** = LoggerFactory.*getLogger*(InsertTweetsDao.**class**);  
  
 **private** ConnectionManager **connectionManager**;  
  
 @Inject  
 **public** InsertTweetsDao(ConnectionManager connectionManager) {  
  
 **this**.**connectionManager** = connectionManager;  
 }  
  
 **public void** insert(Long tweetId, String originalTweetText, String processedTweetText, String classificationCode) {  
  
 DbQueryRunner runner = **new** DbQueryRunner(**connectionManager**.getConnection());  
 runner.executeUpdate(**new** InsertTweetsDbQuery().buildQuery(), tweetId, originalTweetText, processedTweetText, classificationCode);  
 }  
}

**package** twitter.classification.classifier.persist.jdbc;  
  
**import** javax.inject.Inject;  
  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** twitter.classification.classifier.persist.jdbc.queries.InsertUsersDbQuery;  
**import** twitter.classification.common.persist.ConnectionManager;  
**import** twitter.classification.common.persist.jdbc.utils.DbQueryRunner;  
  
**public class** InsertUsersDao {  
  
 **public static final** Logger logger = LoggerFactory.getLogger(InsertTweetsDao.**class**);  
  
 **private** ConnectionManager connectionManager;  
  
 @Inject  
 **public** InsertUsersDao(ConnectionManager connectionManager) {  
  
 **this**.connectionManager = connectionManager;  
 }  
  
 **public void** insert(String username, Long twitterUsernameId) {  
  
 DbQueryRunner runner = **new** DbQueryRunner(connectionManager.getConnection());  
 runner.executeUpdate(**new** InsertUsersDbQuery().buildQuery(), username, twitterUsernameId);  
 }  
}

**package** twitter.classification.classifier.persist.jdbc;  
  
**import** javax.inject.Inject;  
  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** twitter.classification.classifier.persist.jdbc.queries.InsertUserTweetClassificationDbQuery;  
**import** twitter.classification.common.persist.ConnectionManager;  
**import** twitter.classification.common.persist.jdbc.utils.DbQueryRunner;  
  
**public class** InsertUserTweetClassificationDao {  
  
 **public static final** Logger ***logger*** = LoggerFactory.*getLogger*(InsertUserTweetClassificationDao.**class**);  
  
 **private** ConnectionManager **connectionManager**;  
  
 @Inject  
 **public** InsertUserTweetClassificationDao(ConnectionManager connectionManager) {  
  
 **this**.**connectionManager** = connectionManager;  
 }  
  
 **public void** insert(Long twitterUserId, Long twitterTweetId) {  
  
 DbQueryRunner runner = **new** DbQueryRunner(**connectionManager**.getConnection());  
 runner.executeUpdate(**new** InsertUserTweetClassificationDbQuery().buildQuery(), twitterUserId, twitterTweetId);  
 }  
}

### queries

**package** twitter.classification.classifier.persist.jdbc.queries;  
  
**import** twitter.classification.common.persist.jdbc.queries.DbQuery;  
  
**public class** InsertHashtagsDbQuery **implements** DbQuery {  
  
 @Override  
 **public** String buildQuery() {  
  
 **return "INSERT IGNORE INTO "** +  
 **"hashtags (hashtag\_value) "** +  
 **"VALUES (?);"**;  
 }  
}

**package** twitter.classification.classifier.persist.jdbc.queries;  
  
**import** twitter.classification.common.persist.jdbc.queries.DbQuery;  
  
**public class** InsertHashtagTweetClassificationDbQuery **implements** DbQuery {  
  
 @Override  
 **public** String buildQuery() {  
  
 **return "INSERT IGNORE INTO hashtag\_tweet\_classifications "** +  
 **"(hashtag\_id, tweet\_id) "** +  
 **"SELECT hashtags.id, tweets.id "** +  
 **"FROM hashtags, tweets "** +  
 **"WHERE hashtags.hashtag\_value = ? "** +  
 **"AND tweets.tweet\_id = ?;"**;  
 }  
}

**package** twitter.classification.classifier.persist.jdbc.queries;  
  
**import** twitter.classification.common.persist.jdbc.queries.DbQuery;  
  
**public class** InsertTweetsDbQuery **implements** DbQuery {  
  
 @Override  
 **public** String buildQuery() {  
  
 **return "INSERT IGNORE INTO tweets "** +  
 **"(tweet\_id, original\_tweet\_text, processed\_tweet\_text, classification\_id) "** +  
 **"SELECT ?, ?, ?, classification\_types.id "** +  
 **"FROM classification\_types "** +  
 **"WHERE classification\_types.classification\_code = ?;"**;  
 }  
}

**package** twitter.classification.classifier.persist.jdbc.queries;  
  
**import** twitter.classification.common.persist.jdbc.queries.DbQuery;  
  
**public class** InsertUsersDbQuery **implements** DbQuery {  
  
 @Override  
 **public** String buildQuery() {  
  
 **return "INSERT IGNORE INTO "** +  
 **"users (username, twitter\_id) "** +  
 **"VALUES (?, ?);"**;  
 }  
}

**package** twitter.classification.classifier.persist.jdbc.queries;  
  
**import** twitter.classification.common.persist.jdbc.queries.DbQuery;  
  
**public class** InsertUserTweetClassificationDbQuery **implements** DbQuery {  
  
 @Override  
 **public** String buildQuery() {  
  
 **return "INSERT IGNORE INTO users\_tweet\_classifications "** +  
 **"(user\_id, tweet\_id) "** +  
 **"SELECT users.id, tweets.id "** +  
 **"FROM users, tweets "** +  
 **"WHERE users.twitter\_id = ? "** +  
 **"AND tweets.tweet\_id = ?;"**;  
 }  
}

## resource

**package** twitter.classification.classifier.resource;  
  
**import** javax.inject.Inject;  
**import** javax.inject.Singleton;  
**import** javax.ws.rs.Consumes;  
**import** javax.ws.rs.GET;  
**import** javax.ws.rs.POST;  
**import** javax.ws.rs.Path;  
**import** javax.ws.rs.Produces;  
**import** javax.ws.rs.core.MediaType;  
  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** com.fasterxml.jackson.core.JsonProcessingException;  
**import** com.fasterxml.jackson.databind.ObjectMapper;  
**import** twitter.classification.classifier.classification.LabelWeight;  
**import** twitter.classification.classifier.helper.ClassificationFromVerificationCheck;  
**import** twitter.classification.classifier.service.HandleProcessedTweetService;  
**import** twitter.classification.classifier.service.TrainedClassifier;  
**import** twitter.classification.classifier.service.VerificationClassifier;  
**import** twitter.classification.common.models.ClassifierStatusResponse;  
**import** twitter.classification.common.tweetdetails.model.ClassificationModel;  
**import** twitter.classification.common.tweetdetails.model.PreProcessedItem;  
**import** twitter.classification.common.tweetdetails.model.ProcessedTweetModel;  
  
@Singleton  
@Path(**"/classify"**)  
**public class** ClassificationResource {  
  
 **private static final** Logger ***logger*** = LoggerFactory.*getLogger*(ClassificationResource.**class**);  
  
 **private** TrainedClassifier **classifier**;  
 **private** VerificationClassifier **verificationClassifier**;  
 **private** HandleProcessedTweetService **handleProcessedTweetService**;  
 **private** ClassificationFromVerificationCheck **classificationFromVerificationCheck**;  
  
 @Inject  
 **public** ClassificationResource(  
 TrainedClassifier classifier,  
 VerificationClassifier verificationClassifier,  
 HandleProcessedTweetService handleProcessedTweetService,  
 ClassificationFromVerificationCheck classificationFromVerificationCheck  
 ) {  
  
 **this**.**classifier** = classifier;  
 **this**.**verificationClassifier** = verificationClassifier;  
 **this**.**handleProcessedTweetService** = handleProcessedTweetService;  
 **this**.**classificationFromVerificationCheck** = classificationFromVerificationCheck;  
 }  
  
 */\*\*  
 \* Post method to classify the processed item from the preprocessor and a label will be assigned  
 \** ***@param preProcessedItem*** *\** ***@return*** *\*/* @POST  
 @Produces(MediaType.***APPLICATION\_JSON***)  
 @Consumes(MediaType.***APPLICATION\_JSON***)  
 **public** ClassificationModel getClassificationForTweet(PreProcessedItem preProcessedItem) {  
  
 **try** {  
  
 ***logger***.debug(**"PreprocessedItem is {}"**, **new** ObjectMapper().writeValueAsString(preProcessedItem));  
  
 ProcessedTweetModel processedTweetModel = **new** ProcessedTweetModel(preProcessedItem);  
  
 LabelWeight originalClassification = **classifier**.classifyTweet(preProcessedItem.getProcessedTweetBody());  
 String verificationClassification = **verificationClassifier**.classifyTweet(preProcessedItem.getProcessedTweetBody());  
  
 processedTweetModel.setClassificationValue(**classificationFromVerificationCheck**.consolidateClassificationWithVerification(originalClassification, verificationClassification));  
  
 **handleProcessedTweetService**.handle(processedTweetModel);  
  
 } **catch** (JsonProcessingException e) {  
  
 e.printStackTrace();  
 }  
  
 **return new** ClassificationModel().setClassificationLabel(**"return"**);  
 }  
  
 @GET  
 @Produces(MediaType.***APPLICATION\_JSON***)  
 @Path(**"/status"**)  
 **public** ClassifierStatusResponse getClassifiersStatus() {  
  
 **return new** ClassifierStatusResponse().setRunning(**true**);  
 }  
}

## service

**package** twitter.classification.classifier.service;  
  
**import** javax.inject.Inject;  
  
**import** twitter.classification.classifier.helper.ClassificationCodeFromValue;  
**import** twitter.classification.common.persist.DbConnection;  
**import** twitter.classification.common.tweetdetails.model.ProcessedTweetModel;  
  
**public class** HandleProcessedTweetService {  
  
 **private** InsertTweetsService **insertTweetsService**;  
 **private** InsertUsersService **insertUsersService**;  
 **private** InsertUserTweetClassificationService **insertUserTweetClassificationService**;  
 **private** InsertHashtagEntitiesService **insertHashtagEntitiesService**;  
  
 @Inject  
 **public** HandleProcessedTweetService(  
 InsertTweetsService insertTweetsService,  
 InsertUsersService insertUsersService,  
 InsertUserTweetClassificationService insertUserTweetClassificationService,  
 InsertHashtagEntitiesService insertHashtagEntitiesService  
 ) {  
  
 **this**.**insertTweetsService** = insertTweetsService;  
 **this**.**insertUsersService** = insertUsersService;  
 **this**.**insertUserTweetClassificationService** = insertUserTweetClassificationService;  
 **this**.**insertHashtagEntitiesService** = insertHashtagEntitiesService;  
 }  
  
 @DbConnection  
 **public void** handle(ProcessedTweetModel processedTweetModel) {  
  
 **insertTweetsService**.insert(  
 processedTweetModel.getTweetId(),  
 processedTweetModel.getOriginalTweetBody(),  
 processedTweetModel.getProcessedTweetBody(),  
 ClassificationCodeFromValue.*getClassificationCodeFromValue*(processedTweetModel.getClassificationValue())  
 );  
  
 **insertUsersService**.insert(processedTweetModel.getUsername(), processedTweetModel.getUserId());  
  
 **insertUserTweetClassificationService**.insert(processedTweetModel.getUserId(), processedTweetModel.getTweetId());  
  
 **for** (String hashtagValue : processedTweetModel.getHashtags()) {  
  
 **insertHashtagEntitiesService**.insert(hashtagValue, processedTweetModel.getTweetId());  
 }  
 }  
}

**package** twitter.classification.classifier.service;  
  
**import** javax.inject.Inject;  
  
**import** twitter.classification.classifier.persist.jdbc.InsertHashtagTweetClassificationDao;  
**import** twitter.classification.classifier.persist.jdbc.InsertHashtagsDao;  
  
**public class** InsertHashtagEntitiesService {  
  
 **private** InsertHashtagsDao **insertHashtagsDao**;  
 **private** InsertHashtagTweetClassificationDao **insertHashtagTweetClassificationDao**;  
  
 @Inject  
 **public** InsertHashtagEntitiesService(  
 InsertHashtagsDao insertHashtagsDao,  
 InsertHashtagTweetClassificationDao insertHashtagTweetClassificationDao  
 ) {  
  
  
 **this**.**insertHashtagsDao** = insertHashtagsDao;  
 **this**.**insertHashtagTweetClassificationDao** = insertHashtagTweetClassificationDao;  
 }  
  
 **public void** insert(String hashtagValue, Long twitterTweetId) {  
  
 **insertHashtagsDao**.insert(hashtagValue);  
 **insertHashtagTweetClassificationDao**.insert(hashtagValue, twitterTweetId);  
 }  
}

**package** twitter.classification.classifier.service;  
  
**import** javax.inject.Inject;  
  
**import** twitter.classification.classifier.persist.jdbc.InsertTweetsDao;  
  
**public class** InsertTweetsService {  
  
 **private final** InsertTweetsDao **insertTweetsDao**;  
  
 @Inject  
 **public** InsertTweetsService(InsertTweetsDao insertTweetsDao) {  
  
 **this**.**insertTweetsDao** = insertTweetsDao;  
 }  
  
 **public void** insert(Long tweetId, String originalTweetText, String processedTweetText, String classificationCode) {  
  
 **insertTweetsDao**.insert(tweetId, originalTweetText, processedTweetText, classificationCode);  
 }  
}

**package** twitter.classification.classifier.service;  
  
**import** javax.inject.Inject;  
  
**import** twitter.classification.classifier.persist.jdbc.InsertUsersDao;  
  
**public class** InsertUsersService {  
  
 **private final** InsertUsersDao **insertUsersDao**;  
  
 @Inject  
 **public** InsertUsersService(InsertUsersDao insertUsersDao) {  
  
 **this**.**insertUsersDao** = insertUsersDao;  
 }  
  
 **public void** insert(String username, Long twitterUsernameId) {  
  
 **insertUsersDao**.insert(username, twitterUsernameId);  
 }  
}

**package** twitter.classification.classifier.service;  
  
**import** javax.inject.Inject;  
  
**import** twitter.classification.classifier.persist.jdbc.InsertUserTweetClassificationDao;  
  
**public class** InsertUserTweetClassificationService {  
  
 **private final** InsertUserTweetClassificationDao **insertUserTweetClassificationDao**;  
  
 @Inject  
 **public** InsertUserTweetClassificationService(InsertUserTweetClassificationDao insertUserTweetClassificationDao) {  
  
 **this**.**insertUserTweetClassificationDao** = insertUserTweetClassificationDao;  
 }  
  
 **public void** insert(Long twitterUserId, Long twitterTweetId) {  
  
 **insertUserTweetClassificationDao**.insert(twitterUserId, twitterTweetId);  
 }  
}

**package** twitter.classification.classifier.service;  
  
**import** twitter.classification.classifier.classification.LabelWeight;  
  
**public interface** TrainedClassifier {  
  
 Object assignClassifierFromDisc();  
  
 LabelWeight classifyTweet(String tweet);  
}

**package** twitter.classification.classifier.service;  
  
**import** twitter.classification.classifier.classification.LabelWeight;  
  
**public interface** VerificationClassifier {  
  
 Object assignClassifierFromDisc();  
  
 String classifyTweet(String tweet);  
}

### mallet

**package** twitter.classification.classifier.service.mallet;  
  
**import** java.io.FileInputStream;  
**import** java.io.IOException;  
**import** java.io.ObjectInputStream;  
  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** cc.mallet.classify.Classification;  
**import** cc.mallet.classify.Classifier;  
**import** cc.mallet.types.Label;  
**import** twitter.classification.classifier.classification.LabelWeight;  
**import** twitter.classification.classifier.service.TrainedClassifier;  
  
*/\*\*  
 \* A NaiveBayes classifier using the Mallet library  
 \*/***public class** NaiveBayesClassifier **implements** TrainedClassifier {  
  
 **private static final** Logger ***logger*** = LoggerFactory.*getLogger*(NaiveBayesClassifier.**class**);  
  
 **private** Classifier **classifier**;  
  
 **public** NaiveBayesClassifier() {  
 }  
  
 */\*\*  
 \* Assumes that a classifier has been trained and  
 \* serialised to disk and stored in WEB-INF/classes/trained-classifier/classifier.txt  
 \* <p>  
 \* Can use TrainClassifier to train and serialise a classifier to disk  
 \*  
 \** ***@return*** *NaiveBayesClassifier  
 \*/* @Override  
 **public** Classifier assignClassifierFromDisc() {  
  
 ObjectInputStream objectInputStream = **null**;  
 **try** {  
 objectInputStream = **new** ObjectInputStream(**new** FileInputStream(getClass().getClassLoader().getResource(**"trained-classifier/classifier.txt"**).getFile()));  
 } **catch** (IOException e) {  
 ***logger***.error(**"Issue getting serialised object from disk"**);  
 }  
  
 **try** {  
 **assert** objectInputStream != **null**;  
 **classifier** = (Classifier) objectInputStream.readObject();  
 } **catch** (IOException | ClassNotFoundException e) {  
 ***logger***.error(**"Issue reading the serialised object"**);  
 }  
  
 **try** {  
 objectInputStream.close();  
 } **catch** (IOException e) {  
 ***logger***.error(**"Issue closing the object"**);  
 }  
  
 **return classifier**;  
 }  
  
 */\*\*  
 \* Uses the trained classifier to return the  
 \* label for the passed tweet body  
 \*  
 \** ***@param tweet*** *{****@link*** *String}  
 \** ***@return*** *{****@link*** *String} Label  
 \*/* @Override  
 **public** LabelWeight classifyTweet(String tweet) {  
  
 Classification classification = **classifier**.classify(tweet);  
  
 Label label = classification.getLabeling().getBestLabel();  
  
 **return new** LabelWeight(label.toString(), classification.getLabeling().value(label));  
 }  
}

### weka

**package** twitter.classification.classifier.service.weka;  
  
**import** java.io.File;  
**import** java.io.IOException;  
  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** twitter.classification.classifier.classification.LabelWeight;  
**import** twitter.classification.classifier.service.TrainedClassifier;  
**import** twitter.classification.classifier.service.VerificationClassifier;  
**import** twitter.classification.classifier.weka.converter.WekaInstanceFromString;  
**import** weka.classifiers.Evaluation;  
**import** weka.classifiers.meta.FilteredClassifier;  
**import** weka.core.Instances;  
**import** weka.core.SerializationHelper;  
**import** weka.core.converters.TextDirectoryLoader;  
  
*/\*\*  
 \* A NaiveBayes Classifier using the Weka library  
 \*/***public class** NaiveBayesClassifier **implements** VerificationClassifier {  
  
 **private static final** String ***DATASET\_FILE\_LOCATION*** = **"dataset-weka/"**;  
  
 **private static final** Logger ***logger*** = LoggerFactory.*getLogger*(NaiveBayesClassifier.**class**);  
  
 **private** FilteredClassifier classifier;  
 **private** Instances dataset;  
  
 **public** NaiveBayesClassifier() {  
 }  
  
 */\*\*  
 \* Assumes that a classifier has been trained and  
 \* serialised to disk and stored in WEB-INF/classes/trained-classifier/weka-nb-classifier.model  
 \* <p>  
 \* Can use {****@link*** *twitter.classification.classifier.weka.classifier.NaiveBayesClassifier} to train and serialise a classifier to disk  
 \*  
 \** ***@return*** *NaiveBayesClassifier  
 \*/* @Override  
 **public** FilteredClassifier assignClassifierFromDisc() {  
  
 **try** {  
  
 classifier = (FilteredClassifier) SerializationHelper.read(getClass().getClassLoader().getResource(**"trained-classifier/weka-nb-classifier.model"**).getFile());  
 dataset = getDatasetFromFileDirectory();  
 **return** classifier;  
 } **catch** (Exception e) {  
  
 logger.error(**"Issue reading classifier from disc"**);  
 }  
  
 **return null**;  
 }  
  
 */\*\*  
 \* Uses the trained classifier to return the  
 \* label for the passed tweet body  
 \*  
 \** ***@param*** *tweet  
 \** ***@return*** *{****@link*** *String} the classified label  
 \*/* @Override  
 **public** String classifyTweet(String tweet) {  
  
 **try** {  
  
 **return** dataset.classAttribute().value((**int**) classifier.classifyInstance(**new** WekaInstanceFromString().getInstanceFromString(tweet, dataset)));  
 } **catch** (Exception e) {  
  
 e.printStackTrace();  
 }  
  
 **return null**;  
 }  
  
 */\*\*  
 \* Weka requires reuse of the original dataset used for training to retrieve the class information  
 \*  
 \** ***@return*** *{****@link*** *Instances}  
 \** ***@throws*** *IOException  
 \*/* **private** Instances getDatasetFromFileDirectory() **throws** IOException {  
  
 TextDirectoryLoader textDirectoryLoader = **new** TextDirectoryLoader();  
 textDirectoryLoader.setDirectory(**new** File(getClass().getClassLoader().getResource(DATASET\_FILE\_LOCATION).getFile()));  
  
 Instances dataset = textDirectoryLoader.getDataSet();  
 dataset.setClassIndex(dataset.numAttributes() - 1);  
  
 **return** dataset;  
 }  
}

## weka

### classifier

**package** twitter.classification.classifier.weka.classifier;  
  
**import** java.io.File;  
**import** java.io.IOException;  
**import** java.util.Random;  
  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** twitter.classification.classifier.weka.dataset.DatasetLoader;  
**import** twitter.classification.classifier.weka.filter.StringToWordVectorFilter;  
**import** weka.classifiers.bayes.NaiveBayes;  
**import** weka.classifiers.bayes.NaiveBayesMultinomial;  
**import** weka.classifiers.meta.FilteredClassifier;  
**import** weka.core.Instances;  
**import** weka.core.SerializationHelper;  
  
*/\*\*  
 \* For evaluation of Weka classifier, as both had to be evaluated.  
 \*/***public class** NaiveBayesClassifier {  
  
 **private static final** Logger ***logger*** = LoggerFactory.*getLogger*(NaiveBayesClassifier.**class**);  
 **private static final** String ***TRAINED\_CLASSIFIER\_LOCATION*** = **"classifier/src/main/webapp/WEB-INF/classes/trained-classifier/weka-nb-classifier.model"**;  
  
 */\*\*  
 \* Run to train a new {****@link*** *weka.classifiers.meta.FilteredClassifier} and serialise to disc  
 \*  
 \** ***@param args*** *\*/* **public static void** main(String[] args) {  
  
 **try** {  
  
 Instances dataset = **new** DatasetLoader().getInstancesFromFileDirectory();  
  
 NaiveBayesMultinomial naiveBayes = **new** NaiveBayesMultinomial();  
 FilteredClassifier filteredClassifier = **new** FilteredClassifier();  
 filteredClassifier.setFilter(StringToWordVectorFilter.*getStringToWordVector*());  
 filteredClassifier.setClassifier(naiveBayes);  
  
 dataset.randomize(**new** Random(**new** ThreadLocal().hashCode()));  
  
 filteredClassifier.buildClassifier(dataset);  
  
 File classifierFile = **new** File(***TRAINED\_CLASSIFIER\_LOCATION***);  
  
 *// safety to only create the file once* **if** (!classifierFile.exists()) {  
 System.***out***.println(**"Creating a new file"**);  
 classifierFile.createNewFile();  
 } **else** {  
 System.***out***.println(**"Deleting existing file and creating new file"**);  
 classifierFile.delete();  
 classifierFile.createNewFile();  
 }  
  
 SerializationHelper.*write*(***TRAINED\_CLASSIFIER\_LOCATION***, filteredClassifier);  
  
 } **catch** (IOException exception) {  
  
 ***logger***.error(**"Issue getting instances from file directory"**);  
 } **catch** (Exception exception) {  
  
 ***logger***.error(**"Issue serialising new classifier"**);  
 }  
 }  
}

### converter

**package** twitter.classification.classifier.weka.converter;  
  
**import** weka.core.DenseInstance;  
**import** weka.core.Instance;  
**import** weka.core.Instances;  
  
*/\*\*  
 \* Weka requires a {****@link*** *Instance} object for classification, so converting the string  
 \* to a {****@link*** *Instance} object for classifying the tweet text  
 \*/***public class** WekaInstanceFromString {  
  
