Design

Packages

This system is divided in three main packages, data, server and bot. Note that classes that will only be used another one will be declared as nested classes inside of the class that uses them. All three of them under the same package upf.at.app. Each one of them is in charge of one part of the system and being them:

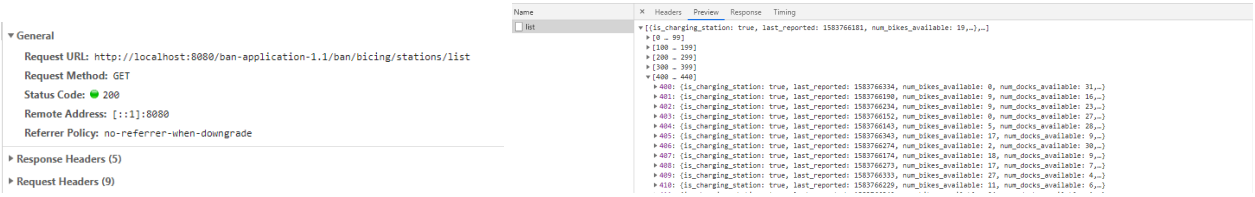
- **data [upf.at.app.data].** This package is in charge of holding the classes that are used to store data. Inside this package there are two folders "bicing" and "client", where the classes of the package bicing are used to help retrieve the stations from the bicing api and then storing the pulled information. On the otherside, the classes in the package client are the ones used to store the subscribed clients of the application.
- **server [upf.at.app.server].** The classes in this package are the ones in charge of the REST API implementation, which we will later cover in detail.
- **bot [upf.at.app.bot].** This package holds all the classes that implement the bot functionality, moreover, there is a subpackage "utils" which stores all the classes that the bot classes use to achieve its functionality.

REST API of BAN

This application is hold on three REST APIS, one that deals with the Bicing stations found as ServicesStations, another one that is in charge of all the Client storage called ServicesClient and finally one that is in charge of Notifications, in this case via telegram, named ServicesNotifier.
Now we will see more in detail each one of these REST APIS.

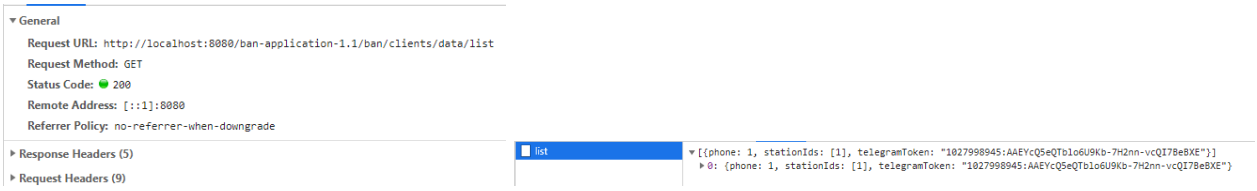
ServicesStations

This class is located under the relative PATH "/stations". Then this class only has one method under the relative PATH "/stations/list". This method takes no parameters and responses with a list of stations or responses with status code 404 if it could not find the stations, here is an example of a request and a response.

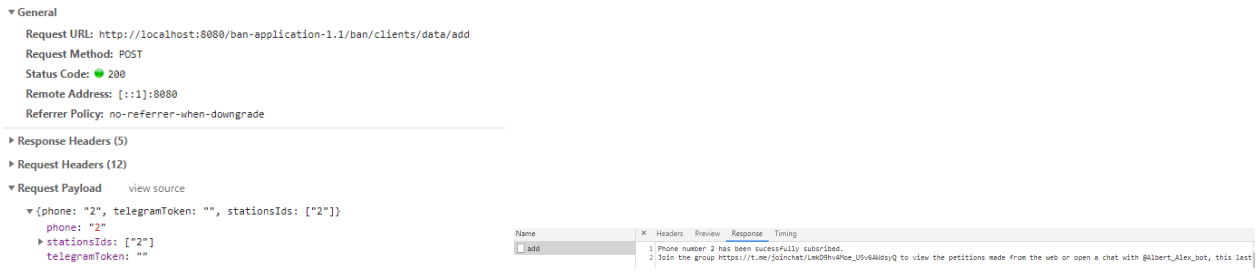


ServicesClient

This class is located under the relative PATH "/data". This class has two methods under the relative PATH "/data/list" and "/data/add". The first method takes no parameters and responses a list of clients. In case the list of clients is empty it will response with status code 225 and a message saying no clients subscribed. This method takes no parameters and here is an example of a request and a response.

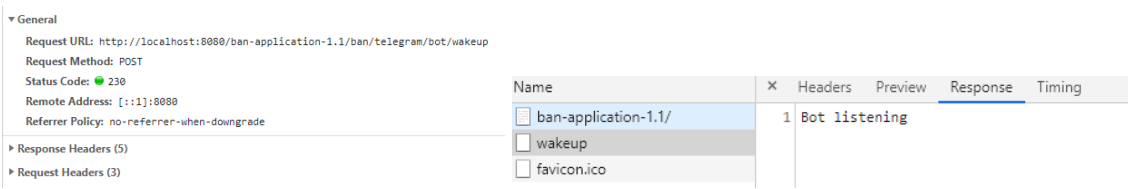


The second method takes a JSON file parameter with structure of a client and responses either with the status code 200 if the client has been subscribed, with a status code 210 if a client has been updated or with a status code 450 if the phone, list of stations or both things were not provided. The message that comes with this status code indicates what was missing.

