Introduction to Software Engineering

Trend It

Filipe Gonçalves, 98083 - Team Manager
Pedro Lopes, 97827 - DevOps
Vicente Costa, 98515 - Architect
João Borges, 98155 - Product Owner

Universidade de Aveiro - 2021/2022

First Iteration

In this first iteration, we want to have a skeleton of the application, and define the key ideas for the project

Personas and User Stories

Made three personas, each one with a case scenario and a user story associated: One persona was made for lazure utilization of the application (browsing and knowing what's happening around the world and which trends to follow), while the other two, for business purposes (management of trends and/or knowing what's trending with the intention of creating a new business).

Example of user story:

- As an aspirant fashionist, Anabela would like to cope up with the latest trends in the fashion world with the minimum time possible.
- While creating an account on the app, she put as a topic of interest "fashion", and with that, on her home page, the most popular topics with the that hashtag should appear
- She can filter the fashion topics to only "carhartt", for example, so that she can see what is trending on her favorite brand

Backlog

In software development, a product backlog is a prioritized list of deliverables (such as new features, bugfixes or hotfixes) that should be implemented as part of a project or product development.
It’s usually a way for team members to organize themselves in a way that two members are doing the same thing or someone is doing something more than once.
This tool is also great to maintain an overview of how the project is going, since for every entry in the backlog we can assign points that should reflect the time/complexity of the work developed.
It’s also great to include ideas in the backlog even if they don’t get implemented. When thinking about backlogging we should think about it as a wishlist where everyone can review everyone's work in order to move forward with the product.
There are many tools to help us develop a backlog, for now we will be using the project tab in github alongside with repository branching to fix bugs or misconceptions, and to report issues developed by the team members.

Project

In the project Trend It we did the index, home, login and register page, in accordance with the user stories defined before.
The index page works as a welcome to the webpage, which will redirect to login.html for login purposes, or in case the user doesn't have an account, to the register.html. With the login/register made, it will show the home page which will be used to show both tweets and statistics.
Because it's still a prototype, it's still too early to know if we will add more/remove some.

Architect Notebook

Software project management and comprehension tool

Maven: maven is a build tool, it is also a software project management and comprehension tool, based on the concept of a project object model (POM), which should make things easier in organization and compilation.

Web java-based application tool

Java Spring boot
Java Spring
Spring Initializr
Spring Tools
Thymeleaf

Each one of these, will make it easier to acquire a RESTful application, with both a normal Java Spring Controller and a Java Spring Boot REST Controller, Model and Views with the help of Thymeleaf and Spring Initializr to help us build the project

Container

Docker: Using docker will help us create a space for both the database and the project to run, whilst not being completely on our computer.

Database

Requisites:

- Database oriented to lines
- Very high performance
- High availability
- Horizontal scalability
- Easy filter of content

MySql

Characteristics	**Positive or Negative aspects**
Oriented to lines	Positive
Great understanding from the developers	Positive
Scale well if configured correctly	Positive/Negative
Vertical scalability	Negative
Very high availability	Positive

MongoDB

Characteristics	**Positive or Negative aspects**
Document database	Positive/Negative
Vertical and horizontal scalability and high availability (with sharding and replica sets)	Positive
Great for transactional, search, analytics	Positive
Performance limitations can happen due to inadequate or inappropriate indexing strategies, or as a consequence of poor schema design patterns	Negative
The developers have experience and understanding of MongoDB	Positive/Negative

Cassandra

Characteristics	**Positive or Negative aspects**
Oriented to column	Positive
distributed (Maybe overkill)	Positive
horizontal scalability	Positive
high performance and availability	Positive
developers of the group are less experienced with this tool	Negative

Looking at the pros and cons of all types of databases, we can say that, for this project, mysql is the worst option, as the application does not require a relational database, with complex schema. From the last two options, we decided to choose MongoDB because of the experience the group members have in comparison with Cassandra, and because of the document-oriented way the database is made. While column-oriented database could help, working with tweets, will make us get the whole tweet object and not a single column from the database.

Twitter Bot

languages	Java	Python
Useful Sites	https://gist.github.com/dueyfinster/2469810	https://realpython.com/twitter-bot-python-tweepy/
	https://medium.com/@SeloSlav/how-to-make-a-scary-russian-twitter-bot-with-java-b7b62768a3ac	https://auth0.com/blog/how-to-make-a-twitter-bot-in-python-using-tweepy/

The twitter bot will be made in java because it will help in simplifying the database connection, with the creation of another maven + spring boot project application.

Second Iteration

In this second iteration, we started working on the concrete application, building the twitter bot to get dynamic data and containerize the project in docker.