 **public** Instance getInstanceFromString(String tweetText, Instances dataset) {  
  
 Instance instance = **new** DenseInstance(2);  
 instance.setDataset(dataset);  
 instance.setValue(dataset.attribute(**"text"**), dataset.attribute(**"text"**).addStringValue(tweetText));  
 instance.setClassMissing();  
  
 **return** instance;  
 }  
}

### dataset

**package** twitter.classification.classifier.weka.dataset;  
  
**import** java.io.IOException;  
**import** java.nio.file.Paths;  
  
**import** weka.core.Instances;  
**import** weka.core.converters.TextDirectoryLoader;  
  
**public class** DatasetLoader {  
  
 **private static final** String ***DATASET\_FILE\_LOCATION*** = **"classifier/src/main/resources/dataset-weka/"**;  
  
 **public** Instances getInstancesFromFileDirectory() **throws** IOException {  
  
 TextDirectoryLoader textDirectoryLoader = **new** TextDirectoryLoader();  
 textDirectoryLoader.setDirectory(Paths.*get*(***DATASET\_FILE\_LOCATION***).toFile());  
  
 Instances dataset = textDirectoryLoader.getDataSet();  
 dataset.setClassIndex(dataset.numAttributes() - 1);  
  
 **return** dataset;  
 }  
}

### filter

**package** twitter.classification.classifier.weka.filter;  
  
**import** twitter.classification.classifier.weka.stopwords.StopWordsHandler;  
**import** weka.filters.unsupervised.attribute.StringToWordVector;  
  
**public class** StringToWordVectorFilter {  
  
 **public static** StringToWordVector getStringToWordVector() {  
  
 StringToWordVector stringToWordVector = **new** StringToWordVector();  
 stringToWordVector.setLowerCaseTokens(**true**);  
 stringToWordVector.setWordsToKeep(5000);  
 stringToWordVector.setStopwordsHandler(StopWordsHandler.*getStopWordsHandler*());  
  
 **return** stringToWordVector;  
 }  
}

### stopwords

**package** twitter.classification.classifier.weka.stopwords;  
  
**import** java.nio.file.Paths;  
  
**import** weka.core.stopwords.StopwordsHandler;  
**import** weka.core.stopwords.WordsFromFile;  
  
**public class** StopWordsHandler {  
  
 **private static final** String ***STOPWORDS\_LOCATION*** = **"classifier/src/main/resources/stopwords/stopwords.txt"**;  
  
 */\*\*  
 \* Stop words gathered from the following location:  
 \*  
 \* https://gist.githubusercontent.com/sebleier/554280/raw/7e0e4a1ce04c2bb7bd41089c9821dbcf6d0c786c/NLTK's%2520list%2520of%2520english%2520stopwords  
 \** ***@return*** *{****@link*** *StopwordsHandler}  
 \*/* **public static** StopwordsHandler getStopWordsHandler() {  
  
 WordsFromFile wordsFromFile = **new** WordsFromFile();  
  
 wordsFromFile.setStopwords(Paths.*get*(***STOPWORDS\_LOCATION***).toFile());  
  
 **return** wordsFromFile;  
 }  
}

# common/src/main/java/twitter/classification/common code listings

## config

**package** twitter.classification.common.config;  
  
**public interface** ConfigurationKey {  
  
 */\*\*  
 \* Returns the name of a configuration key  
 \*  
 \** ***@return*** *configuration key String  
 \*/* String getName();  
}

## exceptions

**package** twitter.classification.common.exceptions;  
  
**public class** ProcessingClientException **extends** Exception {  
  
 **public** ProcessingClientException(String message) {  
  
 **super**(message);  
 }  
  
 **public** ProcessingClientException(Exception exception) {  
  
 **super**(exception);  
 }  
}

**package** twitter.classification.common.exceptions;  
  
**import static** java.lang.String.*format*;  
  
**public class** ProcessingResponseException **extends** ProcessingClientException {  
  
 **private int statusCode**;  
 **private** String **responseContent**;  
  
 **public** ProcessingResponseException(**int** statusCode, String responseContent) {  
  
 **super**(*format*(**"Status code: %d, response responseContent: %s"**, statusCode, responseContent));  
  
 **this**.**statusCode** = statusCode;  
 **this**.**responseContent** = responseContent;  
 }  
  
 **public int** getStatusCode() {  
  
 **return statusCode**;  
 }  
  
 **public** String getResponseContent() {  
  
 **return responseContent**;  
 }  
}

## models

**package** twitter.classification.common.models;  
  
**import** java.io.Serializable;  
  
**import** org.codehaus.jackson.annotate.JsonProperty;  
  
**public class** ClassificationValueForTweets **implements** Serializable {  
  
 @JsonProperty(**"tweetText"**)  
 **private** String **tweetText**;  
 @JsonProperty(**"classificationValue"**)  
 **private** String **classificationValue**;  
 @JsonProperty(**"id"**)  
 **private int id**;  
  
 **public** ClassificationValueForTweets() {  
 }  
  
 **public** String getTweetText() {  
  
 **return tweetText**;  
 }  
  
 **public void** setTweetText(String tweetText) {  
  
 **this**.**tweetText** = tweetText;  
 }  
  
 **public** String getClassificationValue() {  
  
 **return classificationValue**;  
 }  
  
 **public void** setClassificationValue(String classificationValue) {  
  
 **this**.**classificationValue** = classificationValue;  
 }  
  
 **public int** getId() {  
  
 **return id**;  
 }  
  
 **public void** setId(**int** id) {  
  
 **this**.**id** = id;  
 }  
}

**package** twitter.classification.common.models;  
  
**import** java.io.Serializable;  
  
**import** org.codehaus.jackson.annotate.JsonProperty;  
  
**public class** ClassifierStatusResponse **implements** Serializable {  
  
 @JsonProperty(**"isRunning"**)  
 **private** Boolean **isRunning**;  
  
 **public** ClassifierStatusResponse() {  
 }  
  
 **public** Boolean getRunning() {  
  
 **return isRunning**;  
 }  
  
 **public** ClassifierStatusResponse setRunning(Boolean running) {  
  
 **isRunning** = running;  
 **return this**;  
 }  
}

**package** twitter.classification.common.models;  
  
**import** java.io.Serializable;  
  
**import** org.codehaus.jackson.annotate.JsonProperty;  
  
**public class** DashBoardOverviewResponse **implements** Serializable {  
  
 @JsonProperty(**"totalHashtags"**)  
 **private** Integer **totalHashtags**;  
 @JsonProperty(**"totalUsernames"**)  
 **private** Integer **totalUsernames**;  
 @JsonProperty(**"totalRumours"**)  
 **private** Integer **totalRumours**;  
 @JsonProperty(**"totalNonRumours"**)  
 **private** Integer **totalNonRumours**;  
 @JsonProperty(**"totalClassifications"**)  
 **private** Integer **totalClassifications**;  
 @JsonProperty(**"totalTweets"**)  
 **private** Integer **totalTweets**;  
  
 **public** DashBoardOverviewResponse() {  
 }  
  
 **public** Integer getTotalHashtags() {  
  
 **return totalHashtags**;  
 }  
  
 **public void** setTotalHashtags(Integer totalHashtags) {  
  
 **this**.**totalHashtags** = totalHashtags;  
 }  
  
 **public** Integer getTotalUsernames() {  
  
 **return totalUsernames**;  
 }  
  
 **public void** setTotalUsernames(Integer totalUsernames) {  
  
 **this**.**totalUsernames** = totalUsernames;  
 }  
  
 **public** Integer getTotalRumours() {  
  
 **return totalRumours**;  
 }  
  
 **public void** setTotalRumours(Integer totalRumours) {  
  
 **this**.**totalRumours** = totalRumours;  
 }  
  
 **public** Integer getTotalNonRumours() {  
  
 **return totalNonRumours**;  
 }  
  
 **public void** setTotalNonRumours(Integer totalNonRumours) {  
  
 **this**.**totalNonRumours** = totalNonRumours;  
 }  
  
 **public** Integer getTotalClassifications() {  
  
 **return totalClassifications**;  
 }  
  
 **public void** setTotalClassifications(Integer totalClassifications) {  
  
 **this**.**totalClassifications** = totalClassifications;  
 }  
  
 **public** Integer getTotalTweets() {  
  
 **return totalTweets**;  
 }  
  
 **public void** setTotalTweets(Integer totalTweets) {  
  
 **this**.**totalTweets** = totalTweets;  
 }  
  
 **public** DashBoardOverviewResponse setAllToZero() {  
  
 **this**.**totalClassifications** = 0;  
 **this**.**totalHashtags** = 0;  
 **this**.**totalNonRumours** = 0;  
 **this**.**totalRumours** = 0;  
 **this**.**totalTweets** = 0;  
 **this**.**totalUsernames** = 0;  
 **return this**;  
 }  
}

**package** twitter.classification.common.models;  
  
**import** java.io.Serializable;  
**import** java.util.ArrayList;  
**import** java.util.List;  
  
**import** org.codehaus.jackson.annotate.JsonProperty;  
  
**public class** DashBoardServiceStatusResponse **implements** Serializable {  
  
 @JsonProperty(**"serviceList"**)  
 **private** List<ServiceItem> **serviceList**;  
  
 **public** DashBoardServiceStatusResponse() {  
  
 **this**.**serviceList** = **new** ArrayList<>();  
 }  
  
 **public** List<ServiceItem> getServiceList() {  
  
 **return serviceList**;  
 }  
  
 **public void** setServiceList(List<ServiceItem> serviceList) {  
  
 **this**.**serviceList** = serviceList;  
 }  
  
 **public void** addServiceItem(ServiceItem serviceItem) {  
  
 **this**.**serviceList**.add(serviceItem);  
 }  
}

**package** twitter.classification.common.models;  
  
**import** java.io.Serializable;  
  
**import** org.codehaus.jackson.annotate.JsonProperty;  
  
**public class** HashtagResults **implements** Serializable {  
  
 @JsonProperty(**"hashtagValue"**)  
 **private** String **hashtagValue**;  
 @JsonProperty(**"countOfRumours"**)  
 **private** Integer **countOfRumours**;  
 @JsonProperty(**"countOfNonRumours"**)  
 **private** Integer **countOfNonRumours**;  
 @JsonProperty(**"totalCountOfClassifications"**)  
 **private** Integer **totalCountOfClassifications**;  
  
 **public** HashtagResults() {  
 }  
  
 **public** String getHashtagValue() {  
  
 **return hashtagValue**;  
 }  
  
 **public void** setHashtagValue(String hashtagValue) {  
  
 **this**.**hashtagValue** = hashtagValue;  
 }  
  
 **public** Integer getCountOfRumours() {  
  
 **return countOfRumours**;  
 }  
  
 **public void** setCountOfRumours(Integer countOfRumours) {  
  
 **this**.**countOfRumours** = countOfRumours;  
 }  
  
 **public** Integer getCountOfNonRumours() {  
  
 **return countOfNonRumours**;  
 }  
  
 **public void** setCountOfNonRumours(Integer countOfNonRumours) {  
  
 **this**.**countOfNonRumours** = countOfNonRumours;  
 }  
  
 **public** Integer getTotalCountOfClassifications() {  
  
 **return totalCountOfClassifications**;  
 }  
  
 **public void** setTotalCountOfClassifications(Integer totalCountOfClassifications) {  
  
 **this**.**totalCountOfClassifications** = totalCountOfClassifications;  
 }  
}

**package** twitter.classification.common.models;  
  
**import** org.codehaus.jackson.annotate.JsonProperty;  
  
**public class** PreProcessorStatusResponse {  
  
 @JsonProperty(**"isRunning"**)  
 **private** Boolean **isRunning**;  
  
 **public** PreProcessorStatusResponse() {  
 }  
  
 **public** Boolean getRunning() {  
  
 **return isRunning**;  
 }  
  
 **public** PreProcessorStatusResponse setRunning(Boolean running) {  
  
 **isRunning** = running;  
 **return this**;  
 }  
}

**package** twitter.classification.common.models;  
  
**import** java.io.Serializable;  
  
**import** org.codehaus.jackson.annotate.JsonProperty;  
  
**public class** SearchResultsResponse **implements** Serializable {  
  
 @JsonProperty(**"countOfRumours"**)  
 **private** Integer **countOfRumours**;  
 @JsonProperty(**"countOfNonRumours"**)  
 **private** Integer **countOfNonRumours**;  
 @JsonProperty(**"totalCountOfClassifications"**)  
 **private** Integer **totalCountOfClassifications**;  
  
 **public** SearchResultsResponse() {  
 }  
  
 **public** Integer getCountOfRumours() {  
  
 **return countOfRumours**;  
 }  
  
 **public void** setCountOfRumours(Integer countOfRumours) {  
  
 **this**.**countOfRumours** = countOfRumours;  
 }  
  
 **public** Integer getCountOfNonRumours() {  
  
 **return countOfNonRumours**;  
 }  
  
 **public void** setCountOfNonRumours(Integer countOfNonRumours) {  
  
 **this**.**countOfNonRumours** = countOfNonRumours;  
 }  
  
 **public** Integer getTotalCountOfClassifications() {  
  
 **return totalCountOfClassifications**;  
 }  
  
 **public void** setTotalCountOfClassifications(Integer totalCountOfClassifications) {  
  
 **this**.**totalCountOfClassifications** = totalCountOfClassifications;  
 }  
}

**package** twitter.classification.common.models;  
  
**import** java.io.Serializable;  
  
**import** org.codehaus.jackson.annotate.JsonProperty;  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** com.fasterxml.jackson.annotation.JsonInclude;  
  
**public class** ServiceItem **implements** Serializable {  
  
 **private static final** Logger ***logger*** = LoggerFactory.*getLogger*(ServiceItem.**class**);  
  
 @JsonProperty(**"serviceName"**)  
 **private** String **serviceName**;  
 @JsonProperty(**"isRunning"**)  
 **private** Boolean **isRunning**;  
 @JsonProperty(**"filterList"**)  
 @JsonInclude(JsonInclude.Include.***NON\_NULL***)  
 **private** String **filterList**;  
  
 **public** ServiceItem() {  
 }  
  
 **public** ServiceItem(String serviceName, Boolean isRunning, String... filterList) {  
  
 **this**.**serviceName** = serviceName;  
 **this**.**isRunning** = isRunning;  
  
 **if** (filterList.**length** != 0) {  
  
 **this**.**filterList** = filterList[0];  
 }  
 }  
  
 **public** String getServiceName() {  
  
 **return serviceName**;  
 }  
  
 **public void** setServiceName(String serviceName) {  
  
 **this**.**serviceName** = serviceName;  
 }  
  
 **public** Boolean getRunning() {  
  
 **return isRunning**;  
 }  
  
 **public void** setRunning(Boolean running) {  
  
 **isRunning** = running;  
 }  
  
 **public** String getFilterList() {  
  
 **return filterList**;  
 }  
  
 **public void** setFilterList(String filterList) {  
  
 **this**.**filterList** = filterList;  
 }  
}

**package** twitter.classification.common.models;  
  
**import** java.io.Serializable;  
  
**import** org.codehaus.jackson.annotate.JsonProperty;  
  
**public class** SuggestedSearchResult **implements** Serializable {  
  
 @JsonProperty(**"value"**)  
 **private** String **value**;  
  
 **public** SuggestedSearchResult() {  
 }  
  
 **public** String getValue() {  
  
 **return value**;  
 }  
  
 **public** SuggestedSearchResult setValue(String value) {  
  
 **this**.**value** = value;  
  
 **return this**;  
 }  
}

**package** twitter.classification.common.models;  
  
**import** java.io.Serializable;  
**import** java.util.ArrayList;  
**import** java.util.List;  
  
**import** org.codehaus.jackson.annotate.JsonProperty;  
  
**public class** SuggestedSearchTermsResponse **implements** Serializable {  
  
 @JsonProperty(**"suggestedTerms"**)  
 **private** List<SuggestedSearchResult> **searchResultList**;  
  
 **public** SuggestedSearchTermsResponse() {  
  
 **searchResultList** = **new** ArrayList<>();  
 }  
  
 **public** List<SuggestedSearchResult> getSearchResultList() {  
  
 **return searchResultList**;  
 }  
  
 **public** SuggestedSearchTermsResponse setSearchResultList(List<SuggestedSearchResult> searchResultList) {  
  
 **this**.**searchResultList** = searchResultList;  
  
 **return this**;  
 }  
  
 **public void** addSuggestedSearchResult(SuggestedSearchResult suggestedSearchResult) {  
  
 **this**.**searchResultList**.add(suggestedSearchResult);  
 }  
}

**package** twitter.classification.common.models;  
  
**import** java.io.Serializable;  
  
**import** org.codehaus.jackson.annotate.JsonProperty;  
  
**public class** TimeLineForTweets **implements** Serializable {  
  
 @JsonProperty(**"rumoursLastHour"**)  
 **private** Long **rumoursLastHour**;  
 @JsonProperty(**"nonRumoursLastHour"**)  
 **private** Long **nonRumoursLastHour**;  
 @JsonProperty(**"rumoursOverOneHour"**)  
 **private** Long **rumoursOverOneHour**;  
 @JsonProperty(**"nonRumoursOverOneHour"**)  
 **private** Long **nonRumoursOverOneHour**;  
 @JsonProperty(**"rumoursOverTwoHour"**)  
 **private** Long **rumoursOverTwoHour**;  
 @JsonProperty(**"nonRumoursOverTwoHour"**)  
 **private** Long **nonRumoursOverTwoHour**;  
 @JsonProperty(**"rumoursOverThreeHour"**)  
 **private** Long **rumoursOverThreeHour**;  
 @JsonProperty(**"nonRumoursOverThreeHour"**)  
 **private** Long **nonRumoursOverThreeHour**;  
 @JsonProperty(**"rumoursOverFourHour"**)  
 **private** Long **rumoursOverFourHour**;  
 @JsonProperty(**"nonRumoursOverFourHour"**)  
 **private** Long **nonRumoursOverFourHour**;  
  
 **public** TimeLineForTweets() {  
 }  
  
 **public** Long getRumoursLastHour() {  
  
 **return rumoursLastHour**;  
 }  
  
 **public** TimeLineForTweets setRumoursLastHour(Long rumoursLastHour) {  
  
 **this**.**rumoursLastHour** = rumoursLastHour;  
  
 **return this**;  
 }  
  
 **public** Long getNonRumoursLastHour() {  
  
 **return nonRumoursLastHour**;  
 }  
  
 **public** TimeLineForTweets setNonRumoursLastHour(Long nonRumoursLastHour) {  
  
 **this**.**nonRumoursLastHour** = nonRumoursLastHour;  
  
 **return this**;  
 }  
  
 **public** Long getRumoursOverOneHour() {  
  
 **return rumoursOverOneHour**;  
 }  
  
 **public** TimeLineForTweets setRumoursOverOneHour(Long rumoursOverOneHour) {  
  
 **this**.**rumoursOverOneHour** = rumoursOverOneHour;  
  
 **return this**;  
 }  
  
 **public** Long getNonRumoursOverOneHour() {  
  
 **return nonRumoursOverOneHour**;  
 }  
  
 **public** TimeLineForTweets setNonRumoursOverOneHour(Long nonRumoursOverOneHour) {  
  
 **this**.**nonRumoursOverOneHour** = nonRumoursOverOneHour;  
  
 **return this**;  
 }  
  
 **public** Long getRumoursOverTwoHour() {  
  
 **return rumoursOverTwoHour**;  
 }  
  
 **public** TimeLineForTweets setRumoursOverTwoHour(Long rumoursOverTwoHour) {  
  
 **this**.**rumoursOverTwoHour** = rumoursOverTwoHour;  
  
 **return this**;  
 }  
  
 **public** Long getNonRumoursOverTwoHour() {  
  
 **return nonRumoursOverTwoHour**;  
 }  
  
 **public** TimeLineForTweets setNonRumoursOverTwoHour(Long nonRumoursOverTwoHour) {  
  
 **this**.**nonRumoursOverTwoHour** = nonRumoursOverTwoHour;  
  
 **return this**;  
 }  
  
 **public** Long getRumoursOverThreeHour() {  
  
 **return rumoursOverThreeHour**;  
 }  
  
 **public** TimeLineForTweets setRumoursOverThreeHour(Long rumoursOverThreeHour) {  
  
 **this**.**rumoursOverThreeHour** = rumoursOverThreeHour;  
  
 **return this**;  
 }  
  
 **public** Long getNonRumoursOverThreeHour() {  
  
 **return nonRumoursOverThreeHour**;  
 }  
  
 **public** TimeLineForTweets setNonRumoursOverThreeHour(Long nonRumoursOverThreeHour) {  
  
 **this**.**nonRumoursOverThreeHour** = nonRumoursOverThreeHour;  
  
 **return this**;  
 }  
  
 **public** Long getRumoursOverFourHour() {  
  
 **return rumoursOverFourHour**;  
 }  
  
 **public** TimeLineForTweets setRumoursOverFourHour(Long rumoursOverFourHour) {  
  
 **this**.**rumoursOverFourHour** = rumoursOverFourHour;  
  
 **return this**;  
 }  
  
 **public** Long getNonRumoursOverFourHour() {  
  
 **return nonRumoursOverFourHour**;  
 }  
  
 **public** TimeLineForTweets setNonRumoursOverFourHour(Long nonRumoursOverFourHour) {  
  
 **this**.**nonRumoursOverFourHour** = nonRumoursOverFourHour;  
  
 **return this**;  
 }  
}

**package** twitter.classification.common.models;  
  
**import** java.io.Serializable;  
**import** java.util.ArrayList;  
**import** java.util.List;  
  
**import** org.codehaus.jackson.annotate.JsonProperty;  
  
**public class** TopHashtagsResponse **implements** Serializable {  
  
 @JsonProperty(**"topHashtagResults"**)  
 **private** List<HashtagResults> **hashtagResultsList**;  
  
 **public** TopHashtagsResponse() {  
  
 **this**.**hashtagResultsList** = **new** ArrayList<>();  
 }  
  
 **public** List<HashtagResults> getHashtagResultsList() {  
  
 **return hashtagResultsList**;  
 }  
  
 **public void** setHashtagResultsList(List<HashtagResults> hashtagResultsList) {  
  
 **this**.**hashtagResultsList** = hashtagResultsList;  
 }  
  
 **public void** addHashtagResult(HashtagResults hashtagResults) {  
  
 **this**.**hashtagResultsList**.add(hashtagResults);  
 }  
}

**package** twitter.classification.common.models;  
  
**import** java.io.Serializable;  
**import** java.util.ArrayList;  
**import** java.util.List;  
  
**import** org.codehaus.jackson.annotate.JsonProperty;  
  
**public class** TopUsersResponse **implements** Serializable {  
  
 @JsonProperty(**"topUsersResults"**)  
 **private** List<UserResults> **userResultsList**;  
  
 **public** TopUsersResponse() {  
  
 **this**.**userResultsList** = **new** ArrayList<>();  
 }  
  
 **public** List<UserResults> getUserResultsList() {  
  
 **return userResultsList**;  
 }  
  
 **public void** setUserResultsList(List<UserResults> userResultsList) {  
  
 **this**.**userResultsList** = userResultsList;  
 }  
  
 **public void** addUserResult(UserResults userResults) {  
  
 **this**.**userResultsList**.add(userResults);  
 }  
}

**package** twitter.classification.common.models;  
  
**import** java.io.Serializable;  
  
**import** org.codehaus.jackson.annotate.JsonProperty;  
  
**public class** TwitterStreamResponse **implements** Serializable {  
  
 @JsonProperty(**"isRunning"**)  
 **private** Boolean **isRunning**;  
  
 @JsonProperty(**"filterList"**)  
 **private** String **filterList**;  
  
