Milestone Report 2

13.06.2021

Group 5

Mehmet Fatih Aksoy, Muhammed Furkan Atasoy, Zuhal Didem Aytaç, Berkay Bör, Salih Bedirhan Eker, Umut Gün, Ozan Kılıç, Kemal Çağın Sertkaya, Gözde Ünver

Contents

1....Executive summary
    1.1....Summary of project and overall status
    1.2....URI of the deployed application
    1.3....API URL
    1.4....Basic Functionality of the project
    1.5....Challenges met as a group
    1.6....List and status of deliverables
2....Project Plan
3....Work Done by Each Member
4....Evaluation of Tools and Processes
5….Deliverables

1 Executive summary

1.1 Summary of the project and overall status

Our project is creating an amateur sports coordination platform. This platform will be provided on android and a website. The main virtue of the platform is that it brings people that are interested in the same kind of sports together so that they can meet new people, socialize and engage in sports they like. In addition to that, people can create accounts for commercial purposes for example sport shop owners or sport field owners can create an account in the name of their business, create posts and participate in events like every other user can do.

We have finished the requirements and decided the functionalities our project will provide as we provided the Milestone Report 1. Detailed explanation of the project, requirements, scenarios and mockups, class diagram, sequence diagrams, use case diagram and the project plan and RAM can be found in our Milestone Report 1.

After delivering the first Milestone Report, we had a long process of learning and research. We were going to start implementation for the first time, and we had to decide how we would proceed. To be able to make more sensible decisions, we researched many different technologies that are suitable for our project, proposed them in a meeting, and discussed pros and cons. We decided to use PostgreSQL as our database, Django as our web framework. After these were set, we gave ourselves a period to study, learn and practice PostgreSQL, Django, Django REST Framework and Docker. In this time period, we also individually researched some remote APIs that we will be using in our practice apps.

We have each created our practice apps individually. Everybody used at least one remote API of their choice, added a functionality and provided their own API with POST and GET methods. Every group member wrote unit tests and provided a frontend. We pushed our individual codes to different branches. We assigned each member to review someone else's API. After pushing the code to their own branches, members of the group created pull requests and asked for review. After the review process, and if exists, corrections/modifications are completed, the branches were merged into master one by one. Every member also provided documentation for their code and API.

For the web app, we wrote a Dockerfile and built it within docker-compose.We used the Postgres image on DockerHub as our database docker. Using docker, we packed our code into a container and isolated it. We used docker-compose to create two different services for django and for the database. We also dockerized the django application to deploy it safely and quickly. Then we researched the scope of the EC2 free tier. We created an account and created a free EC2 instance. At first, we ran the project in different containers. Then we figured out we should run as seperate services using docker-compose. We did so, and the result was successful.

Moving forward, we will decide who will focus on the frontend, backend and android. We plan to decide these issues before the term ends and have a productive summer where we can learn the technologies decided on better and practice our skills in these areas. This way, we will make a quick start to the next term!

1.2 URI of the Deployed Application

http://3.127.142.97:8000/

Follow these steps to test the application:

To run a practice-app, you install env and some libraries. Then, you need to migrate to build a database. You should run below commands accordingly.

```
python3 -m venv env
source env/bin/activate
pip install django
pip install python-decouple
pip install djangorestframework
pip install psycopg2
pip install psycopg2-binary
pip install requests
python manage.py makemigrations
python manage.py migrate
python manage.py runserver
```

To test it use:

```
python manage.py test
```

Or, you can use this instead of the above:

```
docker-compose up
```

1.3 API URL

Gözde Ünver:

- http://3.127.142.97:8000/weatherCondition/api/

Berkay Bör:

- http://3.127.142.97:8000/airQuality/api/pollution/
- http://3.127.142.97:8000/airQuality/api/locations/

Mehmet Fatih Aksoy:

- http://3.127.142.97:8000/core/api/

Zuhal Didem Aytaç:

- http://3.127.142.97:8000/dailyQuote/api/

Muhammed Furkan Atasoy:

- http://3.127.142.97:8000/sportRelation/api/sports/1
- http://3.127.142.97:8000/sportRelation/api/similar/1
- http://3.127.142.97:8000/sportRelation/api/suggest/1-4-5

Umut Gün:

- http://3.127.142.97:8000/exchangeRateAPI

Kemal Çağın Sertkaya:

- http://3.127.142.97:8000/musicapi

Ozan Kılıç:

- http://3.127.142.97:8000/randomActivity/

1.4 Basic Functionality of the Project

Our project has a home page that has buttons to go to different pages where different APIs with different functionalities are used. These functionalities are weather condition of a given town with the result of the previous search, daily sport quotes are returned and also rating functionality is provided to the returned quotes, registration functionality and getting a random person, searching NBA player, returning information about a random activity, a list of sport songs, getting air quality of a location, learning similarities between different sports and exchange rate functionality. When the http://3.127.142.97:8000/ is clicked, the results of the individual APIs are displayed with the help of html files. Our project also works as an API service. With the appropriate URIs of each functionality, one can retrieve requested data from these functionalities.

weatherCondition API:

It returns the weather condition of the searched town with the data of the previous search. Country, town, x and y coordinations, description of the weather, pressure, temperature, humidity, speed and angle of the wind is provided.

randomActivity Generator API:

Random activity generator API is an application programming interface which shows different activities every time it asked. It generates different types of activities with different number of people. Target user of the API is bored people as it is written at the top of the html page. For generating random activities, as a third party API, boredAPI (https://www.boredapi.com/) is used.

airQuality API:

This is a simple API which returns air quality index and corresponding risk category for a specified location. If provided with a user_id it can store locations for a user and later use them to return air quality data.

This API uses IQ Air AirVisual API to get air quality index data.

Core API:

In this API; you can add user, log in and log out and see how many registered users are there in that API and also you can get a random person's name, gender and age.

You can access my API from main link : http://127.0.0.1:8000/authentication/

DailyQuote API:

The Daily Quote API is an API that provides you daily quotes about sports, in English. It uses the Quotes Rest API (https://quotes.rest/) as the remote API. The remote API provides a different quote for each date (quote of the day functionality). The Daily Quote API filters the quote category as sports and the quote language as English by making a get request to a remote API.Each day's quote, author and date are saved to the database if it has not been saved before. The Daily Quote API adds a rating functionality to the remote API. It supports GET and POST methods. When a user makes a GET request to the API, they see the average rating of the quote and the number of ratings made so far, along with the quote and the author.

ExchangeRate API:

In the API, you can add category, get category by name, get all categories, add event post, get unique event post by post id, get all event posts. In addition to these, the API is also connected to another API which is Open Exchange Rates API which provides all currencies rates based on USD continuously. You can see 1 USD how much others currencies like TRY, CNY, RUB Also, you can convert to any amount USD to others currencies. For example, 100 USD convert to TRY.

You can access to all feates from main link which is http://127.0.0.1:8000/exchangeRateAPI/

MusicAPI:

Musicapi provides the song names of a particular sport playlist. Music API is developed with Django framework and it only supports GET method.

nbaStats API:

You can search for NBA players by their name (and/or surname) and you can get some information about them. API is developed with Django Framework. Also API uses 3rd party API [balldontlie].(https://www.balldontlie.io/api/v1/players)

sportRelationAPI:

Sport Relation API is an API to learn similarities between different sports and take a suggestion from API.

API is developed with Django Framework, and it uses PostgreSQL DBMS. Also, It uses Decathlon API.

There are three capabilities of API.

1. You can fetch a detailed sport information with a sport pk

2. You can learn top 5 similar sports to a sport by giving a sport pk
3. You can get a suggestion from the API to learn which sport is the most relevant sport for you by giving three sports you are interested in the format pk1-pk2-pk3.

API has a user interface which was developed with help of html templates rendering by Django. You can interact with API with help of UI.

1.5 Challenges We Met As A Group

Even though we are in an era where everyone lives online, it was difficult to adapt meeting times to everyone since we were 9 people.

Sometimes we experienced communication breakdowns in meetings due to general internet connection problems.

As in any large project group, it sometimes took a while to reach consensus on some issue. Although we held meetings with the assistant as much as possible, sometimes we could not get accurate and clear answers from the assistant.

Sometimes we couldn't make the distribution of tasks fairly on the subjects that we do not have much knowledge about.

We had difficulties in understanding the general scoring strategy of the course (importance of the issue creating on git, etc.).

It was difficult to keep all 9 people up with the deadlines of different tasks.