Twitter Bot

The twitter bot application is implemented in java using maven and spring boot. It contains only a model, the Tweet class, which has an id, a tag, a timestamp and a description. It also contains a simple controller that can create and delete a tweet and find tweets by id or by tag.

To handle exceptions, we also created simple classes like the one we used on the practical classes (ErrorDetails, GlobalExceptionHandler and ResourceNotFound). For now we only use the ResourceNotFound in the controller.

For the TweetRepository, we decided to store the data with MongoDB so our TweetRepository is a mongo repository and it has a function to find tweets by tag.

Finally, our TwitterBotApp uses the library twitter4j and for now it makes two queries using the twitter API to get a tweet with a certain id and tweets from “hyperlegen”, “KingJames” and “DailyNASA”

Twitter API

For now we are using 3 endpoints of the Twitter API, which give us the possibility to search in the last 7 days: all tweets matching a certain query; the number of tweets posted daily matching a certain query; twitter trends for a certain country, city or even worldwide.

In order for us to interact with the api we are using Retrofit2 and Gson.
This allows us to properly authenticate to certain endpoints that must use a Bearer Token, which Twitter4j2 doesn’t support.

Container

As of the containers we used Docker, docker allows simple deployment and ensures that every member in the team is capable of running the app independent of the system they are using.

So far we have two containers, one for the Spring Boot web application, called Trendit, that will have a database connection to a mongodb server, and the other one for the database service. In order for all containers to communicate we need them to be in the same network and for that we use docker-compose, which is a simpler way of deploying a multi container application, allowing the user to define every service they need in a single file, instead of using a bash script or a batch to initialize every component in the right order and attaching them to the right network manually, since docker-compose offers the possibility of configuring each one of these aspects.

Web Application

The application contains, as of now, two Controllers, with Spring Boot annotation: The first, ViewController, with @Controller, works as the base for the html Views and Models (User and Tweet), so we can have access to the web application; The second, ApiController, with @RestController, works as the middleman for the CRUD actions (Create, Read, Update, Delete), where we can insert both users and tweets, see them, etc.

We also have two repositories, one for each model, which will save us the users/tweets in our database, find our users by username and our tweets by id (per example).

For the Models, we have a User Model, to represent everyone who wishes to register/login on the website, we save the username and the password when registering and compare them with the inputs obtained when signing in. We also have a Tweet Model, to represent our data, and it will be mostly shown in the home page, after the user signs in; all tweets which have the same “hashtags” that the User also has, will be shown, else they won’t.

Third Iteration

In this third iteration, we started by stabilizing both the REST Api and the Bot Api, so the changes we need to make are minimal and will not break the system, as well as trying to get dynamic data on the main page. We also added an ApiService to organize the methods from the ApiController, as well as make some changes in the ViewController.

REST Api

In the REST Api, we have our CRUD methods for both Tweet and User:
Tweet:

- getAllTweets() : will get every tweet possible in the database
- getTweetById(int id) : will get one tweet from the database with id equal to “id”
- insertTweet(Tweet tweet) : will add the tweet to the database
- updateTweet(int id, Tweet tweet) : will update the tweet with id equal to “id” with the data in “tweet”
- deleteTweet(int id) : will delete the tweet with id equal to “id”

User:

- getAllUsers() : will get every user possible in the database
- getUserById(int id) : will get one user from the database with id equal to “id”
- getUserByUsername(String username) : will get one user from the database with username equal to “username”
- insertUser(User user) : will add the user in the database
- updateUserById(int id, User user) : will update the user with id equal to “id” with the data in “user”
- updateUserByUsername(String username) : will update the user with username equal to “username” with the data in “user”
- deleteUserById(int id) : will delete the user with id equal to “id”
- deleteUserByUsername(String username) : will delete the user with username equal to “username”

Bot Api

As explained before, we are using 3 endpoints of the Twitter API, which give us the possibility to search tweets in the last 7 days. With the use of retrofit, present in the TwitterService.java (TTservice is used for easier development), we can execute the queries and obtain the data in an array of arrays with objects format. To process this information, we convert it into various objects, which are described in the model folder, so that we can manipulate them. After the tweet is received and finally sent to the interface by the RabbitMQ broker, the tweet will be shown on the interface and stored there. To understand better, we will explain a bit more in detail the twitter composition when it’s received from the app:
-The models are divided in three categories, according to the three endpoints of the twitter API.
-The search tweets are a model for the endpoint where as a response we get all tweets matching a certain query.
-The count tweets is a model for a search on the number of tweets posted daily about a certain query. It has the amount of tweets done (tweet count), and the date of the start and end.
-The tweet trends has the trends in a specific location. It will return the name of the trend, the query to do the search about this topic, the url to this topic, redirecting to the twitter website, and the tweet volume, which symbolises the amount of tweets per day with the keyword.