 **public** TwitterStreamResponse() {  
 }  
  
 **public** Boolean getRunning() {  
  
 **return isRunning**;  
 }  
  
 **public** TwitterStreamResponse setRunning(Boolean running) {  
  
 **isRunning** = running;  
 **return this**;  
 }  
  
 **public** String getFilterList() {  
  
 **return filterList**;  
 }  
  
 **public** TwitterStreamResponse setFilterList(String filterList) {  
  
 **this**.**filterList** = filterList;  
 **return this**;  
 }  
}

**package** twitter.classification.common.models;  
  
**import** java.io.Serializable;  
  
**import** org.codehaus.jackson.annotate.JsonProperty;  
  
**public class** UserResults **implements** Serializable {  
  
 @JsonProperty(**"username"**)  
 **private** String **username**;  
 @JsonProperty(**"countOfRumours"**)  
 **private** Integer **countOfRumours**;  
 @JsonProperty(**"countOfNonRumours"**)  
 **private** Integer **countOfNonRumours**;  
 @JsonProperty(**"totalCountOfClassifications"**)  
 **private** Integer **totalCountOfClassifications**;  
  
 **public** UserResults() {  
 }  
  
 **public** String getUsername() {  
  
 **return username**;  
 }  
  
 **public void** setUsername(String username) {  
  
 **this**.**username** = username;  
 }  
  
 **public** Integer getCountOfRumours() {  
  
 **return countOfRumours**;  
 }  
  
 **public void** setCountOfRumours(Integer countOfRumours) {  
  
 **this**.**countOfRumours** = countOfRumours;  
 }  
  
 **public** Integer getCountOfNonRumours() {  
  
 **return countOfNonRumours**;  
 }  
  
 **public void** setCountOfNonRumours(Integer countOfNonRumours) {  
  
 **this**.**countOfNonRumours** = countOfNonRumours;  
 }  
  
 **public** Integer getTotalCountOfClassifications() {  
  
 **return totalCountOfClassifications**;  
 }  
  
 **public void** setTotalCountOfClassifications(Integer totalCountOfClassifications) {  
  
 **this**.**totalCountOfClassifications** = totalCountOfClassifications;  
 }  
}

## persist

**package** twitter.classification.common.persist;  
  
**import** java.lang.annotation.ElementType;  
**import** java.lang.annotation.Retention;  
**import** java.lang.annotation.RetentionPolicy;  
**import** java.lang.annotation.Target;  
  
*/\*\*  
 \* Annotation to declare Database Column,  
 \* Used in Java Reflection to map result set  
 \* to the Java class  
 \*/*@Retention(RetentionPolicy.***RUNTIME***)  
@Target({ElementType.***FIELD***})  
**public** @**interface** Column {  
  
 String name();  
}

**package** twitter.classification.common.persist;  
  
**import** java.sql.Connection;  
  
**public interface** ConnectionFactory {  
  
 Connection getConnection();  
}

**package** twitter.classification.common.persist;  
  
**import** java.sql.Connection;  
**import** java.sql.SQLException;  
  
**import** javax.inject.Inject;  
  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** twitter.classification.common.persist.jdbc.MySqlConnectionFactory;  
**import** twitter.classification.common.system.ConfigurationVariable;  
**import** twitter.classification.common.system.helper.ConfigurationVariableParam;  
  
**public class** ConnectionManager {  
  
 **private static final** Logger ***logger*** = LoggerFactory.*getLogger*(ConnectionManager.**class**);  
  
 **private static final** ThreadLocal<Connection> ***connectionManager*** = **new** ThreadLocal<>();  
  
 **private** String **dbUsername**;  
 **private** String **dbPassword**;  
 **private** String **dbUrl**;  
  
 @Inject  
 **public** ConnectionManager(  
 @ConfigurationVariableParam(variable = ConfigurationVariable.***DB\_USERNAME***) String dbUsername,  
 @ConfigurationVariableParam(variable = ConfigurationVariable.***DB\_PASSWORD***) String dbPassword,  
 @ConfigurationVariableParam(variable = ConfigurationVariable.***DB\_URL***) String dbUrl  
 ) {  
  
 **this**.**dbUsername** = dbUsername;  
 **this**.**dbPassword** = dbPassword;  
 **this**.**dbUrl** = dbUrl;  
 }  
  
 **public** Connection getConnection() {  
  
 openConnection();  
  
 **return *connectionManager***.get();  
 }  
  
 **void** openConnection() {  
  
 Connection connection = ***connectionManager***.get();  
  
 **if** (connection == **null**) {  
  
 ConnectionFactory connectionFactory = **new** MySqlConnectionFactory(**dbUsername**, **dbPassword**, **dbUrl**);  
 connection = connectionFactory.getConnection();  
 ***connectionManager***.set(connection);  
 }  
 }  
  
 **void** closeConnection() {  
  
 Connection connection = ***connectionManager***.get();  
  
 **if** (connection != **null**) {  
  
 **try** {  
  
 connection.close();  
 } **catch** (SQLException exception) {  
  
 ***logger***.error(**"Issue closing SQL Connection"**, exception);  
 } **finally** {  
  
 ***connectionManager***.remove();  
 }  
 }  
 }  
}

**package** twitter.classification.common.persist;  
  
**import** java.lang.annotation.ElementType;  
**import** java.lang.annotation.Retention;  
**import** java.lang.annotation.RetentionPolicy;  
**import** java.lang.annotation.Target;  
  
@Retention(RetentionPolicy.***RUNTIME***)  
@Target(ElementType.***METHOD***)  
**public** @**interface** DbConnection {  
}

**package** twitter.classification.common.persist;  
  
**import** javax.inject.Inject;  
  
**import** org.aopalliance.intercept.MethodInterceptor;  
**import** org.aopalliance.intercept.MethodInvocation;  
  
**public class** DbConnectionResolver **implements** MethodInterceptor {  
  
 **private** ConnectionManager **connectionManager**;  
  
 @Inject  
 **public** DbConnectionResolver(ConnectionManager connectionManager) {  
  
 **this**.**connectionManager** = connectionManager;  
 }  
  
 @Override  
 **public** Object invoke(MethodInvocation invocation) **throws** Throwable {  
  
 **try** {  
  
 **connectionManager**.openConnection();  
 **return** invocation.proceed();  
 } **finally** {  
  
 **connectionManager**.closeConnection();  
 }  
 }  
}

**package** twitter.classification.common.persist;  
  
**import** java.lang.annotation.Retention;  
**import** java.lang.annotation.RetentionPolicy;  
  
*/\*\*  
 \* Annotation to declare Database Entity,  
 \* Used in Java Reflection to map result set  
 \* to the Java class  
 \*/*@Retention(RetentionPolicy.***RUNTIME***)  
**public** @**interface** Entity {  
}

**package** twitter.classification.common.persist;  
  
**import** java.lang.reflect.Field;  
**import** java.sql.ResultSet;  
**import** java.sql.ResultSetMetaData;  
**import** java.sql.SQLException;  
**import** java.util.ArrayList;  
**import** java.util.Arrays;  
**import** java.util.List;  
  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
*/\*\*  
 \* Generic mapper to map database result sets to a Java object  
 \* <p>  
 \* Reference: https://oprsteny.com/?p=900  
 \*  
 \** ***@param <T>*** *\*/***public class** ResultSetMapper<T> {  
  
 **public static final** Logger ***logger*** = LoggerFactory.*getLogger*(ResultSetMapper.**class**);  
  
 **public** List<T> mapResultSetToClass(ResultSet resultSet, Class classToMap) {  
  
 List<T> outputList = **new** ArrayList<>();  
 List<Field> fields = getAllFields(**new** ArrayList<>(), classToMap);  
  
 **try** {  
  
 **if** (resultSet != **null**) {  
  
 *// allows access to the database column names* ResultSetMetaData resultSetMetaData = resultSet.getMetaData();  
  
 **while** (resultSet.next()) {  
  
 T mappedClass = (T) classToMap.newInstance();  
  
 **for** (**int** i = 1; i <= resultSetMetaData.getColumnCount(); i++) {  
  
 String columnName = resultSetMetaData.getColumnName(i);  
  
 Object columnValue = resultSet.getObject(i);  
  
 **for** (Field field : fields) {  
  
 *// only want to map the fields which are related to database results* **if** (field.isAnnotationPresent(Column.**class**)) {  
  
 Column column = field.getAnnotation(Column.**class**);  
  
 **if** (column.name().equalsIgnoreCase(columnName) && columnValue != **null**) {  
  
 setProperty(mappedClass, field.getName(), columnValue);  
 **break**;  
 }  
 }  
 }  
 }  
  
 outputList.add(mappedClass);  
 }  
  
 } **else** {  
  
 **return null**;  
 }  
 } **catch** (IllegalAccessException | InstantiationException | SQLException e) {  
  
 logger.error(**"Issue mapping result set to object"**, e);  
 }  
  
 **return** outputList;  
 }  
  
 */\*\*  
 \* Method to get all possible fields including those in the super() class  
 \*  
 \** ***@param*** *fields  
 \** ***@param*** *type  
 \** ***@return*** *\*/* **private** List<Field> getAllFields(List<Field> fields, Class<?> type) {  
 fields.addAll(Arrays.asList(type.getDeclaredFields()));  
  
 **if** (type.getSuperclass() != **null**) {  
 getAllFields(fields, type.getSuperclass());  
 }  
  
 **return** fields;  
 }  
  
 */\*\*  
 \* Method to set the field properties of the class  
 \*  
 \** ***@param*** *clazz  
 \** ***@param*** *fieldName  
 \** ***@param*** *columnValue  
 \*/* **private void** setProperty(Object clazz, String fieldName, Object columnValue) {  
 **try** {  
 Field field;  
 **if** (clazz.getClass().getSuperclass() != **null**) {  
 **try** {  
 field = clazz.getClass().getSuperclass().getDeclaredField(fieldName);  
 } **catch** (NoSuchFieldException exception) {  
 field = clazz.getClass().getDeclaredField(fieldName);  
 }  
 } **else** {  
 field = clazz.getClass().getDeclaredField(fieldName);  
 }  
 *// as the fields are private need to alter it* field.setAccessible(**true**);  
 field.set(clazz, columnValue);  
 } **catch** (NoSuchFieldException | SecurityException | IllegalArgumentException | IllegalAccessException e) {  
  
 logger.error(**"Issue setting field properties"**, e);  
 }  
 }  
}

### jdbc

**package** twitter.classification.common.persist.jdbc;  
  
**import** java.sql.Connection;  
**import** java.sql.DriverManager;  
  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** twitter.classification.common.persist.ConnectionFactory;  
  
**public class** MySqlConnectionFactory **implements** ConnectionFactory {  
  
 **public static final** Logger ***logger*** = LoggerFactory.*getLogger*(MySqlConnectionFactory.**class**);  
  
 **private** String **dbUsername**;  
 **private** String **dbPassword**;  
 **private** String **dbUrl**;  
  
 **public** MySqlConnectionFactory(String dbUsername, String dbPassword, String dbUrl) {  
  
 **this**.**dbUsername** = dbUsername;  
 **this**.**dbPassword** = dbPassword;  
 **this**.**dbUrl** = dbUrl;  
 }  
  
 @Override  
 **public** Connection getConnection() {  
  
 **try** {  
  
 Class.*forName*(**"com.mysql.cj.jdbc.Driver"**);  
 **return** DriverManager.*getConnection*(**dbUrl**, **dbUsername**, **dbPassword**);  
 } **catch** (Exception exception) {  
  
 ***logger***.error(**"Issue getting DB connection"**, exception);  
 }  
  
 **return null**;  
 }  
}

#### queries

**package** twitter.classification.common.persist.jdbc.queries;  
  
**public interface** DbQuery {  
  
 String buildQuery();  
}

#### utils

**package** twitter.classification.common.persist.jdbc.utils;  
  
**import** java.sql.Connection;  
**import** java.sql.PreparedStatement;  
**import** java.sql.SQLException;  
**import** java.util.List;  
  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** twitter.classification.common.persist.ResultSetMapper;  
  
*/\*\*  
 \* Class to run DB queries, such as SELECT, INSERT etc.  
 \*/***public class** DbQueryRunner {  
  
 **private static final** Logger ***logger*** = LoggerFactory.*getLogger*(DbQueryRunner.**class**);  
  
 **private** Connection **connection**;  
  
 **public** DbQueryRunner(Connection connection) {  
  
 **this**.**connection** = connection;  
 }  
  
 */\*\*  
 \* Method for DB Inserts/Deletes  
 \*  
 \** ***@param sql*** *\** ***@param params*** *\*/* **public void** executeUpdate(String sql, Object... params) {  
  
 **try** (PreparedStatement preparedStatement = preparedStatement(sql, params)) {  
  
 **if** (preparedStatement != **null**)  
 preparedStatement.executeUpdate();  
  
 } **catch** (Exception exception) {  
  
 ***logger***.error(**"Issue executing query, "** + sql, exception);  
 }  
 }  
  
 **public** <T> List<T> executeQuery(String sql, Class classToMap, Object... params) **throws** SQLException {  
  
 **try** (PreparedStatement preparedStatement = preparedStatement(sql, params)) {  
  
 **if** (preparedStatement != **null**) {  
 ResultSetMapper<T> resultSetMapper = **new** ResultSetMapper<>();  
  
 **return** resultSetMapper.mapResultSetToClass(preparedStatement.executeQuery(), classToMap);  
 }  
 } **catch** (Exception exception) {  
  
 ***logger***.error(**"Issue executing query, "** + sql, exception);  
 }  
  
 **return null**;  
 }  
  
 */\*\*  
 \* Helper method for setting values of prepared statements  
 \*  
 \** ***@param sql*** *\** ***@param params*** *\** ***@return*** *\** ***@throws*** *SQLException  
 \*/* **private** PreparedStatement preparedStatement(String sql, Object... params) **throws** SQLException {  
  
 **if** (**connection** != **null**) {  
 PreparedStatement preparedStatement = **connection**.prepareStatement(sql);  
  
 **if** (params != **null**) {  
  
 **for** (**int** param = 0; param < params.**length**; param++) {  
  
 **if** (params[param] != **null**) {  
 preparedStatement.setObject(param + 1, params[param]);  
 }  
 }  
 }  
  
 **return** preparedStatement;  
 }  
  
 **return null**;  
 }  
}

## system

**package** twitter.classification.common.system;  
  
**import** twitter.classification.common.config.ConfigurationKey;  
  
**public enum** ConfigurationVariable **implements** ConfigurationKey {  
  
 ***DB\_USERNAME***(**"DB\_USERNAME"**),  
 ***DB\_PASSWORD***(**"DB\_PASSWORD"**),  
 ***DB\_URL***(**"DB\_URL"**),  
 ***CLASSIFICATION\_WEIGHT\_THRESHOLD***(**"CLASSIFICATION\_WEIGHT\_THRESHOLD"**),  
 ***DASHBOARD\_OVERVIEW\_URI***(**"DASHBOARD\_OVERVIEW\_URI"**),  
 ***DASHBOARD\_SERVICE\_STATUS\_URI***(**"DASHBOARD\_SERVICE\_STATUS\_URI"**),  
 ***TWITTER\_STREAM\_STATUS\_URI***(**"TWITTER\_STREAM\_STATUS\_URI"**),  
 ***CLASSIFIER\_STATUS\_URI***(**"CLASSIFIER\_STATUS\_URI"**),  
 ***CLASSIFIER\_CLASSIFICATION\_URI***(**"CLASSIFIER\_CLASSIFICATION\_URI"**),  
 ***PRE\_PROCESSOR\_STATUS\_URI***(**"PRE\_PROCESSOR\_STATUS\_URI"**),  
 ***TOP\_HASHTAGS\_RESULTS\_URI***(**"TOP\_HASHTAGS\_RESULTS\_URI"**),  
 ***TOP\_USERS\_RESULTS\_URI***(**"TOP\_USERS\_RESULTS\_URI"**),  
 ***SEARCH\_RESULTS\_URI***(**"SEARCH\_RESULTS\_URI"**),  
 ***SUGGESTED\_SEARCH\_RESULTS\_URI***(**"SUGGESTED\_SEARCH\_RESULTS\_URI"**),  
 ***TWITTER\_OAUTH\_ACCESS\_KEY***(**"TWITTER\_OAUTH\_ACCESS\_KEY"**),  
 ***TWITTER\_OAUTH\_ACCESS\_SECRET***(**"TWITTER\_OAUTH\_ACCESS\_SECRET"**),  
 ***TWITTER\_OAUTH\_CONSUMER\_KEY***(**"TWITTER\_OAUTH\_CONSUMER\_KEY"**),  
 ***TWITTER\_OAUTH\_CONSUMER\_SECRET***(**"TWITTER\_OAUTH\_CONSUMER\_SECRET"**),  
 ***TWITTER\_FILTER\_LIST***(**"TWITTER\_FILTER\_LIST"**),  
 ***QUEUE\_HOST***(**"QUEUE\_HOST"**),  
 ***QUEUE\_USER***(**"QUEUE\_USER"**),  
 ***QUEUE\_PASSWORD***(**"QUEUE\_PASSWORD"**),  
 ***QUEUE\_NAME***(**"QUEUE\_NAME"**),  
 ***USE\_PRE\_PROCESSING***(**"USE\_PRE\_PROCESSING"**);  
  
 **final** String **name**;  
  
 ConfigurationVariable(String name) {  
  
 **this**.**name** = name;  
 }  
  
 @Override  
 **public** String getName() {  
  
 **return name**;  
 }  
}

### binder

**package** twitter.classification.common.system.binder;  
  
**import** javax.inject.Inject;  
**import** javax.inject.Singleton;  
  
**import** org.glassfish.hk2.api.Factory;  
**import** org.glassfish.hk2.api.InjectionResolver;  
**import** org.glassfish.hk2.api.ServiceLocator;  
**import** org.glassfish.hk2.api.TypeLiteral;  
**import** org.glassfish.hk2.utilities.binding.AbstractBinder;  
**import** org.glassfish.jersey.server.internal.inject.AbstractContainerRequestValueFactory;  
**import** org.glassfish.jersey.server.internal.inject.AbstractValueFactoryProvider;  
**import** org.glassfish.jersey.server.internal.inject.MultivaluedParameterExtractorProvider;  
**import** org.glassfish.jersey.server.internal.inject.ParamInjectionResolver;  
**import** org.glassfish.jersey.server.model.Parameter;  
**import** org.glassfish.jersey.server.spi.internal.ValueFactoryProvider;  
  
**import** twitter.classification.common.system.helper.ConfigurationVariableParam;  
**import** twitter.classification.common.system.helper.FileVariables;  
  
**import static** org.glassfish.jersey.server.model.Parameter.Source.***UNKNOWN***;  
  
**public class** ConfigurationVariableBinder **extends** AbstractBinder {  
  
 @Override  
 **protected void** configure() {  
  
 bind(ConfigurationValueResolverFactory.**class**).to(ValueFactoryProvider.**class**).in(Singleton.**class**);  
 bind(ConfigurationValueResolver.**class**).to(**new** TypeLiteral<InjectionResolver<ConfigurationVariableParam>>() {  
 }).in(Singleton.**class**);  
 }  
  
 @Singleton  
 **public static class** ConfigurationValueResolver **extends** ParamInjectionResolver<ConfigurationVariableParam> {  
  
 @Inject  
 **public** ConfigurationValueResolver() {  
  
 **super**(ConfigurationValueResolverFactory.**class**);  
 }  
 }  
  
 @Singleton  
 **public static class** ConfigurationValueResolverFactory **extends** AbstractValueFactoryProvider {  
  
 @Inject  
 **public** ConfigurationValueResolverFactory(**final** MultivaluedParameterExtractorProvider extractorProvider, **final** ServiceLocator  
 injector) {  
  
 **super**(extractorProvider, injector, ***UNKNOWN***);  
 }  
  
 @Override  
 **protected** Factory<?> createValueFactory(**final** Parameter parameter) {  
  
 **final** Class<?> classType = parameter.getRawType();  
  
 **if** (classType == **null**) {  
 **return null**;  
 }  
  
 **if** (parameter.getAnnotation(ConfigurationVariableParam.**class**) == **null**) {  
 **return null**;  
 }  
  
 **return new** AbstractContainerRequestValueFactory<Object>() {  
  
 @Override  
 **public** Object provide() {  
  
 String value = FileVariables.properties.getProperty(parameter.getAnnotation(ConfigurationVariableParam.**class**).variable().getName());  
  
 **if** (classType.equals(Integer.**class**)) {  
 **return** Integer.valueOf(value);  
 }  
  
 **if** (classType.equals(Boolean.**class**)) {  
 **return** Boolean.parseBoolean(value);  
 }  
  
 **return** value;  
 }  
 };  
 }  
 }  
}

#### factory

**package** twitter.classification.common.system.binder.factory;  
  
**import** org.glassfish.hk2.api.Factory;  
  
**public interface** BaseFactory<T> **extends** Factory<T> {  
  
 @Override  
 **default void** dispose(T instance) {  
 }  
}

### helper

**package** twitter.classification.common.system.helper;  
  
**import** java.lang.annotation.ElementType;  
**import** java.lang.annotation.Retention;  
**import** java.lang.annotation.RetentionPolicy;  
**import** java.lang.annotation.Target;  
  
**import** twitter.classification.common.system.ConfigurationVariable;  
  
@Retention(RetentionPolicy.***RUNTIME***)  
@Target({ElementType.***PARAMETER***, ElementType.***FIELD***})  
**public** @**interface** ConfigurationVariableParam {  
  
 ConfigurationVariable variable();  
}

**package** twitter.classification.common.system.helper;  
  
**import** java.io.IOException;  
**import** java.io.InputStream;  
**import** java.util.Properties;  
  
**import** org.apache.log4j.Level;  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import static** org.apache.log4j.Logger.*getRootLogger*;  
  
**public class** FileVariables {  
  
 **private static final** Logger ***logger*** = LoggerFactory.*getLogger*(FileVariables.**class**);  
  
 **public static** Properties *properties* = **new** Properties();  
  
 **public void** setValuesFromConfigurationFile() {  
  
 InputStream inputStream = getClass().getClassLoader().getResourceAsStream(**"configuration.txt"**);  
  
 **if** (inputStream != **null**) {  
  
 **try** {  
 *properties*.load(inputStream);  
 } **catch** (IOException exception) {  
 ***logger***.error(**"Issue reading configuration values"**, exception);  
 }  
 }  
 }  
  
 **public static void** setLogLevel() {  
  
 String logLevel = FileVariables.*properties*.getProperty(**"LOG\_LEVEL"**);  
  
 *getRootLogger*().setLevel(logLevel != **null** ? Level.*toLevel*(logLevel) : Level.*toLevel*(**"info"**));  
 }  
}

## tweetdetails

### model

**package** twitter.classification.common.tweetdetails.model;  
  
**import** java.io.Serializable;  
  
**import** org.codehaus.jackson.annotate.JsonProperty;  
  
**public class** ClassificationModel **implements** Serializable {  
  
 @JsonProperty(**"label"**)  
 **private** String **classificationLabel**;  
  
 **public** ClassificationModel() {  
  
  
 }  
  
 **public** String getClassificationLabel() {  
  
 **return classificationLabel**;  
 }  
  
 **public** ClassificationModel setClassificationLabel(String classificationLabel) {  
  
 **this**.**classificationLabel** = classificationLabel;  
  
 **return this**;  
 }  
}

**package** twitter.classification.common.tweetdetails.model;  
  
**import** java.io.Serializable;  
**import** java.util.ArrayList;  
**import** java.util.List;  
  
**import** org.codehaus.jackson.annotate.JsonProperty;  
  
**public class** PreProcessedItem **implements** Serializable {  
  
 @JsonProperty(**"hashtags"**)  
 **private** List<String> **hashtags**;  
  
 @JsonProperty(**"username"**)  
 **private** String **username**;  
  
 @JsonProperty(**"userId"**)  
 **private** Long **userId**;  
  
 @JsonProperty(**"tweetId"**)  
 **private** Long **tweetId**;  
  
 @JsonProperty(**"processedTweetBody"**)  
 **private** String **processedTweetBody**;  
  
 @JsonProperty(**"originalTweetBody"**)  
 **private** String **originalTweetBody**;  
  