It was hard to catch up on all tasks we have in Milestone 2 and also, jobs we have to do in other classes.

It was hard to both writing Milestone and code at the limited time (I think timing was not enough)

It's hard to find a free useful API to integrate my API while creating a branch and committing, something happened which I don't understand. Although every terminal code is correct, I cannot create branches. After the computer and terminal reopen, it's fixed.

1.6 List and Status of The Deliverables

Deliverable	Completion date	Status
Code Documentation	09.06.2021	Done
API documentation	09.06.2021	Done
Practice-app	08.06.2021	Done
URI of Our Deployed Application	09.06.2021	Done
API URL	09.06.2021	Done
Dockerizing API	09.06.2021	Done

1.6.1 Code Documentation

In this milestone report, in addition to the descriptions of our API we also added their code documentation. In the code documentation we mainly explained the views.py of our APIs because those files did the main job. In the code documentation we explained how the functions work in detail. If the API documentation is not explanatory enough for a user, code documentation will help to understand the functionality of the API better.

1.6.2 API Documentation

API documentation contains functions, classes, arguments etc. It helps to coders and users to understand APIs how it uses and how it installs. In the next semester, the API documentations will help us to understand the functionality of our APIs. Therefore, it will be a guide for us and we can integrate the functionalities into the new project.

1.6.3 Practice-App

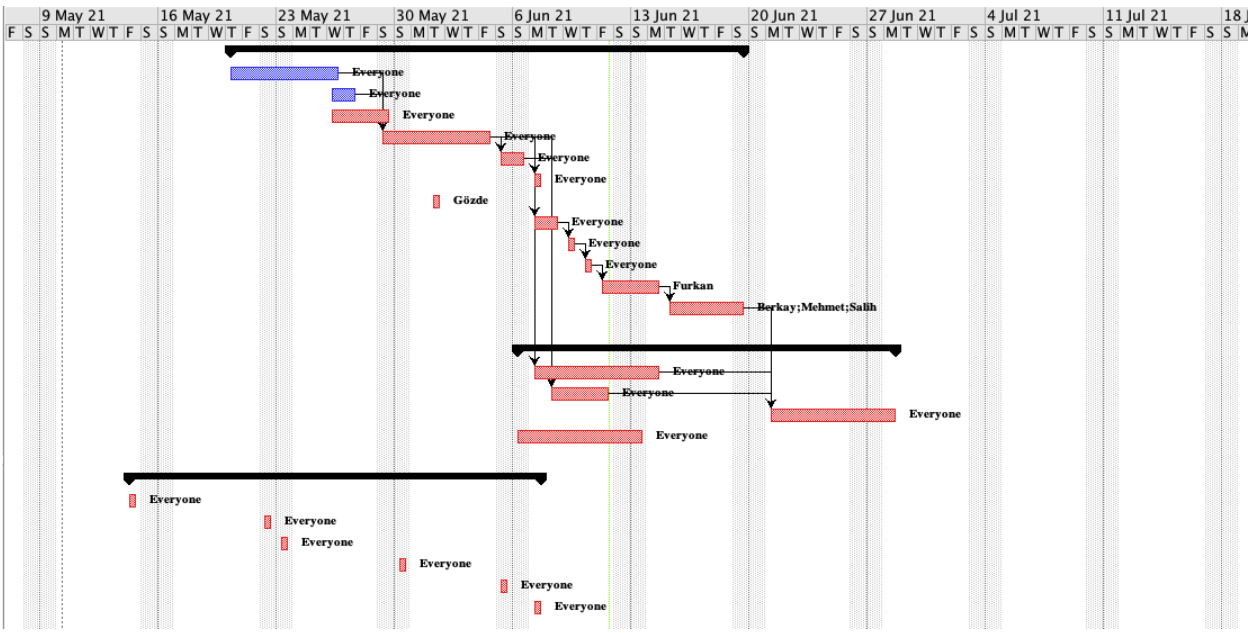
There were three task on practice-app which were front-end, back-end and database. Firstly, we met and decided which APIs were used. Each team member decided external API and endpoints. While implementing code, we used Postgresql for database. Everyone wrote their API codes and unit test on their own branches. After finished APIs, we reviewed each other APIs and gave feedbacks each others. If there was any problem, we fixed our problems. Then, pull requested were accepted by reviewed member and the code was merged successfully. Finally, the code was dockerized and deployed to AWS.