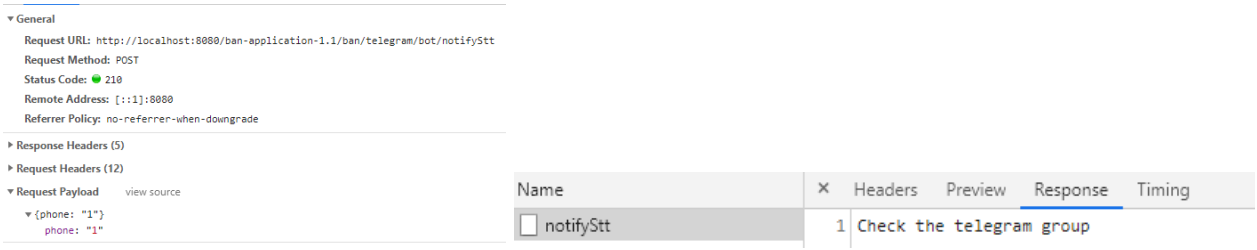


ServicesNotify

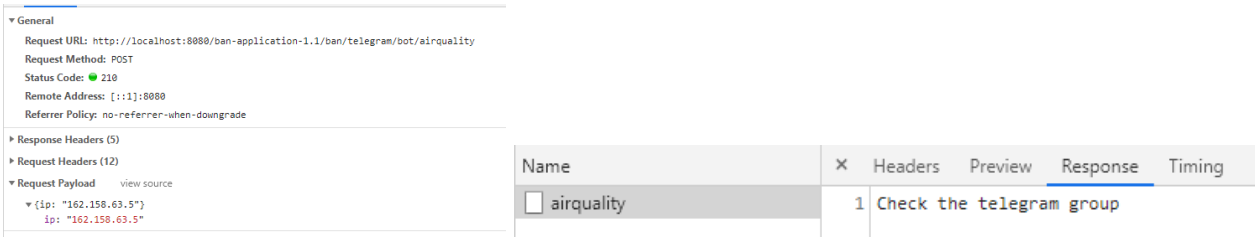
This class is located under the relative PATH "/bot". Then this class has three methods under the relative PATH "/bot/wakeup", "/bot/notifyStt" and "/bot/airquality". The first method takes no parameters and responses with status code 230 if the bot has been awaken or responses with a 235 if the bot was already awaken. This method takes no parameters and here is an example of a request and a response.



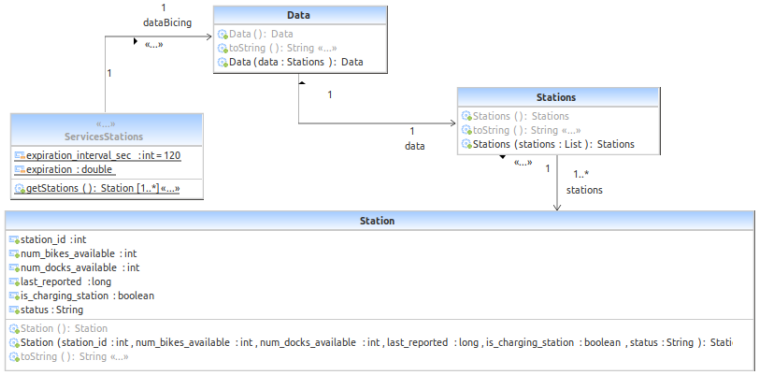
The second method takes a JSON file as a parameter with structure of a phone number and responses with a status code 210 and message check the telegram group. In case the phone was not provided this method will response with a status code 450 and the message, please insert a phone number.



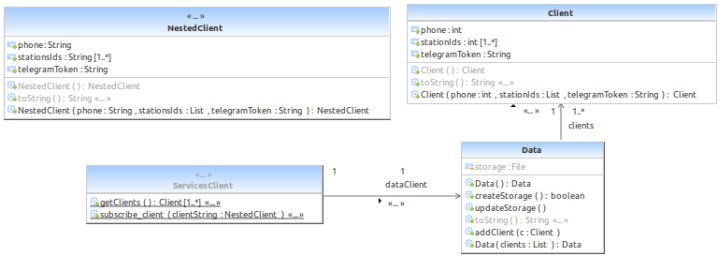
In the third method takes a JSON file as a parameter with structure of an IP and responses with a status code 210 and message check the telegram group.