 **public** PreProcessedItem() {  
  
 **this**.**hashtags** = **new** ArrayList<>();  
 }  
  
 **public** List<String> getHashtags() {  
  
 **return hashtags**;  
 }  
  
 **public void** addHashtag(String hashtag) {  
  
 **this**.**hashtags**.add(hashtag);  
 }  
  
 **public void** setHashtags(List<String> hashtags) {  
  
 **this**.**hashtags** = hashtags;  
 }  
  
 **public** String getUsername() {  
  
 **return username**;  
 }  
  
 **public void** setUsername(String username) {  
  
 **this**.**username** = username;  
 }  
  
 **public** Long getUserId() {  
  
 **return userId**;  
 }  
  
 **public void** setUserId(Long userId) {  
  
 **this**.**userId** = userId;  
 }  
  
 **public** Long getTweetId() {  
  
 **return tweetId**;  
 }  
  
 **public void** setTweetId(Long tweetId) {  
  
 **this**.**tweetId** = tweetId;  
 }  
  
 **public** String getProcessedTweetBody() {  
  
 **return processedTweetBody**;  
 }  
  
 **public void** setProcessedTweetBody(String processedTweetBody) {  
  
 **this**.**processedTweetBody** = processedTweetBody;  
 }  
  
 **public** String getOriginalTweetBody() {  
  
 **return originalTweetBody**;  
 }  
  
 **public void** setOriginalTweetBody(String originalTweetBody) {  
  
 **this**.**originalTweetBody** = originalTweetBody;  
 }  
}

**package** twitter.classification.common.tweetdetails.model;  
  
**import** java.io.Serializable;  
  
**import** org.codehaus.jackson.annotate.JsonProperty;  
  
**public class** ProcessedStatusResponse **implements** Serializable {  
  
 @JsonProperty(**"tweet\_body"**)  
 **private** String **tweetBody**;  
  
 @JsonProperty(**"username"**)  
 **private** String **userName**;  
  
 @JsonProperty(**"hashtag"**)  
 **private** String **hashtag**;  
  
 **public** ProcessedStatusResponse() {  
  
 }  
  
 **public** String getTweetBody() {  
  
 **return tweetBody**;  
 }  
  
 **public void** setTweetBody(String tweetBody) {  
  
 **this**.**tweetBody** = tweetBody;  
 }  
  
 **public** String getUserName() {  
  
 **return userName**;  
 }  
  
 **public void** setUserName(String userName) {  
  
 **this**.**userName** = userName;  
 }  
  
 **public** String getHashtag() {  
  
 **return hashtag**;  
 }  
  
 **public void** setHashtag(String hashtag) {  
  
 **this**.**hashtag** = hashtag;  
 }  
}

**package** twitter.classification.common.tweetdetails.model;  
  
  
**import** org.codehaus.jackson.annotate.JsonProperty;  
  
**public class** ProcessedTweetModel **extends** PreProcessedItem {  
  
 @JsonProperty(**"classificationValue"**)  
 **private** String **classificationValue**;  
  
 **public** ProcessedTweetModel(PreProcessedItem preProcessedItem) {  
  
 setHashtags(preProcessedItem.getHashtags());  
 setOriginalTweetBody(preProcessedItem.getOriginalTweetBody());  
 setProcessedTweetBody(preProcessedItem.getProcessedTweetBody());  
 setTweetId(preProcessedItem.getTweetId());  
 setUserId(preProcessedItem.getUserId());  
 setUsername(preProcessedItem.getUsername());  
 }  
  
 **public** String getClassificationValue() {  
  
 **return classificationValue**;  
 }  
  
 **public void** setClassificationValue(String classificationValue) {  
  
 **this**.**classificationValue** = classificationValue;  
 }  
}

### processing

**package** twitter.classification.common.tweetdetails.processing;  
  
**import** java.util.Optional;  
  
**import** javax.ws.rs.core.Response;  
  
**import** twitter.classification.common.exceptions.ProcessingClientException;  
**import** twitter.classification.common.exceptions.ProcessingResponseException;  
  
**public class** ProcessResponse {  
  
 */\*\*  
 \* Generic response processor for processing http responses, used  
 \* by clients in the various services, as the only thing  
 \* that changes in processing is the type of class which needs to  
 \* be read  
 \*  
 \** ***@param response*** *Response  
 \** ***@param clazz*** *genericClazz  
 \** ***@param <T>*** *genericClazz  
 \** ***@return*** *<T>  
 \** ***@throws*** *ProcessingClientException When processing fails  
 \*/* **public static** <T> Optional<T> processResponse(Response response, Class<T> clazz) **throws** ProcessingClientException {  
  
 **int** responseStatus = response.getStatus();  
  
 **if** (responseStatus == Response.Status.***OK***.getStatusCode()) {  
  
 **try** {  
  
 **return** Optional.*of*(response.readEntity(clazz));  
  
 } **catch** (Exception readingException) {  
  
 **throw new** ProcessingClientException(readingException);  
 }  
 } **else** {  
  
 String content = **""**;  
 **try** {  
  
 **if** (response.hasEntity()) {  
  
 content = response.readEntity(String.**class**);  
 }  
 } **catch** (Exception readingException) {  
  
 **throw new** ProcessingClientException(readingException);  
 }  
  
 **throw new** ProcessingResponseException(responseStatus, content);  
 }  
 }  
}

**package** twitter.classification.common.tweetdetails.processing;  
  
**public class** TweetBodyProcessor {  
  
 **public** String processTweetBody(String tweetBody) {  
  
 tweetBody = tweetBody.toLowerCase();  
  
 *// fix up any new lines/character returns* tweetBody = tweetBody.replaceAll(**"(\r\n|\n|\r)"**, **" "**);  
  
 *// remove the # before hashtags as they don't add to classification* tweetBody = tweetBody.replaceAll(**"#([^\\s]+)"**, **"$1"**);  
  
 *// remove the @ before usernames as they don't add to classification* tweetBody = tweetBody.replaceAll(**"@([^\\s]+)"**, **"$1"**);  
  
 *// remove the complete urls as they don't add to classification* tweetBody = tweetBody.replaceAll(**"((www\\.[^\\s]+)|(https?://[^\\s]+))"**, **""**);  
  
 *// remove special characters* tweetBody = tweetBody.replaceAll(**"([^a-zA-Z0-9]+)"**, **" "**);  
  
 *// to fix up any double/triple white spaces* tweetBody = tweetBody.replaceAll(**"( {2,})"**, **" "**);  
  
 *// will trim any leading white spaces* tweetBody = tweetBody.trim();  
  
 **return** tweetBody;  
 }  
}

# db schema code listings

**CREATE TABLE IF NOT EXISTS** classification\_types  
(  
 **id TINYINT PRIMARY KEY NOT NULL AUTO\_INCREMENT**,  
 **classification\_value VARCHAR**(11),  
 **classification\_code VARCHAR**(3),  
 **CONSTRAINT** code\_and\_value\_unique **UNIQUE** (**classification\_value**, **classification\_code**)  
) **CHARACTER SET** utf8;  
  
**INSERT IGNORE INTO** classification\_types (**classification\_value**, **classification\_code**)  
**VALUES** (**'undefined'**, **'UDF'**), (**'rumour'**, **'RMR'**), (**'non-rumour'**, **'NOR'**);  
  
**CREATE TABLE IF NOT EXISTS** tweets  
(  
 **id BIGINT PRIMARY KEY NOT NULL AUTO\_INCREMENT**,  
 **tweet\_id BIGINT**,  
 **original\_tweet\_text VARCHAR**(300),  
 **processed\_tweet\_text VARCHAR**(300),  
 **classification\_id TINYINT**,  
 **created\_on TIMESTAMP NOT NULL DEFAULT** *now*(),  
 **FOREIGN KEY** (**classification\_id**)  
 **REFERENCES** classification\_types(**id**),  
 **UNIQUE** (**tweet\_id**)  
) **CHARACTER SET** utf8;  
  
**CREATE TABLE IF NOT EXISTS** users  
(  
 **id BIGINT PRIMARY KEY NOT NULL AUTO\_INCREMENT**,  
 **username VARCHAR**(30),  
 **twitter\_id BIGINT**,  
 **created\_on TIMESTAMP NOT NULL DEFAULT** *now*(),  
 **CONSTRAINT** twitter\_id\_and\_username\_unique **UNIQUE** (**username**, **twitter\_id**)  
) **CHARACTER SET** utf8;  
  
**CREATE TABLE IF NOT EXISTS** users\_tweet\_classifications  
(  
 **id BIGINT PRIMARY KEY NOT NULL AUTO\_INCREMENT**,  
 **user\_id BIGINT**,  
 **tweet\_id BIGINT**,  
 **created\_on TIMESTAMP NOT NULL DEFAULT** *now*(),  
 **CONSTRAINT** user\_id\_and\_tweet\_id\_unique **UNIQUE** (**user\_id**, **tweet\_id**),  
 **FOREIGN KEY** (**user\_id**)  
 **REFERENCES** users(**id**),  
 **FOREIGN KEY** (**tweet\_id**)  
 **REFERENCES** tweets(**id**)  
) **CHARACTER SET** utf8;  
  
**CREATE TABLE IF NOT EXISTS** hashtags  
(  
 **id BIGINT PRIMARY KEY NOT NULL AUTO\_INCREMENT**,  
 **hashtag\_value VARCHAR**(30),  
 **created\_on TIMESTAMP NOT NULL DEFAULT** *now*(),  
 **UNIQUE** (**hashtag\_value**)  
) **CHARACTER SET** utf8;  
  
**CREATE TABLE IF NOT EXISTS** hashtag\_tweet\_classifications  
(  
 **id BIGINT PRIMARY KEY NOT NULL AUTO\_INCREMENT**,  
 **hashtag\_id BIGINT**,  
 **tweet\_id BIGINT**,  
 **created\_on TIMESTAMP NOT NULL DEFAULT** *now*(),  
 **CONSTRAINT** hashtag\_id\_and\_tweet\_id\_unique **UNIQUE** (**hashtag\_id**, **tweet\_id**),  
 **FOREIGN KEY** (**hashtag\_id**)  
 **REFERENCES** hashtags(**id**),  
 **FOREIGN KEY** (**tweet\_id**)  
 **REFERENCES** tweets(**id**)  
) **CHARACTER SET** utf8;

# frontend/src/main/java/twitter/classification/web code listings

## application

**package** twitter.classification.web.application;  
  
**import** org.glassfish.jersey.server.ResourceConfig;  
  
**import** twitter.classification.common.system.binder.ConfigurationVariableBinder;  
**import** twitter.classification.common.system.helper.FileVariables;  
**import** twitter.classification.web.application.binder.ClientBinder;  
**import** twitter.classification.web.application.binder.TemplateRenderBinder;  
  
**public class** WebApplication **extends** ResourceConfig {  
  
 */\*\*  
 \* Entry point to the jersey application for the web application,  
 \* will load the configuration values and register the binders which  
 \* bind services for injection later  
 \*/* **public** WebApplication() {  
  
 packages(**"twitter.classification.web.application"**);  
  
 loadConfigurationValues();  
  
 register(**new** ConfigurationVariableBinder());  
 register(**new** TemplateRenderBinder());  
 register(**new** ClientBinder());  
 }  
  
 **private void** loadConfigurationValues() {  
  
 **new** FileVariables().setValuesFromConfigurationFile();  
 }  
}

### binder

**package** twitter.classification.web.application.binder;  
  
**import** org.glassfish.hk2.utilities.binding.AbstractBinder;  
  
**import** twitter.classification.web.clients.AlternativeSearchResultsClient;  
**import** twitter.classification.web.clients.DashBoardOverviewClient;  
**import** twitter.classification.web.clients.DashBoardServiceStatusClient;  
**import** twitter.classification.web.clients.SearchResultsClient;  
**import** twitter.classification.web.clients.TopHashTagsResultsClient;  
**import** twitter.classification.web.clients.TopUsersResultsClient;  
  
**public class** ClientBinder **extends** AbstractBinder {  
  
 */\*\*  
 \* Bind the clients and services which are injected in to the resources  
 \*/* @Override  
 **protected void** configure() {  
  
 bind(DashBoardOverviewClient.**class**).to(DashBoardOverviewClient.**class**);  
 bind(DashBoardServiceStatusClient.**class**).to(DashBoardServiceStatusClient.**class**);  
 bind(TopHashTagsResultsClient.**class**).to(TopHashTagsResultsClient.**class**);  
 bind(TopUsersResultsClient.**class**).to(TopUsersResultsClient.**class**);  
 bind(SearchResultsClient.**class**).to(SearchResultsClient.**class**);  
 bind(AlternativeSearchResultsClient.**class**).to(AlternativeSearchResultsClient.**class**);  
 }  
}

**package** twitter.classification.web.application.binder;  
  
**import** javax.inject.Singleton;  
  
**import** org.glassfish.hk2.utilities.binding.AbstractBinder;  
  
**import** twitter.classification.web.render.HandleBarsTemplateRender;  
**import** twitter.classification.web.render.TemplateRender;  
  
*/\*\*  
 \* Binding the handlebars implementation of the template engine interface  
 \*/***public class** TemplateRenderBinder **extends** AbstractBinder {  
  
 @Override  
 **protected void** configure() {  
  
 bind(HandleBarsTemplateRender.**class**).to(TemplateRender.**class**).in(Singleton.**class**);  
 }  
}

## clients

**package** twitter.classification.web.clients;  
  
**import** java.util.Optional;  
  
**import** javax.inject.Inject;  
**import** javax.ws.rs.client.Client;  
**import** javax.ws.rs.client.ClientBuilder;  
**import** javax.ws.rs.client.WebTarget;  
**import** javax.ws.rs.core.Response;  
  
**import** org.glassfish.jersey.client.ClientConfig;  
  
**import** com.fasterxml.jackson.jaxrs.json.JacksonJsonProvider;  
**import** twitter.classification.common.exceptions.ProcessingClientException;  
**import** twitter.classification.common.models.SearchResultsResponse;  
**import** twitter.classification.common.models.SuggestedSearchTermsResponse;  
**import** twitter.classification.common.system.ConfigurationVariable;  
**import** twitter.classification.common.system.helper.ConfigurationVariableParam;  
**import** twitter.classification.common.tweetdetails.processing.ProcessResponse;  
  
**public class** AlternativeSearchResultsClient {  
  
 **private** Client **client**;  
 **private** String **uri**;  
  
 @Inject  
 **public** AlternativeSearchResultsClient(@ConfigurationVariableParam(variable = ConfigurationVariable.***SUGGESTED\_SEARCH\_RESULTS\_URI***) String uri) {  
  
 **this**.**uri** = uri;  
 **this**.**client** = ClientBuilder.*newClient*(**new** ClientConfig(**new** JacksonJsonProvider()));  
 }  
  
 */\*\*  
 \* Returns alternative search result terms which the user can search  
 \*  
 \** ***@return*** *suggested search terms  
 \** ***@throws*** *ProcessingClientException when the results cannot be serialised  
 \*/* **public** SuggestedSearchTermsResponse get() **throws** ProcessingClientException {  
  
 Response response;  
  
 **try** {  
  
 WebTarget webTarget = **client**.target(**uri**);  
  
 response = **client**.target(webTarget.getUri())  
 .request()  
 .get();  
 } **catch** (Exception e) {  
  
 **throw new** ProcessingClientException(e);  
 }  
  
 Optional<SuggestedSearchTermsResponse> searchResultsResponseOptional = ProcessResponse.*processResponse*(response, SuggestedSearchTermsResponse.**class**);  
  
 **return** searchResultsResponseOptional.orElseGet(SuggestedSearchTermsResponse::**new**);  
 }  
}

**package** twitter.classification.web.clients;  
  
**import** java.util.Optional;  
  
**import** javax.inject.Inject;  
**import** javax.ws.rs.client.Client;  
**import** javax.ws.rs.client.ClientBuilder;  
**import** javax.ws.rs.client.WebTarget;  
**import** javax.ws.rs.core.Response;  
  
**import** org.glassfish.jersey.client.ClientConfig;  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** com.fasterxml.jackson.jaxrs.json.JacksonJsonProvider;  
**import** twitter.classification.common.exceptions.ProcessingClientException;  
**import** twitter.classification.common.models.DashBoardOverviewResponse;  
**import** twitter.classification.common.system.ConfigurationVariable;  
**import** twitter.classification.common.system.helper.ConfigurationVariableParam;  
**import** twitter.classification.common.tweetdetails.processing.ProcessResponse;  
  
**public class** DashBoardOverviewClient {  
  
 **private static final** Logger ***logger*** = LoggerFactory.*getLogger*(DashBoardOverviewClient.**class**);  
  
 **private** Client **client**;  
 **private** String **uri**;  
  
 @Inject  
 **public** DashBoardOverviewClient(  
 @ConfigurationVariableParam(variable = ConfigurationVariable.***DASHBOARD\_OVERVIEW\_URI***) String uri  
 ) {  
  
 **this**.**uri** = uri;  
 **this**.**client** = ClientBuilder.*newClient*(**new** ClientConfig(JacksonJsonProvider.**class**));  
 }  
  
 */\*\*  
 \* The dashboard overview results for the homepage  
 \*  
 \** ***@return*** *dashboard overview results  
 \** ***@throws*** *ProcessingClientException  
 \*/* **public** Optional<DashBoardOverviewResponse> fetch() **throws** ProcessingClientException {  
  
 Response response;  
  
 **try** {  
  
 WebTarget webTarget = **client**.target(**uri**);  
  
 response = **client**.target(webTarget.getUri())  
 .request()  
 .get();  
 } **catch** (Exception exception) {  
  
 **throw new** ProcessingClientException(exception);  
 }  
  
 **return** ProcessResponse.*processResponse*(response, DashBoardOverviewResponse.**class**);  
 }  
}

**package** twitter.classification.web.clients;  
  
**import** java.util.Optional;  
  
**import** javax.inject.Inject;  
**import** javax.ws.rs.client.Client;  
**import** javax.ws.rs.client.ClientBuilder;  
**import** javax.ws.rs.client.WebTarget;  
**import** javax.ws.rs.core.Response;  
  
**import** org.glassfish.jersey.client.ClientConfig;  
  
**import** com.fasterxml.jackson.jaxrs.json.JacksonJsonProvider;  
**import** twitter.classification.common.exceptions.ProcessingClientException;  
**import** twitter.classification.common.models.DashBoardServiceStatusResponse;  
**import** twitter.classification.common.system.ConfigurationVariable;  
**import** twitter.classification.common.system.helper.ConfigurationVariableParam;  
**import** twitter.classification.common.tweetdetails.processing.ProcessResponse;  
  
**public class** DashBoardServiceStatusClient {  
  
 **private** Client **client**;  
 **private** String **uri**;  
  
 @Inject  
 **public** DashBoardServiceStatusClient(@ConfigurationVariableParam(variable = ConfigurationVariable.***DASHBOARD\_SERVICE\_STATUS\_URI***) String uri) {  
  
 **this**.**uri** = uri;  
 **this**.**client** = ClientBuilder.*newClient*(**new** ClientConfig(**new** JacksonJsonProvider()));  
 }  
  
 */\*\*  
 \* The status of the running services in the system  
 \*  
 \** ***@return*** *dashboard services  
 \** ***@throws*** *ProcessingClientException  
 \*/* **public** DashBoardServiceStatusResponse get() **throws** ProcessingClientException {  
  
 Response response;  
  
 **try** {  
  
 WebTarget webTarget = **client**.target(**uri**);  
  
 response = **client**.target(webTarget.getUri())  
 .request()  
 .get();  
 } **catch** (Exception e) {  
  
 **throw new** ProcessingClientException(e);  
 }  
  
 Optional<DashBoardServiceStatusResponse> dashBoardServiceStatusResponseOptional = ProcessResponse.*processResponse*(response, DashBoardServiceStatusResponse.**class**);  
  
 **return** dashBoardServiceStatusResponseOptional.orElseGet(DashBoardServiceStatusResponse::**new**);  
 }  
}

**package** twitter.classification.web.clients;  
  
**import** java.util.Optional;  
  
**import** javax.inject.Inject;  
**import** javax.ws.rs.client.Client;  
**import** javax.ws.rs.client.ClientBuilder;  
**import** javax.ws.rs.client.WebTarget;  
**import** javax.ws.rs.core.Response;  
  
**import** org.glassfish.jersey.client.ClientConfig;  
  
**import** com.fasterxml.jackson.jaxrs.json.JacksonJsonProvider;  
**import** twitter.classification.common.exceptions.ProcessingClientException;  
**import** twitter.classification.common.models.SearchResultsResponse;  
**import** twitter.classification.common.system.ConfigurationVariable;  
**import** twitter.classification.common.system.helper.ConfigurationVariableParam;  
**import** twitter.classification.common.tweetdetails.processing.ProcessResponse;  
  
**public class** SearchResultsClient {  
  
 **private** Client **client**;  
 **private** String **uri**;  
  
 @Inject  
 **public** SearchResultsClient(@ConfigurationVariableParam(variable = ConfigurationVariable.***SEARCH\_RESULTS\_URI***) String uri) {  
  
 **this**.**uri** = uri;  
 **this**.**client** = ClientBuilder.*newClient*(**new** ClientConfig(**new** JacksonJsonProvider()));  
 }  
  
 */\*\*  
 \* Return the search results for a particular term  
 \*  
 \** ***@param searchTerm*** *\** ***@return*** *search results for the term  
 \** ***@throws*** *ProcessingClientException  
 \*/* **public** SearchResultsResponse get(String searchTerm) **throws** ProcessingClientException {  
  
 Response response;  
  
 **try** {  
  
 WebTarget webTarget = **client**.target(**uri**);  
  
 response = **client**.target(webTarget.getUri())  
 .path(searchTerm)  
 .request()  
 .get();  
 } **catch** (Exception e) {  
  
 **throw new** ProcessingClientException(e);  
 }  
  
 Optional<SearchResultsResponse> searchResultsResponseOptional = ProcessResponse.*processResponse*(response, SearchResultsResponse.**class**);  
  
 **return** searchResultsResponseOptional.orElseGet(SearchResultsResponse::**new**);  
 }  
}

**package** twitter.classification.web.clients;  
  
**import** java.util.Optional;  
  
**import** javax.inject.Inject;  
**import** javax.ws.rs.client.Client;  
**import** javax.ws.rs.client.ClientBuilder;  
**import** javax.ws.rs.client.WebTarget;  
**import** javax.ws.rs.core.Response;  
  
**import** org.glassfish.jersey.client.ClientConfig;  
  
**import** com.fasterxml.jackson.jaxrs.json.JacksonJsonProvider;  
**import** twitter.classification.common.exceptions.ProcessingClientException;  
**import** twitter.classification.common.models.TopHashtagsResponse;  
**import** twitter.classification.common.system.ConfigurationVariable;  
**import** twitter.classification.common.system.helper.ConfigurationVariableParam;  
**import** twitter.classification.common.tweetdetails.processing.ProcessResponse;  
  
**public class** TopHashTagsResultsClient {  
  
 **private** Client **client**;  
 **private** String **uri**;  
  
 @Inject  
 **public** TopHashTagsResultsClient(@ConfigurationVariableParam(variable = ConfigurationVariable.***TOP\_HASHTAGS\_RESULTS\_URI***) String uri) {  
  
 **this**.**uri** = uri;  
 **this**.**client** = ClientBuilder.*newClient*(**new** ClientConfig(**new** JacksonJsonProvider()));  
 }  
  
 */\*\*  
 \* Return the top 10 hashtags based on the amount of classifications associated to it  
 \*  
 \** ***@return*** *top 10 hashtags  
 \** ***@throws*** *ProcessingClientException  
 \*/* **public** TopHashtagsResponse get() **throws** ProcessingClientException {  
  
 Response response;  
  
 **try** {  
  
 WebTarget webTarget = **client**.target(**uri**);  
  
 response = **client**.target(webTarget.getUri())  
 .request()  
 .get();  
 } **catch** (Exception e) {  
  
 **throw new** ProcessingClientException(e);  
 }  
  
 Optional<TopHashtagsResponse> topHashtagsResponseOptional = ProcessResponse.*processResponse*(response, TopHashtagsResponse.**class**);  
  
 **return** topHashtagsResponseOptional.orElseGet(TopHashtagsResponse::**new**);  
 }  
}

**package** twitter.classification.web.clients;  
  
**import** java.util.Optional;  
  
**import** javax.inject.Inject;  
**import** javax.ws.rs.client.Client;  
**import** javax.ws.rs.client.ClientBuilder;  
**import** javax.ws.rs.client.WebTarget;  
**import** javax.ws.rs.core.Response;  
  