1.6.4 Dockerizing API

API should be built with docker before the deployment to AWS. Docker creates containers for applications and with help of that, an API can be moved to different platforms without any problem. The dockerfile contains how to build the app, which packages it uses, and etc. Also, the project should have a docker-compose file as well. Because it has a database and with the help of docker-compose, database and app can be in the same server but in different containers.

2 PROJECT PLAN

	Name	Duration	Start	Finish	Predecessors	Resource Names
1	**API**	22 days?	**5/20/21 8:00 AM**	**6/19/21 5:00 PM**		
2	Learning Django, Django Rest framework, PostgreSQL, Docker	7 days?	5/20/21 8:00 AM	5/26/21 5:00 PM		Everyone
3	Learning and creating branches	2 days?	5/26/21 8:00 AM	5/27/21 5:00 PM		Everyone
4	Searching API	4 days?	5/26/21 8:00 AM	5/29/21 5:00 PM		Everyone
5	Implementing own APIs	7 days?	5/29/21 8:00 AM	6/4/21 5:00 PM	2;3	Everyone
6	Implementing unit tests	2 days?	6/5/21 8:00 AM	6/6/21 5:00 PM	5	Everyone
7	Creating ReadMe file	1 day?	6/7/21 8:00 AM	6/7/21 5:00 PM	6	Everyone
8	Creating home module	1 day?	6/1/21 8:00 AM	6/1/21 5:00 PM		Gözde
9	API review and feedback	2 days?	6/7/21 8:00 AM	6/8/21 5:00 PM	5;6	Everyone
10	Fix problems	1 day?	6/9/21 8:00 AM	6/9/21 5:00 PM	9	Everyone
11	Merging APIs	1 day?	6/10/21 8:00 AM	6/10/21 5:00 PM	10	Everyone
12	Dockerization	4 days?	6/11/21 8:00 AM	6/14/21 5:00 PM	11	Furkan
13	Database connection and AWS	4 days?	6/15/21 8:00 AM	6/19/21 5:00 PM	12	Berkay;Mehmet;Salih
15	**Documentation**	16 days?	**6/6/21 8:00 AM**	**6/28/21 5:00 PM**		
16	Creating code documentation	8 days?	6/7/21 8:00 AM	6/14/21 5:00 PM	5;6	Everyone
17	Creating API documentation	4 days?	6/8/21 8:00 AM	6/11/21 5:00 PM	5;6	Everyone
18	Creating milestone2	8 days?	6/21/21 8:00 AM	6/28/21 5:00 PM	13;16;17	Everyone
19	Creating individual report	8 days?	6/6/21 8:00 AM	6/13/21 5:00 PM		Everyone
21	**Meetings**	17 days?	**5/14/21 8:00 AM**	**6/7/21 5:00 PM**		
22	Taking meeting notes (#12)	1 day?	5/14/21 8:00 AM	5/14/21 5:00 PM		Everyone
23	Taking meeting notes (#13)	1 day?	5/22/21 8:00 AM	5/22/21 5:00 PM		Everyone
24	Taking meeting notes (#14)	1 day?	5/23/21 8:00 AM	5/23/21 5:00 PM		Everyone
25	Taking meeting notes (#15)	1 day?	5/30/21 8:00 AM	5/30/21 5:00 PM		Everyone
26	Taking meeting notes (#16)	1 day?	6/5/21 8:00 AM	6/5/21 5:00 PM		Everyone
27	Taking meeting notes (#17)	1 day?	6/7/21 8:00 AM	6/7/21 5:00 PM		Everyone



3 WORK DONE BY EACH MEMBER

Gözde Ünver	- I studied Django, Django REST framework, PostgreSQL, Flask, Docker, how to use postman and how to write unit tests. Issue is here: https://github.com/bounswe/2021SpringGroup5/issues/116 - I created an empty Django project so that everyone can pull from it and work on their branches. Issue is here: https://github.com/bounswe/2021SpringGroup5/issues/123 - I wrote a weather condition API using this site: https://openweathermap.org/ and the