The queries can be rather simple, for example “https://api.twitter.com/2/tweets/search/recent?query=snow” will show the 10 most recent tweets about snow; we can change the search keyword to “counts” or “trends” so we get one of the other endpoints mentioned before. It is important to HTTP encode the link so we can get the data, and it would result in something similar to this: “https://api.twitter.com/2/tweets/search/recent?query=cat%20has%3Amedia%20-grumpy&tweet.fields=created_at&max_results=100 -H "Authorization: Bearer $BEARER_TOKEN" ”; most of this link works like the previous one, it’s pretty intuitive, the max_results show a certain amount of tweets done about a certain topic in the query for example; the bearer token is something needed so we have permission to receive the response.

View Controller

The changes made in the ViewController had the purpose of allowing the application to, when a user signs in on the website, the tweets shown, would only be of those where the user has an interest on, as well as, making statistic charts with the general data show up.

Fourth Iteration

In this iteration, we must complete the Minimal Viable Product (MVP), whilst correcting some errors and debug the code. We started by stabilising the data generation and data insertion on the database, as well as using that data on the graphic interface. The app is running in `192.169.160.237:9001`

Docker

We use containerization technology in our project so that we can isolate our applications, our project can better run in other interfaces, the project scalability is higher and we get a better harmony between our applications.

We made two docker-compose files, one for each maven application, so to run our project we run this command: `docker-compose -f docker-compose.yml -f docker-compose-bot.yml up/build`

The docker-compose files have four services that are connected with the network “backend” : the frontend application (TrendIt), the backend application (TwitterBot), the database service (MongoDB) and the messager trading between our two main applications (RabbitMQ).

The service TrendIt, besides some usual configuration like port (9001) and build, always restarts unless explicitly stopped or the docker is restarted and it depends on the db (mongo), but before the spring boot application starts it awaits for the availability of the RabbitMQ service to be ready to accept new connections. For the environment variables we have for spring MongoDB: data source url, username, password (data source and data MongoDB), host, port (27017) and JPA hibernate ddl auto update (environment variable). And for RabbitMQ: username, password and host.

The service db is an image of the latest version of mongo, like for TrendIt, it always restarts unless we explicitly say to don’t, running on port 27117 (of the host running the containerization). In the environment variables we have the usual username-password and database. For last, we have volumes to keep persistent data with a javascript file as an entrypoint to create an user.

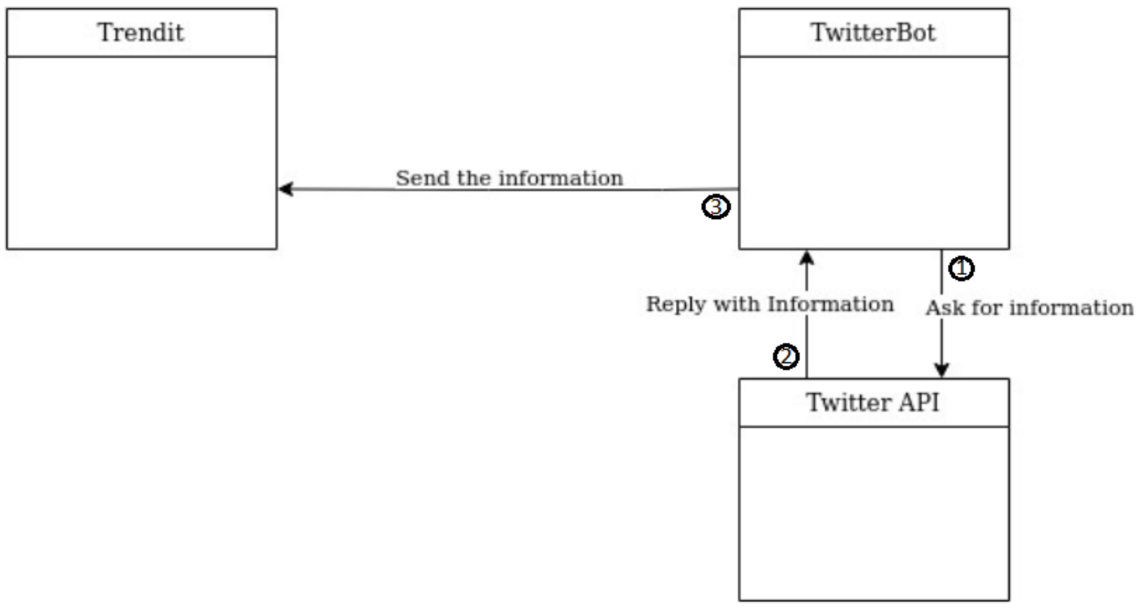
In the other Docker file, the TwitterBot doesn’t restart unless told to, it depends on the RabbitMQ service, running on port 9002. For the environment variables we have the same for MongoDB and RabbitMQ plus the token used for communication with the twitter API. For the entry points we have some small scripts that do a curl to the management specific port of RabbitMQ service, and then using the error code of that curl it decides whether or not to run the application.