**import** org.glassfish.jersey.client.ClientConfig;  
  
**import** com.fasterxml.jackson.jaxrs.json.JacksonJsonProvider;  
**import** twitter.classification.common.exceptions.ProcessingClientException;  
**import** twitter.classification.common.models.TopHashtagsResponse;  
**import** twitter.classification.common.models.TopUsersResponse;  
**import** twitter.classification.common.system.ConfigurationVariable;  
**import** twitter.classification.common.system.helper.ConfigurationVariableParam;  
**import** twitter.classification.common.tweetdetails.processing.ProcessResponse;  
  
**public class** TopUsersResultsClient {  
  
 **private** Client **client**;  
 **private** String **uri**;  
  
 @Inject  
 **public** TopUsersResultsClient(@ConfigurationVariableParam(variable = ConfigurationVariable.***TOP\_USERS\_RESULTS\_URI***) String uri) {  
  
 **this**.**uri** = uri;  
 **this**.**client** = ClientBuilder.*newClient*(**new** ClientConfig(**new** JacksonJsonProvider()));  
 }  
  
 */\*\*  
 \* Return top 10 users results  
 \*  
 \** ***@return*** *top 10 users results  
 \** ***@throws*** *ProcessingClientException  
 \*/* **public** TopUsersResponse get() **throws** ProcessingClientException {  
  
 Response response;  
  
 **try** {  
  
 WebTarget webTarget = **client**.target(**uri**);  
  
 response = **client**.target(webTarget.getUri())  
 .request()  
 .get();  
 } **catch** (Exception e) {  
  
 **throw new** ProcessingClientException(e);  
 }  
  
 Optional<TopUsersResponse> topUsersResponse = ProcessResponse.*processResponse*(response, TopUsersResponse.**class**);  
  
 **return** topUsersResponse.orElseGet(TopUsersResponse::**new**);  
 }  
}

## render

**package** twitter.classification.web.render;  
  
**import** java.io.IOException;  
  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** com.github.jknack.handlebars.Handlebars;  
**import** com.github.jknack.handlebars.Helper;  
**import** com.github.jknack.handlebars.Options;  
**import** com.github.jknack.handlebars.cache.ConcurrentMapTemplateCache;  
**import** com.github.jknack.handlebars.io.ClassPathTemplateLoader;  
**import** com.github.jknack.handlebars.io.TemplateLoader;  
  
**public class** HandleBarsTemplateRender **implements** TemplateRender {  
  
 **private static final** Logger ***logger*** = LoggerFactory.*getLogger*(HandleBarsTemplateRender.**class**);  
  
 **private static final** String ***LIST\_GROUP\_COLOUR*** = **"listGroupColour"**;  
 **private static final** String ***INCREMENT\_INDEX*** = **"increment"**;  
 **private static final** String ***PATH\_IS\_ACTIVE*** = **"pathIsActive"**;  
  
 **private final** Handlebars **handlebars**;  
  
 **public** HandleBarsTemplateRender() {  
  
 TemplateLoader loader = **new** ClassPathTemplateLoader(**"/templates"**, **".hbs"**);  
  
 **handlebars** = **new** Handlebars(loader)  
 .registerHelper(***LIST\_GROUP\_COLOUR***, *listGroupColourHelper*())  
 .registerHelper(***INCREMENT\_INDEX***, *rankIncreaseHelper*())  
 .registerHelper(***PATH\_IS\_ACTIVE***, HandleBarsTemplateRender::*pathEqualsHelper*)  
 .with(**new** ConcurrentMapTemplateCache());  
 }  
  
 */\*\*  
 \* Renders a handlebars template based on the name  
 \*  
 \** ***@param templateName*** *String  
 \** ***@param context*** *Object  
 \** ***@return*** *the constructed html page  
 \*/* @Override  
 **public** String render(String templateName, Object context) {  
  
 **try** {  
  
 **return handlebars**.compile(templateName).apply(context);  
 } **catch** (IOException exception) {  
  
 **throw new** RuntimeException(exception);  
 }  
 }  
  
 */\*\*  
 \* Helper to list the status groups as success (green) or danger (red)  
 \*  
 \** ***@return*** *the list group class for success or danger  
 \*/* **private static** Helper<Object> listGroupColourHelper() {  
  
 **return** ((context, options) -> options.param(0) ? **"list-group-item-success"** : **"list-group-item-danger"**);  
 }  
  
 */\*\*  
 \* Helper which increments the index by 1, as it starts at 0  
 \*  
 \** ***@return*** *index value + 1  
 \*/* **private static** Helper<Object> rankIncreaseHelper() {  
  
 **return** (context, options) -> ((Integer) context + 1);  
 }  
  
 */\*\*  
 \* Helper to highlight which navigation tab  
 \* is highlighted depending on users paths  
 \*  
 \** ***@param context*** *{****@link*** *Object} this is the context of the handlebars page  
 \** ***@param options*** *{****@link*** *Options} Options that are passed to the helper such as current path and comparator  
 \** ***@return*** *{****@link*** *String} either active or nothing to highlight the tabs  
 \*/* **private static** String pathEqualsHelper(Object context, Options options) {  
  
 **if** (options.param(0) == **null**) {  
  
 **return ""**;  
 }  
  
 **if** (options.param(0).equals(options.param(1))) {  
  
 **return "active"**;  
 }  
  
 **return ""**;  
 }  
}

**package** twitter.classification.web.render;  
  
**public interface** TemplateRender {  
  
 */\*\*  
 \* Rendering of a template based on a template name  
 \*  
 \** ***@param templateName*** *String  
 \** ***@param context*** *Object  
 \** ***@return*** *template String  
 \*/* String render(String templateName, Object context);  
}

## resource

**package** twitter.classification.web.resource;  
  
**import** java.util.HashMap;  
**import** java.util.Map;  
**import** java.util.Optional;  
  
**import** javax.inject.Inject;  
**import** javax.inject.Singleton;  
**import** javax.ws.rs.GET;  
**import** javax.ws.rs.Path;  
**import** javax.ws.rs.Produces;  
**import** javax.ws.rs.core.MediaType;  
  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** com.fasterxml.jackson.core.JsonProcessingException;  
**import** com.fasterxml.jackson.databind.ObjectMapper;  
**import** twitter.classification.common.exceptions.ProcessingClientException;  
**import** twitter.classification.common.models.DashBoardOverviewResponse;  
**import** twitter.classification.common.models.DashBoardServiceStatusResponse;  
**import** twitter.classification.common.models.ServiceItem;  
**import** twitter.classification.web.clients.DashBoardOverviewClient;  
**import** twitter.classification.web.clients.DashBoardServiceStatusClient;  
**import** twitter.classification.web.render.TemplateRender;  
  
@Singleton  
@Path(**"/"**)  
**public class** DashBoardResource {  
  
 **private static final** Logger logger = LoggerFactory.getLogger(DashBoardResource.**class**);  
  
 **private static final** String FILTER\_LIST = **"filterList"**;  
 **private static** String TOTAL\_TWEETS = **"totalTweets"**;  
 **private static** String TOTAL\_HASHTAGS = **"totalHashtags"**;  
 **private static** String TOTAL\_RUMOURS = **"totalRumours"**;  
 **private static** String TOTAL\_NON\_RUMOURS = **"totalNonRumours"**;  
 **private static** String TOTAL\_USERNAMES = **"totalUsernames"**;  
 **private static** String TOTAL\_CLASSIFICATIONS = **"totalClassifications"**;  
 **private static** String SERVICES\_LIST = **"serviceList"**;  
  
 **private** TemplateRender templateRender;  
 **private** DashBoardOverviewClient dashBoardOverviewClient;  
 **private** DashBoardServiceStatusClient dashBoardServiceStatusClient;  
  
 @Inject  
 **public** DashBoardResource(  
 TemplateRender templateRender,  
 DashBoardOverviewClient dashBoardOverviewClient,  
 DashBoardServiceStatusClient dashBoardServiceStatusClient  
 ) {  
  
 **this**.templateRender = templateRender;  
 **this**.dashBoardOverviewClient = dashBoardOverviewClient;  
 **this**.dashBoardServiceStatusClient = dashBoardServiceStatusClient;  
 }  
  
 */\*\*  
 \* Returns the dashboard homepage html with the various status and states  
 \* required  
 \*  
 \** ***@return*** *html for the homepage  
 \** ***@throws*** *ProcessingClientException  
 \** ***@throws*** *JsonProcessingException  
 \*/* @GET  
 @Produces(MediaType.TEXT\_HTML)  
 **public** String homePage() **throws** ProcessingClientException, JsonProcessingException {  
  
 Optional<DashBoardOverviewResponse> dashBoardOverviewOptional = dashBoardOverviewClient.fetch();  
  
 Map<String, Object> map = **new** HashMap<>();  
  
 **if** (dashBoardOverviewOptional.isPresent()) {  
  
 DashBoardOverviewResponse dashBoardOverviewResponse = dashBoardOverviewOptional.get();  
  
 map.put(TOTAL\_CLASSIFICATIONS, dashBoardOverviewResponse.getTotalClassifications());  
 map.put(TOTAL\_HASHTAGS, dashBoardOverviewResponse.getTotalHashtags());  
 map.put(TOTAL\_NON\_RUMOURS, dashBoardOverviewResponse.getTotalNonRumours());  
 map.put(TOTAL\_RUMOURS, dashBoardOverviewResponse.getTotalRumours());  
 map.put(TOTAL\_TWEETS, dashBoardOverviewResponse.getTotalTweets());  
 map.put(TOTAL\_USERNAMES, dashBoardOverviewResponse.getTotalUsernames());  
 }  
  
 DashBoardServiceStatusResponse dashBoardServiceStatusResponseOptional = dashBoardServiceStatusClient.get();  
  
 map.put(SERVICES\_LIST, dashBoardServiceStatusResponseOptional.getServiceList());  
  
 **for** (ServiceItem serviceItem : dashBoardServiceStatusResponseOptional.getServiceList()) {  
  
 **if** (serviceItem.getFilterList() != **null** && !serviceItem.getFilterList().isEmpty()) {  
 map.put(FILTER\_LIST, serviceItem.getFilterList().split(**","**));  
 **break**;  
 }  
 }  
  
 **return** templateRender.render(**"dashboard"**, map);  
 }  
}

**package** twitter.classification.web.resource;  
  
**import** java.util.HashMap;  
**import** java.util.Map;  
  
**import** javax.inject.Inject;  
**import** javax.inject.Singleton;  
**import** javax.ws.rs.GET;  
**import** javax.ws.rs.Path;  
**import** javax.ws.rs.Produces;  
**import** javax.ws.rs.core.MediaType;  
  
**import** twitter.classification.common.exceptions.ProcessingClientException;  
**import** twitter.classification.common.models.TopHashtagsResponse;  
**import** twitter.classification.web.clients.TopHashTagsResultsClient;  
**import** twitter.classification.web.render.TemplateRender;  
  
@Singleton  
@Path(**"/hashtags"**)  
**public class** HashtagsResource {  
  
 **private static** String *HASHTAGS\_RESULT\_LIST* = **"hashtagResultsList"**;  
  
 **private** TemplateRender **templateRender**;  
 **private** TopHashTagsResultsClient **topHashTagsResultsClient**;  
  
 @Inject  
 **public** HashtagsResource(  
 TemplateRender templateRender,  
 TopHashTagsResultsClient topHashTagsResultsClient  
 ) {  
  
 **this**.**templateRender** = templateRender;  
 **this**.**topHashTagsResultsClient** = topHashTagsResultsClient;  
 }  
  
 */\*\*  
 \* For returning the HTML for the top hashtags results page  
 \*  
 \** ***@return*** *html for top hashtag results  
 \** ***@throws*** *ProcessingClientException  
 \*/* @GET  
 @Produces(MediaType.***TEXT\_HTML***)  
 **public** String topHashtagsPage() **throws** ProcessingClientException {  
  
 TopHashtagsResponse topHashtagsResponse = **topHashTagsResultsClient**.get();  
  
 Map<String, Object> map = **new** HashMap<>();  
  
 map.put(*HASHTAGS\_RESULT\_LIST*, topHashtagsResponse.getHashtagResultsList());  
  
 **return templateRender**.render(**"hashtags"**, map);  
 }  
}

**package** twitter.classification.web.resource;  
  
**import** java.util.HashMap;  
**import** java.util.Map;  
  
**import** javax.inject.Inject;  
**import** javax.inject.Singleton;  
**import** javax.ws.rs.GET;  
**import** javax.ws.rs.Path;  
**import** javax.ws.rs.PathParam;  
**import** javax.ws.rs.Produces;  
**import** javax.ws.rs.core.MediaType;  
  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** com.fasterxml.jackson.core.JsonProcessingException;  
**import** com.fasterxml.jackson.databind.ObjectMapper;  
**import** twitter.classification.common.exceptions.ProcessingClientException;  
**import** twitter.classification.common.models.SearchResultsResponse;  
**import** twitter.classification.common.models.SuggestedSearchTermsResponse;  
**import** twitter.classification.web.clients.AlternativeSearchResultsClient;  
**import** twitter.classification.web.clients.SearchResultsClient;  
**import** twitter.classification.web.render.TemplateRender;  
  
@Singleton  
@Path(**"/search/{value}"**)  
**public class** SearchResource {  
  
 **private static final** Logger ***logger*** = LoggerFactory.*getLogger*(SearchResource.**class**);  
  
 **private static** String *SEARCH\_TERM\_VALUE* = **"searchTerm"**;  
 **private static** String *COUNT\_OF\_RUMOURS* = **"countOfRumours"**;  
 **private static** String *COUNT\_OF\_NON\_RUMOURS* = **"countOfNonRumours"**;  
 **private static** String *TOTAL\_COUNT\_OF\_CLASSIFICATIONS* = **"totalCountOfClassifications"**;  
 **private static** String *ALTERNATIVE\_SEARCH\_SUGGESTIONS* = **"alternativeSearchSuggestions"**;  
  
 **private** TemplateRender **templateRender**;  
 **private** SearchResultsClient **searchResultsClient**;  
 **private** AlternativeSearchResultsClient **alternativeSearchResultsClient**;  
  
 @Inject  
 **public** SearchResource(  
 TemplateRender templateRender,  
 SearchResultsClient searchResultsClient,  
 AlternativeSearchResultsClient alternativeSearchResultsClient  
 ) {  
  
 **this**.**templateRender** = templateRender;  
 **this**.**searchResultsClient** = searchResultsClient;  
 **this**.**alternativeSearchResultsClient** = alternativeSearchResultsClient;  
 }  
  
 */\*\*  
 \* Search results page if there are results for the particular search term,  
 \* otherwise a no-results page will be displayed with suggested terms  
 \*  
 \** ***@param searchTerm*** *\** ***@return*** *html with results|no results and suggestions  
 \** ***@throws*** *ProcessingClientException  
 \*/* @GET  
 @Produces(MediaType.***TEXT\_HTML***)  
 **public** String get(@PathParam(**"value"**) String searchTerm) **throws** ProcessingClientException {  
  
 SearchResultsResponse searchResultsResponse = **searchResultsClient**.get(searchTerm);  
  
 Map<String, Object> map = **new** HashMap<>();  
  
 *// if there are no results - present user with the no-results page* **if** (searchResultsResponse.getCountOfRumours() == **null** || searchResultsResponse.getCountOfNonRumours() == **null** || searchResultsResponse.getTotalCountOfClassifications() == **null**) {  
  
 SuggestedSearchTermsResponse suggestedSearchTermsResponse = **alternativeSearchResultsClient**.get();  
  
 map.put(*SEARCH\_TERM\_VALUE*, searchTerm);  
 map.put(*ALTERNATIVE\_SEARCH\_SUGGESTIONS*, suggestedSearchTermsResponse.getSearchResultList());  
  
 **return templateRender**.render(**"no-results"**, map);  
 }  
  
 map.put(*SEARCH\_TERM\_VALUE*, searchTerm);  
 map.put(*COUNT\_OF\_RUMOURS*, searchResultsResponse.getCountOfRumours());  
 map.put(*COUNT\_OF\_NON\_RUMOURS*, searchResultsResponse.getCountOfNonRumours());  
 map.put(*TOTAL\_COUNT\_OF\_CLASSIFICATIONS*, searchResultsResponse.getTotalCountOfClassifications());  
  
 **return templateRender**.render(**"search"**, map);  
 }  
}

**package** twitter.classification.web.resource;  
  
**import** java.util.HashMap;  
**import** java.util.Map;  
  
**import** javax.inject.Inject;  
**import** javax.inject.Singleton;  
**import** javax.ws.rs.GET;  
**import** javax.ws.rs.Path;  
**import** javax.ws.rs.Produces;  
**import** javax.ws.rs.core.MediaType;  
  
**import** twitter.classification.common.exceptions.ProcessingClientException;  
**import** twitter.classification.common.models.TopHashtagsResponse;  
**import** twitter.classification.common.models.TopUsersResponse;  
**import** twitter.classification.web.clients.TopHashTagsResultsClient;  
**import** twitter.classification.web.clients.TopUsersResultsClient;  
**import** twitter.classification.web.render.TemplateRender;  
  
@Singleton  
@Path(**"/users"**)  
**public class** UsersResource {  
  
 **private static** String *USERS\_RESULTS\_LIST* = **"userResultsList"**;  
  
 **private** TemplateRender **templateRender**;  
 **private** TopUsersResultsClient **topUsersResultsClient**;  
  
 @Inject  
 **public** UsersResource(  
 TemplateRender templateRender,  
 TopUsersResultsClient topUsersResultsClient  
 ) {  
  
 **this**.**templateRender** = templateRender;  
 **this**.**topUsersResultsClient** = topUsersResultsClient;  
 }  
  
 */\*\*  
 \* Results page for the top hashtag results  
 \*  
 \** ***@return*** *html for the top hashtags results  
 \** ***@throws*** *ProcessingClientException  
 \*/* @GET  
 @Produces(MediaType.***TEXT\_HTML***)  
 **public** String topHashtagsPage() **throws** ProcessingClientException {  
  
 TopUsersResponse topHashtagsResponse = **topUsersResultsClient**.get();  
  
 Map<String, Object> map = **new** HashMap<>();  
  
 map.put(*USERS\_RESULTS\_LIST*, topHashtagsResponse.getUserResultsList());  
  
 **return templateRender**.render(**"users"**, map);  
 }  
}

### exceptions

**package** twitter.classification.web.resource.exceptions;  
  
**import** javax.inject.Inject;  
**import** javax.ws.rs.core.Response;  
**import** javax.ws.rs.ext.ExceptionMapper;  
**import** javax.ws.rs.ext.Provider;  
  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** twitter.classification.web.render.TemplateRender;  
  
**import static** java.util.Collections.*emptyMap*;  
  
*/\*\*  
 \* ExceptionMapper to display custom HTML when no other error handler is thrown and an exception  
 \* is thrown that is not caught in a method/resource  
 \*/*@Provider  
**public class** InternalServerExceptionMapper **implements** ExceptionMapper<Throwable> {  
  
 **private static final** Logger ***logger*** = LoggerFactory.*getLogger*(InternalServerExceptionMapper.**class**);  
  
 **private** TemplateRender **templateRender**;  
  
 @Inject  
 **public** InternalServerExceptionMapper(TemplateRender templateRender) {  
  
 **this**.**templateRender** = templateRender;  
 }  
  
 @Override  
 **public** Response toResponse(Throwable exception) {  
  
 ***logger***.error(**"Encountered an error"**, exception);  
  
 String template = **templateRender**.render(**"exceptions/exception"**, *emptyMap*());  
  
 **return** Response.*status*(Response.Status.***INTERNAL\_SERVER\_ERROR***)  
 .header(**"Content-type"**, **"text/html"**)  
 .entity(template)  
 .build();  
 }  
}

**package** twitter.classification.web.resource.exceptions;  
  
**import** javax.inject.Inject;  
**import** javax.ws.rs.NotFoundException;  
**import** javax.ws.rs.core.Response;  
**import** javax.ws.rs.ext.ExceptionMapper;  
**import** javax.ws.rs.ext.Provider;  
  
**import** twitter.classification.web.render.TemplateRender;  
  
**import static** java.util.Collections.*emptyMap*;  
  
*/\*\*  
 \* ExceptionMapper to display custom HTML when a 404 is encountered  
 \* through no route matching the url or by explicitly throwing {****@link*** *NotFoundException}  
 \* in a method/resource  
 \*/*@Provider  
**public class** NotFoundExceptionResourceMapper **implements** ExceptionMapper<NotFoundException> {  
  
 **private** TemplateRender **render**;  
  
 @Inject  
 **public** NotFoundExceptionResourceMapper(TemplateRender render) {  
  
 **this**.**render** = render;  
 }  
  
 @Override  
 **public** Response toResponse(NotFoundException exception) {  
  
 String template = **render**.render(**"exceptions/not-found"**, *emptyMap*());  
  
 **return** Response.*status*(Response.Status.***NOT\_FOUND***)  
 .header(**"Content-type"**, **"text/html"**)  
 .entity(template)  
 .build();  
 }  
}

# pre-processor/src/main/java/twitter/classification/preprocessor code listings

## application

**package** twitter.classification.preprocessor.application;  
  
**import** org.glassfish.jersey.server.ResourceConfig;  
  
**import** twitter.classification.common.system.binder.ConfigurationVariableBinder;  
**import** twitter.classification.common.system.helper.FileVariables;  
**import** twitter.classification.preprocessor.application.binder.ConfigurationBinder;  
  
**public class** WebApplication **extends** ResourceConfig {  
  
 **public** WebApplication() {  
  
 packages(**"twitter.classification.preprocessor.application"**);  
  
 loadConfigurationValues();  
  
 register(**new** ConfigurationVariableBinder());  
 register(**new** ConfigurationBinder());  
 }  
  
 **private void** loadConfigurationValues() {  
  
 **new** FileVariables().setValuesFromConfigurationFile();  
 }  
}

### binder

**package** twitter.classification.preprocessor.application.binder;  
  
**import** org.glassfish.hk2.utilities.binding.AbstractBinder;  
  
**import** twitter.classification.preprocessor.application.binder.factory.ClassificationClientFactory;  
**import** twitter.classification.preprocessor.client.ClassificationClient;  
  
**public class** ConfigurationBinder **extends** AbstractBinder {  
  
 @Override  
 **protected void** configure() {  
  
 bindFactory(ClassificationClientFactory.**class**).to(ClassificationClient.**class**);  
 }  
}

#### factory

**package** twitter.classification.preprocessor.application.binder.factory;  
  
**import** javax.inject.Inject;  
  
**import** twitter.classification.common.system.ConfigurationVariable;  
**import** twitter.classification.common.system.binder.factory.BaseFactory;  
**import** twitter.classification.common.system.helper.ConfigurationVariableParam;  
**import** twitter.classification.preprocessor.client.ClassificationClient;  
  
**public class** ClassificationClientFactory **implements** BaseFactory<ClassificationClient> {  
  
 **private** String **classifierUri**;  
  
 @Inject  
 **public** ClassificationClientFactory(  
 @ConfigurationVariableParam(variable = ConfigurationVariable.***CLASSIFIER\_CLASSIFICATION\_URI***) String classifierUri  
 ) {  
  
 **this**.**classifierUri** = classifierUri;  
 }  
  
 @Override  
 **public** ClassificationClient provide() {  
  
 **return new** ClassificationClient(**classifierUri**);  
 }  
}

## client

**package** twitter.classification.preprocessor.client;  
  
**import** java.util.Optional;  
  
**import** javax.ws.rs.ProcessingException;  
**import** javax.ws.rs.client.Client;  
**import** javax.ws.rs.client.ClientBuilder;  
**import** javax.ws.rs.client.Entity;  
**import** javax.ws.rs.client.WebTarget;  
**import** javax.ws.rs.core.MediaType;  
**import** javax.ws.rs.core.Response;  
  
**import** org.glassfish.jersey.client.ClientConfig;  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** com.fasterxml.jackson.jaxrs.json.JacksonJsonProvider;  
**import** twitter.classification.common.exceptions.ProcessingClientException;  
**import** twitter.classification.common.tweetdetails.model.ClassificationModel;  
**import** twitter.classification.common.tweetdetails.processing.ProcessResponse;  
  