	related information can be found in this issue: https://github.com/bounswe/2021SpringGroup5/issues/122. weatherCondition module in our project both acts like an API server and also provides a UI via its html files. I also wrote its API documentation. I wrote unit tests for my API: https://github.com/bounswe/2021SpringGroup5/issues/129 - I made a pull request about my weatherCondition API. I made modifications in my code according to the feedbacks I received. Related information can be seen here: https://github.com/bounswe/2021SpringGroup5/pull/131 - I created the home module for our team API. I created the html file that creates buttons to go different functionalities that we provide in our project. I linked the urls of all other modules to the home module and added buttons for those modules in the html file. Related information can be found here: https://github.com/bounswe/2021SpringGroup5/issues/130 - I made pull requests for the home connection of our team API. It can be seen here: https://github.com/bounswe/2021SpringGroup5/pull/168 , https://github.com/bounswe/2021SpringGroup5/pull/182 , https://github.com/bounswe/2021SpringGroup5/pull/190 https://github.com/bounswe/2021SpringGroup5/pull/190 - I reviewed Kemal's code and gave feedbacks. Related information can be found here: https://github.com/bounswe/2021SpringGroup5/pull/171 , https://github.com/bounswe/2021SpringGroup5/pull/169 - I reviewed Mehmet Fatih's code: Issue is here: https://github.com/bounswe/2021SpringGroup5/issues/188 - I reviewed Furkan's modifications on home app: https://github.com/bounswe/2021SpringGroup5/pull/207 - I wrote a table of deliverables, explanation of the code documentation deliverable and my weatherCondition API code documentation. Issue is here: https://github.com/bounswe/2021SpringGroup5/issues/192 - I reviewed first attempts of deploying the project. Issue is here: https://github.com/bounswe/2021SpringGroup5/issues/174 - Created v.0.1 tag. Issue is here: https://github.com/bounswe/2021SpringGroup5/issues/220
Muhammet Furkan Atasoy	-Searched for API which will be interacted in the project and shared it with my group on Discord. I decided to Decathlon Sport API. Issue: #119 - Created Sport Relation API for practice-app. Pushed to atasoybranch which can be found in: https://github.com/bounswe/2021SpringGroup5/tree/atasoybranch . Issue: #120

	- Learned Django with help of official Django Documentation. Issue: #121 - Learned Django REST framework with help of official documentation. Issue: #121 -Learned PostgreSQL integration and usage in desktop with pdAdmin4. Issue: #121 - Published Meeting Notes #14. Can be accessible in here: Meeting Notes #14 Issue: #127 - Reviewed Home page for practice-app. Pul request is made by Gozde Unver, and I was reviewer. Found some .gitignore issues and reported to Gozde Unver. Pull request can be found in here: https://github.com/bounswe/2021SpringGroup5/pull/190. Issue: #194 -Created README file for sport_relation app in order to make reviewer Didem Aytac work easy. Can be found in: https://github.com/bounswe/2021SpringGroup5/tree/master/practice-app/group5_api/sport_relation#readme. Issue: #180 - Wrote unit tests for sport_relation API. Django test library is used for the task. Can be found in: https://github.com/bounswe/2021SpringGroup5/blob/master/practice-app/group5_api/sport_relation/tests.py. Issue: #196 -Created individual Milestone 2 report, and added to Moodle. Issue: #197 - Wrote Personal Efforts with help of issues I created on Github. Can be found in Milestone 2. Issue: #198 -Reviewed Berkaybranch. Read the code, cloned the repository to my local, run it, and test it via Postman and also with unit tests he wrote. Also, made some suggestions for Berkay Bor which was the owner of pull request. Pull request can be found in: https://github.com/bounswe/2021SpringGroup5/pull/167. Issue: #199 -Wrote API Documentation for sport_relation app with help of Microsoft Word. Didem Aytac is reviewed the API documentation. The resulting API documentation can be seen at: https://github.com/bounswe/2021SpringGroup5/blob/master/practice-app/group5_api/sport_relation/Sport_Relation_Code_and_API_Documentation.pdf. Issue: #201 -Wrote Code Documentation for sport_relation app with help of Microsoft Word. Didem Aytac is reviewed the API documentation. The resulting code documentation can be seen at: https://github.com/bounswe/2021SpringGroup5/blob/master/practice-app/group5_api/sport_relation/Sport_Relation_Code_and_API_Documentation.pdf. Issue: #201 -Wrote weekly efforts week by week to wiki page. Can be found in: https://github.com/bounswe/2021SpringGroup5/wiki/Muhammed-Furkan-Atasoy-Weekly-Efforts. Issue: #203 -Created pull request, resolved conflicts and merged sport_relation to master branch. Can be found in: https://github.com/bounswe/2021SpringGroup5/pull/176. Issue: #205 -Created pull request, resolved several bugs and conflicts after merged home branch to master branch. Can be found in: https://github.com/bounswe/2021SpringGroup5/pull/207. Issue: #206 -Added UI to Sport Relation API with help of Django templates. Can be found in:

	https://github.com/bounswe/2021SpringGroup5/tree/master/practice-app/group5_api/sport_relation/templates/sport_relation Issue: #208 -Learned Docker from Docker official tutorial in their website. Write dockerfile and docker-compose for practice-app with help of Berkay Bor. Can be found in: https://github.com/bounswe/2021SpringGroup5/tree/master/practice-app/group5_api Issue: #209 -Updated .gitignore to cover pycache and data came from docker. Can be found in: https://github.com/bounswe/2021SpringGroup5/blob/master/.gitignore. Issue: #212 -Reviewed the evaluation of tools and processed part which made by Didem Aytac. Added Visual Studio Code to report. Can be found in Milestone 2. Issue: #170
Umut Gün	- I learned about Django, Rest API, PostgreSQL, Docker #132 - I learned postman to use get and post methods - I created my own braches and pull the master project #136 - I wrote basic app. I created post category method, get all category, get category by name, post event post, get all event post, get event post by post id. #137 - I wrote an excange rate API using this site: https://openexchangerates.org/ . My API gets currencies rate based on USD and offers to users to exchange their amount money based on USD to another currency like TRY, CNY, RUB. Also, this exchange progess is saved on database #137 - I wrote unit tests to test API - I wrote Readme file to explain my API and how to use it #187 - I sent pull request to merged my API into origin master #149 - I got some feedbacks from Salih and I fixed them #159 - I reviewed Didem's API and I gave her feedback #143 - I pull origin master after everyone merged their APIs. I haven't faced with any issue with my API. #214 - I checked dockerization - I wrote the status of deliverables #216 - I created project plan and gantt chart #215
Mehmet Fatih Aksoy	- I learned about Django, Rest API, PostgreSQL, Docker - I am registered to AWS and learned about AWS EC2 - I created pull requests for my code, there were bugs but I fixed it. - I wrote authentication API that allows you to sign up and log in using your credentials.

	- I Created aksoybranch and implemented my API - I got feedbacks from Gözde and fixed some issues related to my branch. - I accompanied Berkay to install AWS EC2 instance. - I contributed milestone 2 report. - I took notes during Meeting. - I realized we must take an API from web and created another function random name: It takes data from https://randomuser.me/api/ and prints random name,gender and age. - I wrote unit tests for my API. - I wrote readme file for API. - I review dockerization.
Salih Bedirhan Eker	- I learned about Django, Django REST Framework, PostgreSQL, Docker - I wrote an API using tools mentioned above. It is in nbaStats directory in out application. Also I used a third party API https://www.balldontlie.io/api/v1/players. - I provide tests for my API. - I provide a simple interface using html. Html files are in nbaStats/templates directory. - I learned how to write API documentation and I provide an API documentation in README.md file in nbaStats/README.md. - I learned how to make a pull request an I made a pull request on github. Also I configure my pull Request. - I reviewed and an API code and tests. Also I commented a pull request and I gave feedback to Umut. - Also after the necessary modifications are made, I merged that pull request into our master branch. - I opened an AWS EC2 instance and run it. - I reviewed our project requirements and I made a small correction on it. - Me, Berkay and Fatih deployed our projects first attempt on that EC2 instance. - We handled docker containers and PostgreSQL work on EC2 instance. - We deployed our project on EC2 instance - After deployement I reviewed the work we have done.
Kemal Çağın	-I learned Django, Rest Framework and PostgreSQL. issue:

Sertkaya	https://github.com/bounswe/2021SpringGroup5/issues/134 -I wrote MusicAPI for practice-app. It gets the song names of a particular sport playlist on spotify. -I created my own branch and pull it. -I wrote the Readme file of my API to explain how it works. issue: https://github.com/bounswe/2021SpringGroup5/issues/191 -I reviewed Berkay's API and gave him a review. issue: https://github.com/bounswe/2021SpringGroup5/issues/175 -I wrote the unit test for musicAPI. -I sent pull request to merge my musicAPI into master. -I wrote the challenges we met as a group. issue: https://github.com/bounswe/2021SpringGroup5/issues/210 -I wrote meeting notes for tenth and sixteenth meetings which can be found on our wiki page. https://github.com/bounswe/2021SpringGroup5/wiki/Meeting-16--Notes----05.06.2021 and https://github.com/bounswe/2021SpringGroup5/wiki/Meeting-%2310-Notes---1.05.2021
Zuhal Didem Aytaç	- I researched some remote APIs that will help me understand the API logic and also may be useful for the project. Related issue: https://github.com/bounswe/2021SpringGroup5/issues/124 - Before writing my practice API, I learned Django, Django REST Framework, Docker, Flask and Postgres. Related issue: https://github.com/bounswe/2021SpringGroup5/issues/126 - I wrote an API that gives the user the sports quote of the day, that also has a rate feature. Related issue: https://github.com/bounswe/2021SpringGroup5/issues/125 Related page: https://github.com/bounswe/2021SpringGroup5/tree/zudiaybranch - I started a Django project to write my API. - I set up a PostgreSQL database in my computer. I created an application called DailyQuote in the project using the PyCharm interface. I created necessary models and serializers. - I wrote an HTML file that shows the quote and takes rating input from the user. I created necessary views. I implemented POST and GET methods. - I tested the API using Postman and the Web interface. I wrote some unit tests using rest_framework.test. - I pushed the code to the repository to a new branch - I created a pull request and notified Umut to review my code. I talked with Umut, he set up the app in his local and asked some questions. He reported everything works well and there are no issues. We merged the branch to master. Related issue: https://github.com/bounswe/2021SpringGroup5/issues/144 Related page:

	https://github.com/bounswe/2021SpringGroup5/pull/143 - I wrote the 'Summary of Project and Overall Status' and 'Evaluation of Tools and Processes' sections for our group M2 report. Related issue: https://github.com/bounswe/2021SpringGroup5/issues/170 - I reviewed Furkan's practice app, read his code and api documentation, tested it using UI, Postman and run the unit tests. I shared my feedbacks with him. Related issue: https://github.com/bounswe/2021SpringGroup5/issues/147 Related page: https://github.com/bounswe/2021SpringGroup5/pull/176 - I wrote a single & detailed documentation where - I explained the app briefly - I explained the code, classes, serializers, views and functions in detail - I documented the API endpoints, the parameters, expected responses and some example responses. - I documented how to run the app and how to use the user interface - I added some screenshots of the app. - I provided the unit tests. Related issues: https://github.com/bounswe/2021SpringGroup5/issues/178 https://github.com/bounswe/2021SpringGroup5/issues/179 - I added the API documentation to ReadME Related page: https://github.com/bounswe/2021SpringGroup5/blob/master/practice-app/group5_api/dailyQuote/README.md - I made a small simplification in views.py code of the daily quote API. I pushed it to zudiaybranch. I opened a pull request and asked Umut for review. After the review, the code was merged into master. Related issue: https://github.com/bounswe/2021SpringGroup5/issues/144 Related page: https://github.com/bounswe/2021SpringGroup5/pull/18 - I updated the code documentation according to feedbacks after the meeting with TA. It can be found in the daily quote readme or as a pdf. Related issues: https://github.com/bounswe/2021SpringGroup5/issues/178 https://github.com/bounswe/2021SpringGroup5/issues/179 Related pages: https://github.com/bounswe/2021SpringGroup5/blob/master/practice-app/group5_api/dailyQuote/README.md https://github.com/bounswe/2021SpringGroup5/blob/master/practice-app/group5_api/dailyQuote/DAILY%20SPORTS%20QUOTE%20APP.pdf - There has been some issue in the server. We worked on it to solve the problem. I also reviewed the Dockerfile and server to see if there are any problems. Related issues: https://github.com/bounswe/2021SpringGroup5/issues/218 https://github.com/bounswe/2021SpringGroup5/issues/174