Implementation

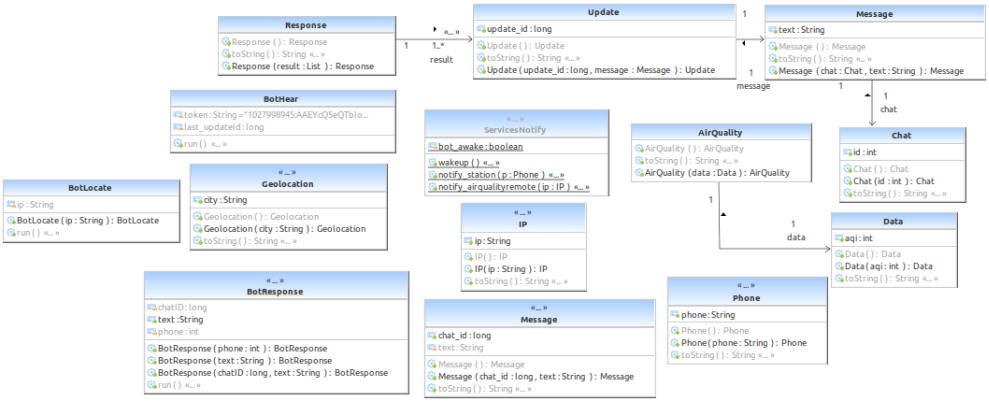


In this diagram we can observe all the classes that are used by the driving class of the Stations REST API. Once the method getStations is called, method that is in the relative path "/data/list", it will first check if the dataBicing does not contain anything or if the data that it contains is outdated. In case there is no data or that the data is outdated, it will get in contact with the Bicing API acting as a client to this server to retrieve all the information about the bicing stations in Barcelona. Once this data is retrieved it will get stored in the dataBicing attribute and we will set the expiration time of this data by taking the current time of the system and adding two minutes in milliseconds, then we store this expiration time on the attribute expiration. Now that the content of the if statement has finished this method getStations() will return the list of stations stored on the attribute dataBicing. In case there is data and is not outdated we directly return it.



In this diagram we can observe all the classes that are used by the driving class of the Stations REST API The method getClients acts as we expressed previously. Then inside subscribe_client method, we will transform all the received data as NestedClient to the primitive types that the class Client understands, in order to create a new Client and add it to the list of of clients inside Data.

Note that as we mentioned before, this method will give different responses depending if the client has been subscribed or it updated its previous state.



All this classes are the responsible of all the notification system. Since this part is a little more complex than previous ones, and we do not have a lot of space to explain it in details, we will sum up.
It all comes from the ServicesNotify since it is the REST API. First there is a method called wakeup, which is in charge of awakening the bot so it starts hearing all the incoming messages and answer them with the requested information (it only understands /getSlots {phone}, since telegram does not send the IP). Then there is the method notify_station, which uses the nested class Phone to understand the received JSON file and creates a thread of BotResponse passing the phone, and this thread will know what to do. Then there is the notify_airqualityremote, which uses the nested class IP to understand the received JSON, then it delegates the task to BotLocate passing the IP. Then BotLocate will use the classes AirQuality, Data and Geolocation, to locate the IP, get the air quality and convert it to a text that defines it. it will later send that text to ResponseBot that will know what it must be done.

BotResponse

This class is in charge of sending messages via telegram. It has three constructors, and depending of the constructor that was used, the parameters received, this class will be able to resolve what it must be done. The first constructor receives a phone number, so when this thread gets executed the chatID will be assigned as the default one, then we will look up for a client with the given phone and later we will check if the stationsID list is null. In case it is null, it means that there is no client subscribed with this phone. On the flip side, if it is not null, we will use the IDs of the stations to create a message, by appending the number of available docks in these stations, which we will later send over Telegram.
Then there is a second constructor that receives a string, once the thread is executed the chatID will be assigned to the default one and the phone number will take the value integer.MIN_VALUE. This phone number will make the bot skip all the finding client process and directly send a message with the string received in the constructor.
Finally there is a third constructor, which receives a chatID and a string text, then we split the string text by spaces and check if there are two resulting strings where the first resulting string is equal to "/getSlots". In case everything matches, we will allow the thread to get into the search client section, where it will search for a client with the given phone number (otained from the second substring) and obtain the available docks for the stations he is subscribed to. Finally we will send this message over telegram the same way we did when explaining the first constructor but with the main difference that the chatID is not the default one, so the message of available docks will be sent over the received chatID.

Execution

To run the application we need to deploy the .war file to webapps folder in tomcat directory, then open the server and go to http://localhost:8080/ban-application-1.1/.

- When you click the 'List Barcelona Bicing Stations' list all the bicing stations in BCN.
- When clicking the button list all subscribed clients, the server will respond with the list of all the clients that are subscribed to the application.
- When clicking subscribe now, the server will try to subscribe a client with the parameters introduced.
- Notify me button, sends to the Telegram group chat the available docks of the stations the client with the passed phone number is subscribed to.
- Air quality IP location button, sends to the Telegram group chat the air quality of the city where the received IP is found at.

Conclusions

This application worked perfectly in localhost, although the Telegram bot does not work in AWS, furthermore we have not delivered an URL to the AWS as it changes every time we close the instant. Notice that the application forces the subscribed clients the default Telegram token.

Note: We could not get in details as much as we wanted since we have already surpassed 4 pages long.