**public class** ClassificationClient {  
  
 **private static final** Logger ***logger*** = LoggerFactory.*getLogger*(ClassificationClient.**class**);  
  
 **private** Client **client**;  
 **private** String **uri**;  
  
 **public** ClassificationClient(String uri) {  
  
 **this**.**client** = ClientBuilder.*newClient*(**new** ClientConfig(JacksonJsonProvider.**class**));  
 **this**.**uri** = uri;  
 }  
  
 */\*\*  
 \* Method to post the processed item to the classifier for classification purposes  
 \*  
 \** ***@param processedItem*** *\** ***@return*** *{****@link*** *Optional*<*ClassificationModel*>*}  
 \** ***@throws*** *ProcessingClientException  
 \*/* **public** Optional<ClassificationModel> postProcessedTweetItem(String processedItem) **throws** ProcessingClientException {  
  
 Response response;  
  
 **try** {  
  
 WebTarget target = **client**.target(**this**.**uri**);  
  
 response = **client**.target(target.getUri())  
 .request()  
 .post(Entity.*entity*(processedItem, MediaType.***APPLICATION\_JSON***));  
  
 } **catch** (ProcessingException exception) {  
  
 **throw new** ProcessingClientException(exception);  
 }  
  
 **return** ProcessResponse.*processResponse*(response, ClassificationModel.**class**);  
 }  
}

## resource

**package** twitter.classification.preprocessor.resource;  
  
**import** javax.inject.Inject;  
**import** javax.inject.Singleton;  
**import** javax.ws.rs.GET;  
**import** javax.ws.rs.POST;  
**import** javax.ws.rs.Path;  
**import** javax.ws.rs.Produces;  
**import** javax.ws.rs.core.MediaType;  
  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** com.fasterxml.jackson.core.JsonProcessingException;  
**import** com.fasterxml.jackson.databind.ObjectMapper;  
**import** twitter.classification.common.exceptions.ProcessingClientException;  
**import** twitter.classification.common.models.PreProcessorStatusResponse;  
**import** twitter.classification.common.system.ConfigurationVariable;  
**import** twitter.classification.common.system.helper.ConfigurationVariableParam;  
**import** twitter.classification.common.tweetdetails.model.PreProcessedItem;  
**import** twitter.classification.common.tweetdetails.model.ProcessedStatusResponse;  
**import** twitter.classification.common.tweetdetails.processing.TweetBodyProcessor;  
**import** twitter.classification.preprocessor.client.ClassificationClient;  
**import** twitter4j.HashtagEntity;  
**import** twitter4j.Status;  
**import** twitter4j.TwitterException;  
**import** twitter4j.TwitterObjectFactory;  
  
@Singleton  
@Path(**"/process"**)  
**public class** ReceiveTweetStatusResource {  
  
 **private static final** Logger ***logger*** = LoggerFactory.*getLogger*(ReceiveTweetStatusResource.**class**);  
  
 **private** ClassificationClient **classificationClient**;  
 **private boolean usePreProcessing**;  
  
 @Inject  
 **public** ReceiveTweetStatusResource(  
 ClassificationClient classificationClient,  
 @ConfigurationVariableParam(variable = ConfigurationVariable.***USE\_PRE\_PROCESSING***) String usePreProcessing  
 ) {  
  
 **this**.**classificationClient** = classificationClient;  
 **this**.**usePreProcessing** = Boolean.*parseBoolean*(usePreProcessing);  
 }  
  
 @POST  
 @Produces(MediaType.***APPLICATION\_JSON***)  
 **public** ProcessedStatusResponse receiveStatus(String tweetDetails) {  
  
 ***logger***.debug(**"Tweet body is {}"**, tweetDetails);  
  
 ProcessedStatusResponse processedStatusResponse = **new** ProcessedStatusResponse();  
  
 **try** {  
  
 *// converting the JSON to the Twitter4J Status object* Status status = TwitterObjectFactory.createStatus(tweetDetails);  
  
 processedStatusResponse.setTweetBody(status.getText());  
 processedStatusResponse.setUserName(status.getUser().getScreenName());  
 processedStatusResponse.setHashtag(status.getHashtagEntities()[0] != **null** ? status.getHashtagEntities()[0].getText() : **"NO-HASHTAG"**);  
  
 *// preparing the preprocessed item which will be sent for classification* PreProcessedItem preProcessedItem = **new** PreProcessedItem();  
  
 preProcessedItem.setUsername(status.getUser().getScreenName());  
 preProcessedItem.setTweetId(status.getId());  
 preProcessedItem.setUserId(status.getUser().getId());  
 preProcessedItem.setOriginalTweetBody(status.getText());  
  
 *// this is where the pre processing will occur if the configuration value is set* **if** (usePreProcessing) {  
 preProcessedItem.setProcessedTweetBody(**new** TweetBodyProcessor().processTweetBody(status.getText()));  
 } **else** {  
 preProcessedItem.setProcessedTweetBody(status.getText());  
 }  
  
 logger.debug(**"Tweet body is: {}"**, preProcessedItem.getProcessedTweetBody());  
  
 **for** (HashtagEntity hashtagEntity : status.getHashtagEntities()) {  
  
 *// all hashtags will be kept in lowercase format* preProcessedItem.addHashtag(hashtagEntity.getText().toLowerCase());  
 }  
  
 classificationClient.postProcessedTweetItem(**new** ObjectMapper().writeValueAsString(preProcessedItem));  
  
 } **catch** (TwitterException e) {  
 logger.error(**"Issue creating status from tweet details"**, e);  
 } **catch** (ProcessingClientException | JsonProcessingException e) {  
 logger.error(**"Issue processing object"**, e);  
 }  
  
 **return** processedStatusResponse;  
 }  
  
 @GET  
 @Produces(MediaType.APPLICATION\_JSON)  
 @Path(**"/status"**)  
 **public** PreProcessorStatusResponse getPreProcessorStatus() {  
  
 **return new** PreProcessorStatusResponse().setRunning(**true**);  
 }  
}

# queue-reader/src/main/java/queuereader code listings

## application

**package** twitter.classification.queuereader.application;  
  
**import** com.google.inject.Guice;  
**import** com.google.inject.Injector;  
**import** twitter.classification.common.system.helper.FileVariables;  
**import** twitter.classification.queuereader.module.ConfigurationModule;  
**import** twitter.classification.queuereader.reader.QueueReader;  
  
**public class** QueueReaderApplication {  
  
 **public static void** main(String[] args) {  
  
 **new** QueueReaderApplication().loadConfigurationValues();  
  
 Injector injector = Guice.*createInjector*(  
 **new** ConfigurationModule()  
 );  
  
 injector.getInstance(QueueReader.**class**).run();  
 }  
  
 **private void** loadConfigurationValues() {  
  
 **new** FileVariables().setValuesFromConfigurationFile();  
 }  
}

### exceptions

**package** twitter.classification.queuereader.application.exceptions;  
  
**public class** IgnoredHashtagEntity **extends** Exception {  
  
 **public** IgnoredHashtagEntity(String message) {  
  
 **super**(message);  
 }  
}

## consumer

**package** twitter.classification.queuereader.consumer;  
  
**import** java.io.IOException;  
**import** java.util.Optional;  
  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** com.fasterxml.jackson.databind.ObjectMapper;  
**import** com.rabbitmq.client.AMQP;  
**import** com.rabbitmq.client.Channel;  
**import** com.rabbitmq.client.DefaultConsumer;  
**import** com.rabbitmq.client.Envelope;  
**import** twitter.classification.common.tweetdetails.model.ProcessedStatusResponse;  
**import** twitter.classification.queuereader.application.exceptions.IgnoredHashtagEntity;  
**import** twitter.classification.queuereader.tweetdetails.TweetDetailsClient;  
**import** twitter4j.HashtagEntity;  
**import** twitter4j.Status;  
**import** twitter4j.TwitterObjectFactory;  
  
**public class** TweetConsumer **extends** DefaultConsumer {  
  
 **private static final** Logger ***logger*** = LoggerFactory.*getLogger*(TweetConsumer.**class**);  
  
 **private** TweetDetailsClient **client**;  
 **private** String[] **hashtagIgnoreList**;  
  
 **public** TweetConsumer(Channel channel, TweetDetailsClient client, String hashtagIgnoreList) {  
  
 **super**(channel);  
  
 **this**.**client** = client;  
  
 **this**.**hashtagIgnoreList** = hashtagIgnoreList.split(**","**);  
 }  
  
 @Override  
 **public void** handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties, **byte**[] body)  
 **throws** IOException {  
  
 String message = **new** String(body, **"UTF-8"**);  
  
 **try** {  
  
 ***logger***.debug(**"Handling message with body of {}"**, message);  
  
 Status status = TwitterObjectFactory.*createStatus*(message);  
  
 **for** (HashtagEntity hashtagEntity : status.getHashtagEntities()) {  
 **for**(String hashtagIgnore : **hashtagIgnoreList**) {  
 **if** (hashtagEntity.getText().toLowerCase().equals(hashtagIgnore)) {  
 **throw new** IgnoredHashtagEntity(String.*format*(**"Hashtag %s is in the ignore list"**, hashtagEntity.getText().toLowerCase()));  
 }  
 }  
 }  
  
 Optional<ProcessedStatusResponse> response = **client**.postStatusForProcessing(message);  
  
 **if** (response.isPresent())  
 ***logger***.debug(**"Response handled correctly: {}"**, **new** ObjectMapper().writeValueAsString(response.get()));  
  
 } **catch** (IgnoredHashtagEntity e) {  
 ***logger***.error(e.getMessage());  
 } **catch** (Exception e) {  
 ***logger***.error(**"Issue handling message"**, e);  
 }  
 }  
}

## module

**package** twitter.classification.queuereader.module;  
  
**import** java.io.IOException;  
**import** java.util.concurrent.TimeoutException;  
  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** com.google.inject.AbstractModule;  
**import** com.google.inject.Provides;  
**import** com.google.inject.name.Named;  
**import** com.google.inject.name.Names;  
**import** com.rabbitmq.client.Channel;  
**import** com.rabbitmq.client.Connection;  
**import** com.rabbitmq.client.ConnectionFactory;  
**import** twitter.classification.common.system.helper.FileVariables;  
**import** twitter.classification.queuereader.consumer.TweetConsumer;  
**import** twitter.classification.queuereader.reader.QueueReader;  
**import** twitter.classification.queuereader.tweetdetails.TweetDetailsClient;  
  
**public class** ConfigurationModule **extends** AbstractModule {  
  
 **private static final** Logger ***logger*** = LoggerFactory.*getLogger*(ConfigurationModule.**class**);  
  
 @Override  
 **protected void** configure() {  
  
 Names.*bindProperties*(binder(), FileVariables.*properties*);  
 }  
  
 @Provides  
 **public** QueueReader provideQueueReader(  
 @Named(**"QUEUE\_USER"**) String queueUsername,  
 @Named(**"QUEUE\_PASSWORD"**) String queuePassword,  
 @Named(**"QUEUE\_HOST"**) String queueHost,  
 @Named(**"QUEUE\_URI"**) String queueUri,  
 @Named(**"HASHTAG\_IGNORE\_LIST"**) String hashtagIgnoreList) **throws** IOException, TimeoutException {  
  
 ConnectionFactory connectionFactory = **new** ConnectionFactory();  
 connectionFactory.setUsername(queueUsername);  
 connectionFactory.setPassword(queuePassword);  
 connectionFactory.setHost(queueHost);  
  
 Connection connection = connectionFactory.newConnection();  
  
 Channel channel = connection.createChannel();  
  
 channel.queueDeclare(**"tweets"**, **false**, **false**, **false**, **null**);  
  
 **return new** QueueReader(channel, **new** TweetConsumer(channel, **new** TweetDetailsClient(queueUri), hashtagIgnoreList));  
 }  
}

## reader

**package** twitter.classification.queuereader.reader;  
  
**import** java.io.IOException;  
  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** com.rabbitmq.client.Channel;  
**import** com.rabbitmq.client.Consumer;  
  
**public class** QueueReader {  
  
 **private static final** Logger ***logger*** = LoggerFactory.*getLogger*(QueueReader.**class**);  
  
 **private** Channel **channel**;  
  
 **private** Consumer **consumer**;  
  
 **public** QueueReader(Channel channel, Consumer consumer) {  
  
 **this**.**channel** = channel;  
 **this**.**consumer** = consumer;  
 }  
  
 **public void** run() {  
  
 **try** {  
  
 **channel**.basicConsume(**"tweets"**, **true**, **consumer**);  
 } **catch** (IOException e) {  
 ***logger***.error(**"Issue consuming the queue"**, e);  
 }  
 }  
}

## tweetdetails

**package** twitter.classification.queuereader.tweetdetails;  
  
**import** java.util.Optional;  
  
**import** javax.ws.rs.ProcessingException;  
**import** javax.ws.rs.client.Client;  
**import** javax.ws.rs.client.ClientBuilder;  
**import** javax.ws.rs.client.Entity;  
**import** javax.ws.rs.client.WebTarget;  
**import** javax.ws.rs.core.MediaType;  
**import** javax.ws.rs.core.Response;  
  
**import** org.glassfish.jersey.client.ClientConfig;  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** com.fasterxml.jackson.jaxrs.json.JacksonJsonProvider;  
**import** twitter.classification.common.exceptions.ProcessingClientException;  
**import** twitter.classification.common.tweetdetails.model.ProcessedStatusResponse;  
**import** twitter.classification.common.tweetdetails.processing.ProcessResponse;  
  
**public class** TweetDetailsClient {  
  
 **private static final** Logger ***logger*** = LoggerFactory.*getLogger*(TweetDetailsClient.**class**);  
  
 **private** Client **client**;  
 **private** String **uri**;  
  
 **public** TweetDetailsClient(String uri) {  
  
 **this**.**client** = ClientBuilder.*newClient*(**new** ClientConfig(JacksonJsonProvider.**class**));  
 **this**.**uri** = uri;  
 }  
  
 **public** Optional<ProcessedStatusResponse> postStatusForProcessing(String status) **throws** ProcessingClientException {  
  
 Response response;  
  
 **try** {  
 WebTarget target = **client**.target(**uri**);  
  
 response = **client**.target(target.getUri())  
 .request()  
 .post(Entity.*entity*(status, MediaType.***APPLICATION\_JSON***));  
  
 } **catch** (ProcessingException exception) {  
  
 **throw new** ProcessingClientException(exception);  
 }  
  
 **return** ProcessResponse.*processResponse*(response, ProcessedStatusResponse.**class**);  
 }  
}

# stream/src/main/java/twitter/classification/stream code listings

## application

**package** twitter.classification.stream.application;  
  
**import** org.glassfish.jersey.server.ResourceConfig;  
  
**import** twitter.classification.common.system.binder.ConfigurationVariableBinder;  
**import** twitter.classification.common.system.helper.FileVariables;  
**import** twitter.classification.stream.application.binder.MessageQueueBinder;  
**import** twitter.classification.stream.application.binder.TwitterStreamBinder;  
  
**public class** WebApplication **extends** ResourceConfig {  
  
 **public** WebApplication() {  
  
 packages(**"twitter.classification.stream.application"**);  
  
 loadConfigurationValues();  
 register(**new** ConfigurationVariableBinder());  
 register(**new** TwitterStreamBinder());  
 register(**new** MessageQueueBinder());  
 }  
  
 **private void** loadConfigurationValues() {  
  
 **new** FileVariables().setValuesFromConfigurationFile();  
 }  
}

### binder

**package** twitter.classification.stream.application.binder;  
  
**import** org.glassfish.hk2.utilities.binding.AbstractBinder;  
  
**import** twitter.classification.stream.application.binder.factory.TwitterStreamFactory;  
**import** twitter4j.TwitterStream;  
  
**public class** TwitterStreamBinder **extends** AbstractBinder {  
  
 @Override  
 **protected void** configure() {  
  
 bindFactory(TwitterStreamFactory.**class**).to(TwitterStream.**class**);  
 }  
}

**package** twitter.classification.stream.application.binder;  
  
**import** org.glassfish.hk2.utilities.binding.AbstractBinder;  
  
**import** com.rabbitmq.client.Connection;  
**import** twitter.classification.stream.application.binder.factory.MessageQueueFactory;  
  
**public class** MessageQueueBinder **extends** AbstractBinder {  
  
 @Override  
 **protected void** configure() {  
  
 bindFactory(MessageQueueFactory.**class**).to(Connection.**class**);  
 }  
}

#### factory

**package** twitter.classification.stream.application.binder.factory;  
  
**import** java.io.IOException;  
**import** java.util.concurrent.TimeoutException;  
  
**import** javax.inject.Inject;  
  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** com.rabbitmq.client.Connection;  
**import** com.rabbitmq.client.ConnectionFactory;  
**import** twitter.classification.common.system.ConfigurationVariable;  
**import** twitter.classification.common.system.binder.factory.BaseFactory;  
**import** twitter.classification.common.system.helper.ConfigurationVariableParam;  
  
**public class** MessageQueueFactory **implements** BaseFactory<Connection> {  
  
 **private static final** Logger ***logger*** = LoggerFactory.*getLogger*(MessageQueueFactory.**class**);  
  
 **private** Connection **connection**;  
  
 @Inject  
 **public** MessageQueueFactory(  
 @ConfigurationVariableParam(variable = ConfigurationVariable.***QUEUE\_HOST***) String queueHost,  
 @ConfigurationVariableParam(variable = ConfigurationVariable.***QUEUE\_USER***) String queueUser,  
 @ConfigurationVariableParam(variable = ConfigurationVariable.***QUEUE\_PASSWORD***) String queuePassword  
 ) {  
  
 ConnectionFactory factory = **new** ConnectionFactory();  
 factory.setHost(queueHost);  
 factory.setUsername(queueUser);  
 factory.setPassword(queuePassword);  
  
 **try** {  
  
 **connection** = factory.newConnection();  
 } **catch** (IOException | TimeoutException e) {  
  
 ***logger***.error(**"Issue creating queue"**, e);  
 }  
 }  
  
 @Override  
 **public** Connection provide() {  
  
 **return connection**;  
 }  
}

**package** twitter.classification.stream.application.binder.factory;  
  
**import** javax.inject.Inject;  
  
**import** twitter.classification.common.system.ConfigurationVariable;  
**import** twitter.classification.common.system.binder.factory.BaseFactory;  
**import** twitter.classification.common.system.helper.ConfigurationVariableParam;  
**import** twitter4j.TwitterStream;  
**import** twitter4j.conf.ConfigurationBuilder;  
  
**public class** TwitterStreamFactory **implements** BaseFactory<TwitterStream> {  
  
 **private final** TwitterStream **twitterStream**;  
  
 @Inject  
 **public** TwitterStreamFactory(  
 @ConfigurationVariableParam(variable = ConfigurationVariable.***TWITTER\_OAUTH\_ACCESS\_KEY***) String oauthAccessKey,  
 @ConfigurationVariableParam(variable = ConfigurationVariable.***TWITTER\_OAUTH\_ACCESS\_SECRET***) String oauthAccessSecret,  
 @ConfigurationVariableParam(variable = ConfigurationVariable.***TWITTER\_OAUTH\_CONSUMER\_KEY***) String oauthConsumerKey,  
 @ConfigurationVariableParam(variable = ConfigurationVariable.***TWITTER\_OAUTH\_CONSUMER\_SECRET***) String oauthConsumerSecret  
 ) {  
  
 **twitterStream** = **new** twitter4j.TwitterStreamFactory(**new** ConfigurationBuilder()  
 .setTweetModeExtended(**true**)  
 .setOAuthConsumerKey(oauthConsumerKey)  
 .setOAuthConsumerSecret(oauthConsumerSecret)  
 .setOAuthAccessToken(oauthAccessKey)  
 .setOAuthAccessTokenSecret(oauthAccessSecret)  
 .setJSONStoreEnabled(**true**)  
 .build()  
 ).getInstance();  
 }  
  
 @Override  
 **public** TwitterStream provide() {  
  
 **return twitterStream**;  
 }  
}

## listener

**package** twitter.classification.stream.listener;  
  
**import** java.io.IOException;  
  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** com.google.gson.JsonNull;  
**import** com.google.gson.JsonObject;  
**import** com.google.gson.JsonParser;  
**import** twitter4j.StallWarning;  
**import** twitter4j.Status;  
**import** twitter4j.StatusDeletionNotice;  
**import** twitter4j.StatusListener;  
**import** twitter4j.TwitterObjectFactory;  
  
**import static** twitter.classification.stream.resource.StreamTweetsResource.*channel*;  
  
**public class** NewTweetListener **implements** StatusListener {  
  
 **private final static** Logger ***logger*** = LoggerFactory.*getLogger*(NewTweetListener.**class**);  
  
 @Override  
 **public void** onStatus(Status status) {  
  
 JsonObject jsonObject = **new** JsonParser().parse(TwitterObjectFactory.*getRawJSON*(status)).getAsJsonObject();  
  
 **if** (!status.isRetweet() && jsonObject.get(**"in\_reply\_to\_status\_id\_str"**).getAsJsonNull() == JsonNull.***INSTANCE*** && status.getHashtagEntities().**length** > 0) {  
  
 **try** {  
  
 *channel*.basicPublish(**""**, **"tweets"**, **null**, TwitterObjectFactory.*getRawJSON*(status).getBytes());  
 } **catch** (IOException e) {  
  
 ***logger***.error(**"Issue adding to queue"**, e);  
 }  
 }  
 }  
  
 @Override  
 **public void** onDeletionNotice(StatusDeletionNotice statusDeletionNotice) {  
 }  
  
 @Override  
 **public void** onTrackLimitationNotice(**int** numberOfLimitedStatuses) {  
 }  
  
 @Override  
 **public void** onScrubGeo(**long** userId, **long** upToStatusId) {  
 }  
  
 @Override  
 **public void** onStallWarning(StallWarning warning) {  
 }  
  
 @Override  
 **public void** onException(Exception ex) {  
 }  
}

## resource

**package** twitter.classification.stream.resource;  
  
**import** javax.inject.Inject;  
**import** javax.inject.Singleton;  
**import** javax.ws.rs.GET;  
**import** javax.ws.rs.Path;  
**import** javax.ws.rs.Produces;  
**import** javax.ws.rs.core.MediaType;  
  
**import** org.slf4j.Logger;  
**import** org.slf4j.LoggerFactory;  
  
**import** com.rabbitmq.client.Channel;  
**import** com.rabbitmq.client.Connection;  
**import** twitter.classification.common.models.TwitterStreamResponse;  
**import** twitter.classification.common.system.ConfigurationVariable;  
**import** twitter.classification.common.system.helper.ConfigurationVariableParam;  
**import** twitter.classification.stream.listener.NewTweetListener;  
**import** twitter4j.FilterQuery;  
**import** twitter4j.TwitterStream;  
  
@Singleton  
@Path(**"/stream"**)  
**public class** StreamTweetsResource {  
  
 **private static final** Logger logger = LoggerFactory.getLogger(StreamTweetsResource.**class**);  
  
 **private static boolean** isRunning = **false**;  
  
 **private** TwitterStream twitterStream;  
 **private** String filterList;  
  
 **public static** Channel channel;  
  
 @Inject  
 **public** StreamTweetsResource(  
 @ConfigurationVariableParam(variable = ConfigurationVariable.QUEUE\_NAME) String queueName,  
 @ConfigurationVariableParam(variable = ConfigurationVariable.TWITTER\_FILTER\_LIST) String filterList,  
 TwitterStream twitterStream,  
 Connection connection  
 ) {  
  
 **this**.twitterStream = twitterStream;  
  
 **try** {  
  
 channel = connection.createChannel();  
  
 channel.queueDeclare(queueName, **false**, **false**, **false**, **null**);  
 } **catch** (Exception e) {  
  
 logger.error(**"Issue creating queue"**, e);  
 }  
  
 twitterStream.addListener(**new** NewTweetListener());  
  
 **this**.filterList = filterList;  
 }  
  
 @GET  
 @Path(**"/start"**)  
 @Produces(MediaType.APPLICATION\_JSON)  
 **public** TwitterStreamResponse startStream() {  
  