For last, the RabbitMQ service is an image of rabbitmq:3-management, it always restarts, it runs in the ports 15672 and 5672 and has a default user and password in the environment variables ("pedro" and "pedro")

RabbitMQ

Like it was said before, the RabbitMQ is used for the communication between frontend and backend applications.

Down below we made a simple graph that shows the working sequence of our project.



In TwitterBot we have a Runner, whose purpose is to send a message using RabbitMQ, and in TrendIt we only have a Receiver that receives the data from RabbitMQ.

The Runner doesn't need initialization because it is a component and, since it is a commandLineRunner, it will run when the application fully starts. This class initially gets Trends, searches tweets and puts it in a queue and then, periodically (every 0.5 seconds, defined by us), it sends through the RabbitMQ to TrendIt.

The Receiver is used by the MessageListenerAdapter and translates the message to a tweet counter (TweetCount), trends (TweetTrendsJson) or tweet data (Datum), depending on the message.

FrontEnd

For the frontend, we decided to use 4 simple html pages, one being the index, two for authentication, register and login, and lastly for our home page after the login.

The most important part is the home html page, where we show our data in various ways and let the user visualise both tweets and statistical charts of them.

Firstly, when registering the user will be asked to select from a set of trends, trending in the United Kingdom (firstly was supposed to be in Portugal, but the amount of data we get from UK trends are much greater than the ones from Portugal, so we decided to change it). Upon login, the home page is divided in two halves: the left side, where the user can see their account data (incompleted because it was not exactly necessary for the project but will be talked about later) and 2 sets of charts, one for the user interests, and another for all trends in UK; the right side, where it shows all tweets, that we could gather, from the interests of the user, so if the user chose “Lakers” as a trend, it will only show up tweets with the tag (query, for the TwitterBot) “Lakers”.

The charts were made with the help of google charts, and it depends on the TweetCount, talked before, and because of that, sometimes we get tweets with the tag “Lakers“, for example, but we don’t have a TweetCount for the tag “Lakers”. However, the trends we do have TweetCount will show up on the graphs.

Future

This Minimal Viable Product is far from perfect, and so we know it needs some modifications, as well as new features to make it commercially appealing. As such, we present some features we want to add in the near future:

- Better graphs and charts for statistical purposes
- Interest selection inside the home page
- Account settings, so the user can update his stats
- A way to update user trends
- Favourite section for tweets and trends, so the user can save important data

Bibliography

First Iteration:
https://www.w3schools.com/w3css/w3css_templates.asp
https://www.mongodb.com/
https://cassandra.apache.org/_/index.html
https://www.mysql.com
https://www.perforce.com/resources/hns/agile-product-backlog-basics

Second Iteration:
https://developer.twitter.com/en/docs/twitter-api/tweets/search/integrate/build-a-query
https://developer.twitter.com/en/docs/twitter-api/tweets/counts/api-reference/get-tweets-counts-recent
https://developer.twitter.com/en/docs/twitter-api/v1/trends/trends-for-location/api-reference/get-trends-place
https://developer.twitter.com/en/docs/twitter-api/tweets/counts/api-reference/get-tweets-counts-recent
https://developer.twitter.com/en/docs/twitter-api/v1/trends/trends-for-location/api-reference/get-trends-place
https://github.com/twitterdev/Twitter-API-v2-sample-code
https://json2csharp.com/json-to-pojo
https://learning.oreilly.com/library/view/docker-deep-dive/9781800565135/chap11.xhtml#leanpub-auto-compose-files

Third Iteration:
https://spring.io/guides/gs/serving-web-content/

Fourth Iteration:
https://developers.google.com/chart
https://docs.spring.io/spring-boot/docs/1.5.6.RELEASE/reference/html/common-application-properties.html
https://stackoverflow.com/questions/46606427/rabbitmq-sending-objects
https://subscription.packtpub.com/book/application-development/9781849516501/1/ch01lvl1sec12/using-body-serialization-with-json
https://www.baeldung.com/gson-deserialization-guide
https://sites.google.com/site/gson/gson-user-guide
https://stackoverflow.com/questions/31746182/docker-compose-wait-for-container-x-before-starting-y
https://x-team.com/blog/set-up-rabbitmq-with-docker-compose/
https://spring.io/guides/gs/messaging-rabbitmq/
https://woeplanet.org/id/23424925/