 **if** (!isRunning) {  
 String[] filter = filterList.split(**","**);  
  
 twitterStream.filter(**new** FilterQuery(filter).language(**"en"**));  
  
 isRunning = **true**;  
  
 logger.info(**"Running with filter list: {}"**, filterList);  
  
 **return new** TwitterStreamResponse().setRunning(isRunning).setFilterList(filterList);  
  
 } **else** {  
  
 **return new** TwitterStreamResponse().setRunning(isRunning).setFilterList(filterList);  
 }  
 }  
  
 @GET  
 @Path(**"/stop"**)  
 @Produces(MediaType.APPLICATION\_JSON)  
 **public** TwitterStreamResponse stopStream() {  
  
 twitterStream.shutdown();  
  
 isRunning = **false**;  
  
 **return new** TwitterStreamResponse().setRunning(isRunning).setFilterList(filterList);  
 }  
  
 @GET  
 @Path(**"/running"**)  
 @Produces(MediaType.APPLICATION\_JSON)  
 **public** TwitterStreamResponse isRunning() {  
  
 **return new** TwitterStreamResponse().setRunning(isRunning).setFilterList(filterList);  
 }  
}

# HTML and JS files

## master.hbs

<!doctype **html**>  
<**html lang="en"**>  
<**head**>  
 *<!-- Required meta tags -->* <**meta charset="utf-8"**>  
 <**meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no"**>  
  
 *<!-- Bootstrap CSS -->* <**link rel="stylesheet" href="/assets/css/bootstrap/bootstrap.min.css" type="text/css"**>  
 <**link rel="stylesheet" href="/assets/css/bootstrap/bootstrap-tabs-x.min.css" type="text/css"**>  
 <**link rel="stylesheet" href="/assets/css/bootstrap/footer.css" type="text/css"**>  
  
 <**title**>Tweet Classification</**title**>  
  
 <**style**>  
 .**filter a** {  
 **text-decoration**: **none**;  
 **color**: **#fff**;  
 **display**: **inline-block**;  
 **padding**: 5**px** 20**px**;  
 **margin**: 1**px**;  
 **border-radius**:4**px**;  
 **margin-top**: 6**px**;  
 **background-color**: **black**;  
 **border**: **solid** 1**px #fff**;  
 }  
  
 .**chart-container** {  
 **position**: **relative**;  
 **height**: 40**vh**;  
 **width**: 40**vw**;  
 **margin**: 20**px auto**;  
 }  
 </**style**>  
</**head**>  
<**body**>  
  
*<!-- Optional JavaScript -->  
<!-- jQuery first, then Popper.js, then Bootstrap JS -->*<**script src="/assets/js/jquery/jquery-3.3.1.min.js" type="text/javascript"**></**script**>  
<**script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.12.9/umd/popper.min.js" type="text/javascript"**></**script**>  
<**script src="/assets/js/bootstrap/bootstrap.min.js" type="text/javascript"**></**script**>  
<**script src="/assets/js/bootstrap/bootstrap-tabs-x.min.js" type="text/javascript"**></**script**>  
<**script src="/assets/js/bootstrap/jquery.twbsPagination.min.js" type="text/javascript"**></**script**>  
<**script src="/assets/js/chart/Chart.bundle.min.js" type="text/javascript"**></**script**>  
  
<**script src="/assets/js/navigation.js" type="text/javascript"**></**script**>  
  
*<!--  
 Navbar will be consistent throughout  
 -->*<**nav class="navbar sticky-top navbar-expand-lg navbar-dark bg-dark navbar-fixed"**>  
 <**div class="container"**>  
 <**a class="navbar-brand" href="/"**>Tweet Classification</**a**>  
 <**button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbar" aria-controls="navbar"  
 aria-expanded="false" aria-label="Toggle navigation"**>  
 <**span class="navbar-toggler-icon"**></**span**>  
 </**button**>  
  
 <**div class="collapse navbar-collapse" id="navbar"**>  
 <**ul class="navbar-nav ml-auto"**>  
 <**form class="form-inline my-2 my-lg-0"**>  
 <**input class="form-control mr-sm-2" type="text" placeholder="Search hashtags/users" id="searchTermValue"**>  
 <**button class="btn btn-outline-success my-2 my-sm-0" type="submit" id="searchTerm"**>Search</**button**>  
 </**form**>  
 </**ul**>  
 <**ul class="navbar-nav ml-auto"**>  
 <**li class="nav-item"**><**a class="nav-link {{pathIsActive this path 'dashboard'}}" href="/"**>Dashboard</**a**>  
 </**li**>  
 <**li class="nav-item"**><**a class="nav-link {{pathIsActive this path 'users'}}" href="/users"**>Users</**a**></**li**>  
 <**li class="nav-item"**><**a class="nav-link {{pathIsActive this path 'hashtags'}}"  
 href="/hashtags"**>Hashtags</**a**></**li**>  
 <**li class="nav-item dropdown"**>  
 <**a class="nav-link dropdown-toggle" id="navDropDown" data-toggle="dropdown" aria-haspopup="true"  
 aria-expanded="false"**>More</**a**>  
 <**div class="dropdown-menu dropdown-menu-right" aria-labelledby="navDropDown"**>  
 <**a class="dropdown-item" href="/terms"**>Terms of Service</**a**>  
 </**div**>  
 </**li**>  
 </**ul**>  
 </**div**>  
 </**div**>  
</**nav**>  
  
<**section class="messages"**></**section**>  
  
<**div class="content"**>  
<**div class="container" style="margin-top**: 50**px**; **margin-bottom**: 50**px**;**"**>  
 <**main**>  
 {{#block "content"}}{{/block}}  
 </**main**>  
</**div**>  
</**div**>  
  
<**footer id="myFooter"**>  
 <**div class="container"**>  
 <**div class="row"**>  
 <**div class="col-sm-3"**>  
 <**h5**>About</**h5**>  
 <**p class="text-light"**>A Final Year Project for BEng in Software Engineering for Ulster University.</**p**>  
 </**div**>  
 <**div class="col-sm-6"**>  
 <**div class="text-center" style="margin-top**: 20**px**;**"**>  
 <**img src="https://www.aspenpeople.co.uk/UU/images/logo\_white.png" style="width**: 300**px**; **height**: 100**px**;**"**>  
 </**div**>  
 </**div**>  
 <**div class="col-sm-3 info"**>  
 <**h5**>Contact</**h5**>  
 <**p class="text-light"**>Thomas Franklin</**p**>  
 <**p class="text-light"**>franklin-t@ulster.ac.uk</**p**>  
 </**div**>  
 </**div**>  
 </**div**>  
 <**div class="second-bar"**>  
 <**div class="container"**>  
 <**h2 class="logo"**><**a href="/"**>Tweet Classification</**a**></**h2**>  
 <**div class="footer-links"**>  
 <**a href="/users"**>Users</**a**>  
 <**a href="/hashtags"**>Hashtags</**a**>  
 </**div**>  
 </**div**>  
 </**div**>  
</**footer**>  
</**body**>  
</**html**>

## no-result.hbs

{{#partial "content"}}  
<**div class="row"**>  
 <**div class="col-md-12 mb-1"**>  
 <**div class="visualisation-boards"**>  
 <**h1**>No results found for: <**span class="search-term"**>{{searchTerm}}</**span**></**h1**>  
  
 <**p**>The search function only allows to search for particular Twitter users, using their usernames or  
 particular hashtags that have been streamed in to the service with the current filter.</**p**>  
  
 <**p**>If you would like, you could preform a new search for one of the suggested terms below:</**p**>  
 <**ul**>  
 {{#each alternativeSearchSuggestions}}  
 <**li**>{{this.value}}</**li**>  
 {{/each}}  
 </**ul**>  
  
 </**div**>  
 </**div**>  
</**div**>  
  
{{/partial}}  
{{> master path="search"}}

## search.hbs

{{#partial "content"}}  
<**div class="row"**>  
 <**div class="col-md-8 mb-1"**>  
 <**div class="visualisation-boards"**>  
 <**h1**>Search results for: <**span class="search-term"**>{{searchTerm}}</**span**></**h1**>  
 <**div id="searchTerm" class="container" data-rumour="{{countOfRumours}}" data-non-rumour="{{countOfNonRumours}}"**><**br**>  
 {{> partials/visualisation-board value="searchTerm" }}  
 </**div**>  
 </**div**>  
 </**div**>  
 <**div class="col-md-12 mb-1 mt-1"**>  
 <**div class="raw-row-data"**>  
 <**div id="searchTermRawData" class="container"  
 data-size="{{totalCountOfClassifications}}"**><**br**>  
 <**div class="card"**>  
 <**div class="card-body"**>  
 <**h5 class="card-title text-center"**>Raw Data for {{searchTerm}}</**h5**>  
 <**hr**>  
 <**div class="total-results-tip justify-content-center text-right" id="searchTermPaginationTip"**>  
 <**p**>Total results: {{totalCountOfClassifications}}</**p**>  
 </**div**>  
 <**div id="searchTermTable"**></**div**>  
 <**div class="justify-content-center"**>  
 <**nav**>  
 <**ul class="pagination justify-content-center" id="searchTermPagination"**></**ul**>  
 </**nav**>  
 </**div**>  
 </**div**>  
 </**div**>  
 </**div**>  
 </**div**>  
 </**div**>  
</**div**>  
  
<**script src="/assets/js/search.js" type="text/javascript"**></**script**>  
  
{{/partial}}  
{{> master path="search"}}

## users.hbs

{{#partial "content"}}  
  
<**div class="row"**>  
 <**div class="col-md-8 mb-1"**>  
 <**div class="visualisation-boards"**>  
 <**ul class="nav nav-tabs mr-auto d-none" role="tablist"**>  
 <**li class="nav-item"**>  
 <**a class="nav-link" role="tab" data-toggle="tab"  
 href="#default"**>Default Tab</**a**>  
 </**li**>  
 {{#each userResultsList}}  
 <**li class="nav-item"**>  
 <**a class="nav-link" role="tab" data-toggle="tab"  
 href="#{{this.username}}"**>{{this.username}}</**a**>  
 </**li**>  
 {{/each}}  
 </**ul**>  
 <**div class="tab-content"**>  
 <**div id="default" class="container tab-pane active"**><**br**>  
 <**h5**>Top Users</**h5**>  
 <**p**>The users classifications will be displayed using various techniques, such as word clouds,  
 charts etc.</**p**>  
 <**p**>In the table on the right, please select the one you wish to view.</**p**>  
 </**div**>  
 {{#each userResultsList}}  
 <**div id="{{this.username}}" class="container tab-pane fade" data-rumour="{{this.countOfRumours}}" data-non-rumour="{{this.countOfNonRumours}}"**><**br**>  
 {{> partials/visualisation-board value=this.username }}  
 </**div**>  
 {{/each}}  
 </**div**>  
 </**div**>  
 </**div**>  
 <**div class="col-md-4"**>  
 <**div class="users-table"**>  
 <**div class="card"**>  
 <**div class="card-body"**>  
 <**h5 class="card-title text-center"**>Users</**h5**>  
 <**hr**>  
 <**table class="table table-striped"**>  
 <**thead**>  
 <**tr**>  
 <**th scope="col"**>Rank</**th**>  
 <**th scope="col"**>Username</**th**>  
 <**th scope="col"**></**th**>  
 </**tr**>  
 </**thead**>  
 <**tbody**>  
 {{#each userResultsList}}  
 <**tr**>  
 <**th scope="row"**>{{increment @index}}</**th**>  
 <**td**>  
 <**div class="radiotext"**>  
 <**label for="{{this.username}}RadioInput"**>{{this.username}}</**label**>  
 </**div**>  
 </**td**>  
 <**td**>  
 <**div class="radio"**>  
 <**label**>  
 <**input type="radio" id="{{this.username}}RadioInput"  
 name="userRadioGroup"  
 data-target="#{{this.username}}"**>  
 </**label**>  
 </**div**>  
 </**td**>  
 </**tr**>  
 {{/each}}  
 </**tbody**>  
 </**table**>  
 </**div**>  
 </**div**>  
 </**div**>  
 </**div**>  
 <**div class="col-md-12 mb-1 mt-1"**>  
 <**div class="raw-row-data"**>  
 <**ul class="nav nav-tabs mr-auto d-none" role="tablist"**>  
 <**li class="nav-item"**>  
 <**a class="nav-link" role="tab" data-toggle="tab" href="#none"**>Default Tab</**a**>  
 </**li**>  
 {{#each userResultsList}}  
 <**li class="nav-item"**>  
 <**a class="nav-link" role="tab" data-toggle="tab"  
 href="#{{this.username}}RawData"**>{{this.username}} Raw Data</**a**>  
 </**li**>  
 {{/each}}  
 </**ul**>  
 <**div class="tab-content"**>  
 <**div id="none" class="container tab-pane active"**><**br**>  
 </**div**>  
  
 {{#each userResultsList}}  
 <**div id="{{this.username}}RawData" class="container tab-pane fade"  
 data-size="{{this.totalCountOfClassifications}}"**><**br**>  
 <**div class="card"**>  
 <**div class="card-body"**>  
 <**h5 class="card-title text-center"**>Raw Data for {{this.username}}</**h5**>  
 <**hr**>  
 <**div class="total-results-tip justify-content-center text-right" id="{{this.username}}PaginationTip"**>  
 <**p**>Total results: {{this.totalCountOfClassifications}}</**p**>  
 </**div**>  
 <**div id="{{this.username}}Table"**></**div**>  
 <**div class="justify-content-center"**>  
 <**nav**>  
 <**ul class="pagination justify-content-center" id="{{this.username}}Pagination"**></**ul**>  
 </**nav**>  
 </**div**>  
 </**div**>  
 </**div**>  
 </**div**>  
 {{/each}}  
 </**div**>  
 </**div**>  
  
 </**div**>  
</**div**>  
  
<**script src="/assets/js/users.js" type="text/javascript"**></**script**>  
  
{{/partial}}  
{{> master path="users"}}

## dashboard.hbs

{{#partial "content"}}  
  
<**div class="row"**>  
 <**div class="col-md-9 mb-1"**>  
 <**div class="overview-board mb-1"**>  
 <**div class="card"**>  
 <**div class="card-body"**>  
 <**h5 class="card-title text-center"**>Overview</**h5**>  
 <**hr**>  
 <**div class="row"**>  
 <**div class="col-sm-12 mt-1 mb-1"**>  
 <**div class="card"**>  
 <**div class="card-body"**>  
 <**p class="card-title text-center"**>Total Tweets</**p**>  
 <**hr**>  
 <**h1 class="card-text text-center"**>{{totalTweets}}</**h1**>  
 </**div**>  
 </**div**>  
 </**div**>  
 <**div class="col-sm-6 mt-1 mb-1"**>  
 <**div class="card"**>  
 <**div class="card-body"**>  
 <**p class="card-title text-center"**>Total Hashtags</**p**>  
 <**hr**>  
 <**h1 class="card-text text-center"**>{{totalHashtags}}</**h1**>  
 </**div**>  
 </**div**>  
 </**div**>  
 <**div class="col-sm-6 mt-1 mb-1"**>  
 <**div class="card"**>  
 <**div class="card-body"**>  
 <**p class="card-title text-center"**>Total Usernames</**p**>  
 <**hr**>  
 <**h1 class="card-text text-center"**>{{totalUsernames}}</**h1**>  
 </**div**>  
 </**div**>  
 </**div**>  
 <**div class="col-sm-6 mt-1 mb-1"**>  
 <**div class="card"**>  
 <**div class="card-body"**>  
 <**p class="card-title text-center"**>Total Rumour Classifications</**p**>  
 <**hr**>  
 <**h1 class="card-text text-center"**>{{totalRumours}}</**h1**>  
 </**div**>  
 </**div**>  
 </**div**>  
 <**div class="col-sm-6 mt-1 mb-1"**>  
 <**div class="card"**>  
 <**div class="card-body"**>  
 <**p class="card-title text-center"**>Total Non-Rumour Classifications</**p**>  
 <**hr**>  
 <**h1 class="card-text text-center"**>{{totalNonRumours}}</**h1**>  
 </**div**>  
 </**div**>  
 </**div**>  
 <**div class="col-sm-12 mt-1 mb-1"**>  
 <**div class="card"**>  
 <**div class="card-body"**>  
 <**p class="card-title text-center"**>Total Classifications</**p**>  
 <**hr**>  
 <**h1 class="card-text text-center"**>{{totalClassifications}}</**h1**>  
 </**div**>  
 </**div**>  
 </**div**>  
 </**div**>  
 </**div**>  
 </**div**>  
 </**div**>  
 <**div class="filter-list-board mt-1"**>  
 <**div class="card"**>  
 <**div class="card-body"**>  
 <**h5 class="card-title text-center"**>Filter List</**h5**>  
 <**hr**>  
 <**div class="row"**>  
 <**div class="col-sm-12 mt-1 mb-1"**>  
 <**div class="card"**>  
 <**div class="card-body"**>  
 <**div class="filter"**>  
 {{#each filterList}}  
 <**a class="text-light"**>{{this}}</**a**>  
 {{/each}}  
 </**div**>  
 </**div**>  
 </**div**>  
 </**div**>  
 </**div**>  
 </**div**>  
 </**div**>  
 </**div**>  
 </**div**>  
 <**div class="col-md-3"**>  
 <**div class="service-board"**>  
 <**div class="card"**>  
 <**div class="card-body"**>  
 <**h5 class="card-title text-center"**>Service Status</**h5**>  
 <**hr**>  
 <**ul class="list-group"**>  
 {{#each serviceList}}  
 <**li class="list-group-item {{{listGroupColour this this.running}}}"**>{{this.serviceName}}</**li**>  
 {{/each}}  
 </**ul**>  
 </**div**>  
 </**div**>  
 </**div**>  
 </**div**>  
</**div**>  
  
{{/partial}}  
{{> master path="dashboard"}}

## hashtags.hbs

{{#partial "content"}}  
  
<**div class="row"**>  
 <**div class="col-md-8 mb-1"**>  
 <**div class="visualisation-boards"**>  
 <**ul class="nav nav-tabs mr-auto d-none" role="tablist"**>  
 <**li class="nav-item"**>  
 <**a class="nav-link" role="tab" data-toggle="tab"  
 href="#default"**>Default Tab</**a**>  
 </**li**>  
 {{#each hashtagResultsList}}  
 <**li class="nav-item"**>  
 <**a class="nav-link" role="tab" data-toggle="tab"  
 href="#{{this.hashtagValue}}"**>{{this.hashtagValue}}</**a**>  
 </**li**>  
 {{/each}}  
 </**ul**>  
 <**div class="tab-content"**>  
 <**div id="default" class="container tab-pane active"**><**br**>  
 <**h5**>Top Hashtags</**h5**>  
 <**p**>The hashtags classifications will be displayed using various techniques, such as word clouds,  
 charts etc.</**p**>  
 <**p**>In the table on the right, please select the one you wish to view.</**p**>  
 </**div**>  
 {{#each hashtagResultsList}}  
 <**div id="{{this.hashtagValue}}" class="container tab-pane fade" data-rumour="{{this.countOfRumours}}" data-non-rumour="{{this.countOfNonRumours}}"**><**br**>  
 {{> partials/visualisation-board value=this.hashtagValue }}  
 </**div**>  
 {{/each}}  
 </**div**>  
 </**div**>  
 </**div**>  
 <**div class="col-md-4"**>  
 <**div class="hashtag-table"**>  
 <**div class="card"**>  
 <**div class="card-body"**>  
 <**h5 class="card-title text-center"**>Hashtags</**h5**>  
 <**hr**>  
 <**table class="table table-striped"**>  
 <**thead**>  
 <**tr**>  
 <**th scope="col"**>Rank</**th**>  
 <**th scope="col"**>Hashtag</**th**>  
 <**th scope="col"**></**th**>  
 </**tr**>  
 </**thead**>  
 <**tbody**>  
 {{#each hashtagResultsList}}  
 <**tr**>  
 <**th scope="row"**>{{increment @index}}</**th**>  
 <**td**>  
 <**div class="radiotext"**>  
 <**label for="{{this.hashtagValue}}RadioInput"**>{{this.hashtagValue}}</**label**>  
 </**div**>  
 </**td**>  
 <**td**>  
 <**div class="radio"**>  
 <**label**>  
 <**input type="radio" id="{{this.hashtagValue}}RadioInput"  
 name="hashtagRadioGroup"  
 data-target="#{{this.hashtagValue}}"**>  
 </**label**>  
 </**div**>  
 </**td**>  
 </**tr**>  
 {{/each}}  
 </**tbody**>  
 </**table**>  
 </**div**>  
 </**div**>  
 </**div**>  
 </**div**>  
 <**div class="col-md-12 mb-1 mt-1"**>  
 <**div class="raw-row-data"**>  
 <**ul class="nav nav-tabs mr-auto d-none" role="tablist"**>  
 <**li class="nav-item"**>  
 <**a class="nav-link" role="tab" data-toggle="tab" href="#none"**>Default Tab</**a**>  
 </**li**>  
 {{#each hashtagResultsList}}  
 <**li class="nav-item"**>  
 <**a class="nav-link" role="tab" data-toggle="tab"  
 href="#{{this.hashtagValue}}RawData"**>{{this.hashtagValue}} Raw Data</**a**>  
 </**li**>  
 {{/each}}  
 </**ul**>  
 <**div class="tab-content"**>  
 <**div id="none" class="container tab-pane active"**><**br**>  
 </**div**>  
  
 {{#each hashtagResultsList}}  
 <**div id="{{this.hashtagValue}}RawData" class="container tab-pane fade"  
 data-size="{{this.totalCountOfClassifications}}"**><**br**>  
 <**div class="card"**>  
 <**div class="card-body"**>  
 <**h5 class="card-title text-center"**>Raw Data for #{{this.hashtagValue}}</**h5**>  
 <**hr**>  
 <**div class="total-results-tip justify-content-center text-right" id="{{this.hashtagValue}}PaginationTip"**>  
 <**p**>Total results: {{this.totalCountOfClassifications}}</**p**>  
 </**div**>  
 <**div id="{{this.hashtagValue}}Table"**></**div**>  
 <**div class="justify-content-center"**>  
 <**nav**>  
 <**ul class="pagination justify-content-center" id="{{this.hashtagValue}}Pagination"**></**ul**>  
 </**nav**>  
 </**div**>  
 </**div**>  
 </**div**>  
 </**div**>  
 {{/each}}  
 </**div**>  
 </**div**>  
  
 </**div**>  
</**div**>  
  
<**script src="/assets/js/hashtags.js" type="text/javascript"**></**script**>  
  
{{/partial}}  
{{> master path="hashtags"}}

## partials/visualtisation-board.hbs

<**div class="tabs-x align-center tabs-above tab-bordered"**>  
 <**ul class="nav nav-tabs mr-auto" role="tablist"**>  
 <**li class="nav-item"**>  
 <**a class="nav-link active show" id="{{value}}PieChartTab" role="tab" data-toggle="tab" href="#{{value}}PieChart" data-target="#{{value}}PieChart"**>Pie Chart</**a**>  
 </**li**>  
 <**li class="nav-item"**>  
 <**a class="nav-link" id="{{value}}BarChartTab" role="tab" data-toggle="tab" href="#{{value}}BarChart" data-target="#{{value}}BarChart"**>Bar Chart</**a**>  
 </**li**>  
 <**li class="nav-item"**>  
 <**a class="nav-link" id="{{value}}TimeLineChartTab" role="tab" data-toggle="tab" href="#{{value}}TimeLineChart" data-target="#{{value}}TimeLineChart"**>Timeline</**a**>  
 </**li**>  
 </**ul**>  
 <**div class="tab-content"**>  
 <**div id="{{value}}PieChart" class="container tab-pane fade show active" aria-labelledby="{{value}}PieChartTab"**><**br**>  
 <**div class="chart-container"**>  
 <**canvas id="{{value}}PieChartCanvas"**></**canvas**>  
 </**div**>  
 </**div**>  
 <**div id="{{value}}BarChart" class="container tab-pane fade" aria-labelledby="{{value}}BarChartTab"**><**br**>  
 <**div class="chart-container"**>  
 <**canvas id="{{value}}BarChartCanvas"**></**canvas**>  
 </**div**>  
 </**div**>  
 <**div id="{{value}}TimeLineChart" class="container tab-pane fade" aria-labelledby="{{value}}TimeLineChartTab"**><**br**>  
 <**div class="chart-container"**>  
 <**canvas id="{{value}}TimeLineChartCanvas"**></**canvas**>  
 </**div**>  
 </**div**>  
 </**div**>  
</**div**>

## exceptions/

### exceptions.hbs

{{#partial "content" }}  
  
<**h1**>Oops, an encounter has occurred and has been logged.</**h1**>  
  
<**p**>Please try go to the <**a href="/"**>home page</**a**>, if this occurs frequently please contact support.</**p**>  
  
{{/partial}}  
{{> master path="exception"}}

### not-found.hbs

{{#partial "content" }}  
  
<**h1**>Oops, page has not been found.</**h1**>  
  
<**p**>Please go to the <**a href="/"**>home page</**a**>, if this page has been found in error please contact support.</**p**>  
  
{{/partial}}  
{{> master path="not-found"}}

## hashtags.js

**$**(**'input[name="hashtagRadioGroup"]'**).click(**function** () {  
 **var** target = **$**(**this**).data(**"target"**);  
 $(**"a[href='"** + target + **"']"**).tab(**'show'**);  
 $(**"a[href='"** + target + **"RawData']"**).tab(**'show'**);  
  
 getHashtagTable(target, 0);  
 getHashtagPieChart(target);  
 getHashtagBarChart(target);  
 getHashtagTimeLineChart(target);  
});  
  
**function** getHashtagTable(target, page) {  
  
 $.ajax({  
 url: **'http://localhost:9000/hashtags/'** + target.substring(1, target.length) + **'/'** + page + **'/10'**,  
 dataType: **'json'**,  
 async: **true**,  
 success: **function** (data) {  
  
 $(**"div"** + target + **"Table"**).empty();  
  
 **var** table = **"<table class=\"table table-striped table-sm\">"** +  
 **"<thead>"** +  
 **"<tr>"** +  
 **"<th scope=\"col\">Tweet ID</th>"** +  
 **"<th scope=\"col\">Classification Value</th>"** +  
 **"<th scope=\"col\">Tweet Text</th>"** +  
 **"</tr>"** +  
 **"</thead>"** +  
 **"<tbody>"**;  
  
 data.forEach(**function** (tweet) {  
 table += **'<tr><th scope="row">'** + tweet.id + **'</th><td><p>'** + tweet.classificationValue + **'</p></td><td>'** + tweet.tweetText + **'</td></tr>'** });  
  
 table += **"</tbody></table>"**;  
  
 $(**"div"** + target + **"Table"**).append(table);  
  
 $(target + **"Pagination"**).twbsPagination({  
 totalPages: Math.ceil($(**"div"** + target + **"RawData"**).data(**"size"**) / 10),  
 visiblePages: 4,  
 onPageClick: **function** (event, page) {  
  
 **var** dbPage;  
  
 **if** (page === 1) {  
 *// page 1 on pagination should relate to offset 0 on DB* dbPage = 0;  
 } **else** {  
 dbPage = (page \* 10) - 10;  
 }  
  
 getHashtagTable(target, dbPage);  
 }  
 });  
 },  
 error: **function** () {  
 $(**"div"** + target + **"Table"**).empty();  
 $(**"div"** + target + **"Table"**).append(**"<h3>Issue retrieving table for "** + target + **"</h3>"**);  
 }  
 })  
}  
  
**function** getHashtagPieChart(target) {  
  
 **var** canvas = $(**"canvas"** + target + **"PieChartCanvas"**);  
  
 (**new** Chart(canvas, {  
 type: **'doughnut'**,  
 data: {  
 labels: [**"rumour"**, **"non-rumour"**],  
 datasets: [{  
 data: [$(**"div"**+target+**""**).data(**"rumour"**), $(**"div"**+target+**""**).data(**"non-rumour"**)],  
 backgroundColor: [  
 **'rgb(255, 99, 132)'**,  
 **'rgb(75, 192, 192)'** ]  
 }]  
 },  
  
 options: {  
 responsive: **true**,  
 maintainAspectRatio: **false**,  
 legend: {  
 position: **'bottom'** },  
 title: {  
 display: **true**,  
 text: target + **" Pie Chart"** },  
 tooltips: {  
 callbacks: {  
 label: **function**(tooltipItem, data) {  
 **var** dataset = data.datasets[tooltipItem.datasetIndex];  
 **var** total = dataset.data.reduce(**function** (previousValue, currentValue) {  
 **return** previousValue + currentValue;  
 });  
 **var** currentValue = dataset.data[tooltipItem.index];  
 **var** percentage = Math.floor(((currentValue/total) \* 100) + 0.5);  
 **return** currentValue + **" ("** + percentage + **"%)"**;  
 }  
 }  
 }  
 }  
 }));  
}  
  
**function** getHashtagBarChart(target) {  
  
 **var** canvas = $(**"canvas"** + target + **"BarChartCanvas"**);  
  
 (**new** Chart(canvas, {  
 type: **'bar'**,  
 data: {  
 datasets: [{  
 label: **"rumour"**,  
 data: [$(**"div"**+target+**""**).data(**"rumour"**)],  
 backgroundColor: **'rgb(255, 99, 132)'** }, {  
 label: **"non-rumour"**,  
 data: [$(**"div"**+target+**""**).data(**"non-rumour"**)],  
 backgroundColor: **'rgb(75, 192, 192)'** }]  
 },  
  
 options: {  
 responsive: **true**,  
 maintainAspectRatio: **false**,  
 legend: {  
 position: **'bottom'** },  
 title: {  
 display: **true**,  
 text: target + **" Bar Chart"** },  
 tooltips: {  
 callbacks: {  
 label: **function**(tooltipItem, data) {  
 **var** dataset = data.datasets;  
 **var** total = dataset.reduce(**function** (previousValue, currentValue) {  
 **return** previousValue.data[0] + currentValue.data[0];  
 });  
 **var** currentValue = tooltipItem.yLabel;  
 **var** percentage = Math.floor(((currentValue/total) \* 100) + 0.5);  
 **return** currentValue + **" ("** + percentage + **"%)"**;  
 }  
 }  
 },  
 scales: {  
 yAxes: [{  
 ticks: {  
 beginAtZero: **true** }  
 }]  
 }  
 }  
 }));  
}  
  
**function** getHashtagTimeLineChart(target) {  
 $.ajax({  
 url: **'http://localhost:9000/hashtags/'**+target.substring(1, target.length)+**'/timeline'**,  
 dataType: **'json'**,  
 async: **true**,  
 success: **function** (data) {  
 **var** canvas = $(**"canvas"** + target + **"TimeLineChartCanvas"**);  
  
 (**new** Chart(canvas, {  
 type: **'line'**,  
 data: {  
 labels: [**"Within last hour"**, **"1-2 hours ago"**, **"2-3 hours ago"**, **"3-4 hours ago"**, **"4-5 hours ago"**],  
 datasets: [{  
 label: **"rumour"**,  
 data: [data.rumoursLastHour, data.rumoursOverOneHour, data.rumoursOverTwoHour, data.rumoursOverThreeHour, data.rumoursOverFourHour],  
 borderColor: **'rgb(255, 99, 132)'**,  
 backgroundColor: **'rgb(255, 99, 132)'**,  
 fill: **false** }, {  
 label: **"non-rumour"**,  
 data: [data.nonRumoursLastHour, data.nonRumoursOverOneHour, data.nonRumoursOverTwoHour, data.nonRumoursOverThreeHour, data.nonRumoursOverFourHour],  
 borderColor: **'rgb(75, 192, 192'**,  
 backgroundColor: **'rgb(75, 192, 192'**,  
 fill: **false** }]  
 },  
  
 options: {  
 responsive: **true**,  
 maintainAspectRatio: **false**,  
 legend: {  
 position: **'bottom'** },  
 title: {  
 display: **true**,  
 text: **"Timeline of Rumours and Non-Rumours for "** + target.substring(1, target.length) + **" within last 5 hours"** },  
 scales: {  
 yAxes: [{  
 ticks: {  
 beginAtZero: **true** }  
 }]  
 }  
 }  
 }));  
 }  
 });  
}

## navigation.js

**$**(**document**).ready(**function** () {  
  
 **$**(**"button#searchTerm"**).click(**function** (event) {  
 **console**.log(**"Searching..."**);  
 event.preventDefault();  
  
 **if** (**$**(**"#searchTermValue"**).val() !== **''**) {  
  
 **var** searchTerm = **$**(**"#searchTermValue"**).val();  
  
 **console**.log(searchTerm);  
  
 **window**.**location**.replace(**"/search/"** + searchTerm);  
  
 } **else** {  
  
 **$**(**"#searchTermValue"**).addClass(**"alert-danger"**);  
  
 **$**(**"section.messages"**).**children**().remove();  
  
 **$**(**"section.messages"**).append(**"<div class='alert alert-warning alert-dismissible fade show' role='alert'>"** +  
 **" <button type='button' class='close' data-dismiss='alert' aria-label='Close'>"** +  
 **" <span aria-hidden='true'>&times;</span>"** +  
 **" </button>"** +  
 **" Search term must not be empty"** +  
 **"</div>"**);  
  
 **$**(**"section.messages .alert"**).**delay**(10000).**slideUp**(200, **function** () {  
 **$**(**this**).**alert**(**"close"**);  
 **$**(**"#searchTermValue"**).removeClass(**"alert-danger"**);  
 });  
 }  
 });  
});

## search.js

**$**(**document**).ready(**function** () {  
  
 **var** searchTerm = **$**(**"span.search-term"**).text();  
  
 *getTableForSearch*(searchTerm, 0);  
 *getSearchResultsBarChart*(searchTerm);  
 *getSearchResultsPieChart*(searchTerm);  
 *getSearchTimeLineChart*(searchTerm);  
});  
  
**function** *getSearchResultsPieChart*(searchTerm) {  
  
 **var** canvas = **$**(**"canvas#searchTermPieChartCanvas"**);  
  
 (**new** *Chart*(canvas, {  
 **type**: **'doughnut'**,  
 **data**: {  
 **labels**: [**"rumour"**, **"non-rumour"**],  
 **datasets**: [{  
 **data**: [**$**(**"div#searchTerm"**).data(**"rumour"**), **$**(**"div#searchTerm"**).data(**"non-rumour"**)],  
 **backgroundColor**: [  
 **'rgb(255, 99, 132)'**,  
 **'rgb(75, 192, 192)'** ]  
 }]  
 },  
  
 **options**: {  
 **responsive**: **true**,  
 **maintainAspectRatio**: **false**,  
 **legend**: {  
 **position**: **'bottom'** },  
 **title**: {  
 **display**: **true**,  
 **text**: searchTerm + **" Pie Chart"** },  
 **tooltips**: {  
 **callbacks**: {  
 label: **function**(tooltipItem, data) {  
 **var** dataset = data.**datasets**[tooltipItem.**datasetIndex**];  
 **var** total = dataset.**data**.reduce(**function** (previousValue, currentValue) {  
 **return** previousValue + currentValue;  
 });  
 **var** currentValue = dataset.**data**[tooltipItem.**index**];  
 **var** percentage = ***Math***.floor(((currentValue/total) \* 100) + 0.5);  
 **return** currentValue + **" ("** + percentage + **"%)"**;  
 }  
 }  
 }  
 }  
 }));  
}  
  
**function** *getSearchResultsBarChart*(searchTerm) {  
  
 **var** canvas = **$**(**"canvas#searchTermBarChartCanvas"**);  
  
 (**new** *Chart*(canvas, {  
 **type**: **'bar'**,  
 **data**: {  
 **datasets**: [{  
 **label**: **"rumour"**,  
 **data**: [**$**(**"div#searchTerm"**).data(**"rumour"**)],  
 **backgroundColor**: **'rgb(255, 99, 132)'** }, {  
 **label**: **"non-rumour"**,  
 **data**: [**$**(**"div#searchTerm"**).data(**"non-rumour"**)],  
 **backgroundColor**: **'rgb(75, 192, 192)'** }]  
 },  
  
 **options**: {  
 **responsive**: **true**,  
 **maintainAspectRatio**: **false**,  
 **legend**: {  
 **position**: **'bottom'** },  
 **title**: {  
 **display**: **true**,  
 **text**: searchTerm + **" Bar Chart"** },  
 **tooltips**: {  
 **callbacks**: {  
 label: **function**(tooltipItem, data) {  
 **var** dataset = data.**datasets**;  
 **var** total = dataset.reduce(**function** (previousValue, currentValue) {  
 **return** previousValue.**data**[0] + currentValue.data[0];  
 });  
 **var** currentValue = tooltipItem.yLabel;  
 **var** percentage = Math.floor(((currentValue/total) \* 100) + 0.5);  
 **return** currentValue + **" ("** + percentage + **"%)"**;  
 }  
 }  
 },  
 scales: {  
 yAxes: [{  
 ticks: {  
 beginAtZero: **true** }  
 }]  
 }  
 }  
 }));  
}  
  
  
  
**function** getTableForSearch(searchTerm, page) {  
  
 $.ajax({  
 url: **'http://localhost:9000/search/'** + searchTerm + **'/'** + page + **'/10'**,  
 dataType: **'json'**,  
 async: **true**,  
 success: **function** (data) {  
  
 $(**"div#searchTermTable"**).empty();  
  
 **var** table = **"<table class=\"table table-striped table-sm\">"** +  
 **"<thead>"** +  
 **"<tr>"** +  
 **"<th scope=\"col\">Tweet ID</th>"** +  
 **"<th scope=\"col\">Classification Value</th>"** +  
 **"<th scope=\"col\">Tweet Text</th>"** +  
 **"</tr>"** +  
 **"</thead>"** +  
 **"<tbody>"**;  
  
 data.forEach(**function** (tweet) {  
 table += **'<tr><th scope="row">'** + tweet.id + **'</th><td><p>'** + tweet.classificationValue + **'</p></td><td>'** + tweet.tweetText + **'</td></tr>'** });  
  
 table += **"</tbody></table>"**;  
  
 $(**"div#searchTermTable"**).append(table);  
  
 $(**"#searchTermPagination"**).twbsPagination({  
 totalPages: Math.ceil($(**"div#searchTermRawData"**).data(**"size"**) / 10),  
 visiblePages: 4,  
 onPageClick: **function** (event, page) {  
  
 **var** dbPage;  
  
 **if** (page === 1) {  
 *// page 1 on pagination should relate to offset 0 on DB* dbPage = 0;  
 } **else** {  
 dbPage = (page \* 10) - 10;  
 }  
  
 getTableForSearch(searchTerm, dbPage);  
 }  
 });  
 },  
 error: **function** () {  
 $(**"div#searchTermTable"**).empty();  
 $(**"div#searchTermTable"**).append(**"<h3>Issue retrieving table for "** + searchTerm + **"</h3>"**);  
 }  
 })  
}  
  
**function** getSearchTimeLineChart(searchTerm) {  
 $.ajax({  
 url: **'http://localhost:9000/search/'**+searchTerm+**'/timeline'**,  
 dataType: **'json'**,  
 async: **true**,  
 success: **function** (data) {  
 **var** canvas = $(**"canvas#searchTermTimeLineChartCanvas"**);  
  
 (**new** Chart(canvas, {  
 type: **'line'**,  
 data: {  
 labels: [**"Within last hour"**, **"1-2 hours ago"**, **"2-3 hours ago"**, **"3-4 hours ago"**, **"4-5 hours ago"**],  
 datasets: [{  
 label: **"rumour"**,  
 data: [data.rumoursLastHour, data.rumoursOverOneHour, data.rumoursOverTwoHour, data.rumoursOverThreeHour, data.rumoursOverFourHour],  
 borderColor: **'rgb(255, 99, 132)'**,  
 backgroundColor: **'rgb(255, 99, 132)'**,  
 fill: **false** }, {  
 label: **"non-rumour"**,  
 data: [data.nonRumoursLastHour, data.nonRumoursOverOneHour, data.nonRumoursOverTwoHour, data.nonRumoursOverThreeHour, data.nonRumoursOverFourHour],  
 borderColor: **'rgb(75, 192, 192'**,  
 backgroundColor: **'rgb(75, 192, 192'**,  
 fill: **false** }]  
 },  
  
 options: {  
 responsive: **true**,  
 maintainAspectRatio: **false**,  
 legend: {  
 position: **'bottom'** },  
 title: {  
 display: **true**,  
 text: **"Timeline of Rumours and Non-Rumours for "** + searchTerm + **" within last 5 hours"** },  
 scales: {  
 yAxes: [{  
 ticks: {  
 beginAtZero: **true** }  
 }]  
 }  
 }  
 }));  
 }  
 });  
}

## users.js

**$**(**'input[name="userRadioGroup"]'**).click(**function** () {  
 **var** target = **$**(**this**).data(**"target"**);  
 **$**(**"a[href='"** + target + **"']"**).**tab**(**'show'**);  
 **$**(**"a[href='"** + target + **"RawData']"**).**tab**(**'show'**);  
  
 *getUsersTable*(target, 0);  
 *getUsersPieChart*(target);  
 *getUsersBarChart*(target);  
 *getUsersTimeLineChart*(target);  
});  
  
**function** *getUsersTable*(target, page) {  
  
 **$**.**ajax**({  
 **url**: **'http://localhost:9000/users/'** + target.substring(1, target.**length**) + **'/'** + page + **'/10'**,  
 **dataType**: **'json'**,  
 **async**: **true**,  
 success: **function** (data) {  
  
 **$**(**"div"** + target + **"Table"**).empty();  
  
 **var** table = **"<table class=\"table table-striped table-sm\">"** +  
 **"<thead>"** +  
 **"<tr>"** +  
 **"<th scope=\"col\">Tweet ID</th>"** +  
 **"<th scope=\"col\">Classification Value</th>"** +  
 **"<th scope=\"col\">Tweet Text</th>"** +  
 **"</tr>"** +  
 **"</thead>"** +  
 **"<tbody>"**;  
  
 data.forEach(**function** (tweet) {  
 table += **'<tr><th scope="row">'** + tweet.**id** + **'</th><td><p>'** + tweet.classificationValue + **'</p></td><td>'** + tweet.tweetText + **'</td></tr>'** });  
  
 table += **"</tbody></table>"**;  
  
 **$**(**"div"** + target + **"Table"**).append(table);  
  
 **$**(target + **"Pagination"**).**twbsPagination**({  
 **totalPages**: ***Math***.ceil(**$**(**"div"** + target + **"RawData"**).data(**"size"**) / 10),  
 **visiblePages**: 4,  
 onPageClick: **function** (event, page) {  
  
 **var** dbPage;  
  
 **if** (page === 1) {  
 *// page 1 on pagination should relate to offset 0 on DB* dbPage = 0;  
 } **else** {  
 dbPage = (page \* 10) - 10;  
 }  
  
 *getUsersTable*(target, dbPage);  
 }  
 });  
 },  
 error: **function** () {  
 **$**(**"div"** + target + **"Table"**).empty();  
 **$**(**"div"** + target + **"Table"**).append(**"<h3>Issue retrieving table for "** + target.substring(1, target.**length**) + **"</h3>"**);  
 }  
 })  
}  
  
**function** *getUsersPieChart*(target) {  
  
 **var** canvas = **$**(**"canvas"** + target + **"PieChartCanvas"**);  
  
 (**new** *Chart*(canvas, {  
 **type**: **'doughnut'**,  
 **data**: {  
 **labels**: [**"rumour"**, **"non-rumour"**],  
 **datasets**: [{  
 **data**: [**$**(**"div"**+target+**""**).data(**"rumour"**), **$**(**"div"**+target+**""**).data(**"non-rumour"**)],  
 backgroundColor: [  
 **'rgb(255, 99, 132)'**,  
 **'rgb(75, 192, 192)'** ]  
 }]  
 },  
  
 options: {  
 responsive: **true**,  
 maintainAspectRatio: **false**,  
 legend: {  
 position: **'bottom'** },  
 title: {  
 display: **true**,  
 text: target.substring(1, target.length) + **" Pie Chart"** },  
 tooltips: {  
 callbacks: {  
 label: **function**(tooltipItem, data) {  
 **var** dataset = data.datasets[tooltipItem.datasetIndex];  
 **var** total = dataset.data.reduce(**function** (previousValue, currentValue) {  
 **return** previousValue + currentValue;  
 });  
 **var** currentValue = dataset.data[tooltipItem.index];  
 **var** percentage = Math.floor(((currentValue/total) \* 100) + 0.5);  
 **return** currentValue + **" ("** + percentage + **"%)"**;  
 }  
 }  
 }  
 }  
 }));  
}  
  
**function** getUsersBarChart(target) {  
  
 **var** canvas = $(**"canvas"** + target + **"BarChartCanvas"**);  
  
 (**new** Chart(canvas, {  
 type: **'bar'**,  
 data: {  
 datasets: [{  
 label: **"rumour"**,  
 data: [$(**"div"**+target+**""**).data(**"rumour"**)],  
 backgroundColor: **'rgb(255, 99, 132)'** }, {  
 label: **"non-rumour"**,  
 data: [$(**"div"**+target+**""**).data(**"non-rumour"**)],  
 backgroundColor: **'rgb(75, 192, 192)'** }]  
 },  
  
 options: {  
 responsive: **true**,  
 maintainAspectRatio: **false**,  
 legend: {  
 position: **'bottom'** },  
 title: {  
 display: **true**,  
 text: target.substring(1, target.length) + **" Bar Chart"** },  
 tooltips: {  
 callbacks: {  
 label: **function**(tooltipItem, data) {  
 **var** dataset = data.datasets;  
 **var** total = dataset.reduce(**function** (previousValue, currentValue) {  
 **return** previousValue.data[0] + currentValue.data[0];  
 });  
 **var** currentValue = tooltipItem.yLabel;  
 **var** percentage = Math.floor(((currentValue/total) \* 100) + 0.5);  
 **return** currentValue + **" ("** + percentage + **"%)"**;  
 }  
 }  
 },  
 scales: {  
 yAxes: [{  
 ticks: {  
 beginAtZero: **true** }  
 }]  
 }  
 }  
 }));  
}  
  
**function** getUsersTimeLineChart(target) {  
 $.ajax({  
 url: **'http://localhost:9000/users/'**+target.substring(1, target.length)+**'/timeline'**,  
 dataType: **'json'**,  
 async: **true**,  
 success: **function** (data) {  
 **var** canvas = $(**"canvas"** + target + **"TimeLineChartCanvas"**);  
  
 (**new** Chart(canvas, {  
 type: **'line'**,  
 data: {  
 labels: [**"Within last hour"**, **"1-2 hours ago"**, **"2-3 hours ago"**, **"3-4 hours ago"**, **"4-5 hours ago"**],  
 datasets: [{  
 label: **"rumour"**,  
 data: [data.rumoursLastHour, data.rumoursOverOneHour, data.rumoursOverTwoHour, data.rumoursOverThreeHour, data.rumoursOverFourHour],  
 borderColor: **'rgb(255, 99, 132)'**,  
 backgroundColor: **'rgb(255, 99, 132)'**,  
 fill: **false** }, {  
 label: **"non-rumour"**,  
 data: [data.nonRumoursLastHour, data.nonRumoursOverOneHour, data.nonRumoursOverTwoHour, data.nonRumoursOverThreeHour, data.nonRumoursOverFourHour],  
 borderColor: **'rgb(75, 192, 192'**,  
 backgroundColor: **'rgb(75, 192, 192'**,  
 fill: **false** }]  
 },  
  
 options: {  
 responsive: **true**,  
 maintainAspectRatio: **false**,  
 legend: {  
 position: **'bottom'** },  
 title: {  
 display: **true**,  
 text: **"Timeline of Rumours and Non-Rumours for "** + target.substring(1, target.length) + **" within last 5 hours"** },  
 scales: {  
 yAxes: [{  
 ticks: {  
 beginAtZero: **true** }  
 }]  
 }  
 }  
 }));  
 }  
 });  
}

# Excluded files

This code listing only contains the SQL and Java files as a result of the project, the likes of the Docker files and Gradle files for building and deploying the project are excluded and can be provided separately if required. The test files are also excluded from this code listing.