Ozan Kılıç	-Required techs are learned. I worked on PostgreSQL, Django, Django Rest Framework, Flask, Docker. Issue link is: https://github.com/bounswe/2021SpringGroup5/issues/158 -API of randomActivity is written. In the process, learned techs are used. In the API, boredAPI is used for getting data. randomActivity API can be reached here: https://github.com/bounswe/2021SpringGroup5/tree/master/practice-app/group5_api/randomActivity -Test is written for the API. Test can be reched from the link: https://github.com/bounswe/2021SpringGroup5/blob/master/practice-app/group5_api/randomActivity/tests.py -For user usage, API needed a html file. A basic html code is added to templates folder. HTML code renders the API result and show it with some corrections. The html file can be reached here:https://github.com/bounswe/2021SpringGroup5/blob/master/practice-app/group5_api/randomActivity/templates/activity/home.html -Personal branch is created to work on randomActivity API seperate from the main project. After adding codes, branch was merged by Mehmet Fatih. The branch can be reached here: https://github.com/bounswe/2021SpringGroup5/tree/ozanbranch -Unit test is required for the API. I added unit test to control if boredAPI returns the asked JSON values. The unit test can be reached here: https://github.com/bounswe/2021SpringGroup5/blob/ozanbranch/practice-app/group5_api/randomActivity/tests.py -To describe what is the API and how to use it, README.md file is added to the folder of randomActivity. It briefly explains details of the API. README.md file can be reached here: https://github.com/bounswe/2021SpringGroup5/blob/master/practice-app/group5_api/randomActivity/README.md -As a responsibility in the group every member should review another one's pull request of him/her API. So, I reviewed the API of Salilh Bedirhan's nbaStats. After checking the documents and code merge request accepted. Link of the issue is: https://github.com/bounswe/2021SpringGroup5/pull/154 and https://github.com/bounswe/2021SpringGroup5/pull/184 -Dockerization is made by Furkan Atasoy and I follow his steps. -I gathered the informations for Milestone 2 and prepared the Milestone PDF. PDF can be reached from here: https://github.com/bounswe/2021SpringGroup5/tree/master/deliverables Also I will submit the PDF to Moodle.

Berkay Bör	- I learned and used Django, Django REST Framework. https://github.com/bounswe/2021SpringGroup5/issues/128 - I attended Docker PS and learned Docker. https://github.com/bounswe/2021SpringGroup5/issues/128 - I created a branch named 'berkaybranch' to implement my API. https://github.com/bounswe/2021SpringGroup5/tree/berkaybranch - I wrote a basic API called AirQualityAPI. It takes air quality data using coordinates from AirVisual API (https://www.iqair.com/air-pollution-data-api) and correctly categorizes the air quality and adds description to it. - I created a pull request and waited for my code to be reviewed. https://github.com/bounswe/2021SpringGroup5/pull/167 - I modified my code according to code review. https://github.com/bounswe/2021SpringGroup5/pull/167 - I merged my branch to the main branch. https://github.com/bounswe/2021SpringGroup5/pull/167 - I wrote a basic readme file for the API. https://github.com/bounswe/2021SpringGroup5/issues/128 - I added my button and modified url configirations to make my API reachable from home. - I reviewed Gözde's code. https://github.com/bounswe/2021SpringGroup5/issues/151 - I installed Docker and docker-compose on AWS EC2 instance. https://github.com/bounswe/2021SpringGroup5/issues/174 - I cloned from main and started django and postgres services on AWS. I used docker-compose and I wrote a simple dockerfile and docker-compose.yml to test the app https://github.com/bounswe/2021SpringGroup5/issues/174 - I contributed milestone 2 deliverables report. https://github.com/bounswe/2021SpringGroup5/issues/219 - I reviewed and contributed to Summary of Project and Overall Status report, modified Docker part. https://github.com/bounswe/2021SpringGroup5/issues/170 - I reviewed and tested the modifications on Dockerfile and docker-compose.yml files made by Furkan. https://github.com/bounswe/2021SpringGroup5/pull/211

4 EVALUATION OF TOOLS AND PROCESSES

4.1 WhatsApp

We created a WhatsApp group to communicate on emergency, when we need to inform other team members about meetings, works that need to be done at the moment

4.2 Discord

Discord is a platform to communicate via voice, video and text. We used it for meetings,file sharing and communication

4.3 GitHub

GitHub is the key tool of this project. We use it to document our weekly efforts, meeting notes, project requirements, scenarios, and research. We use Git issues to assign works and track them. And now with the first implementation tasks, we used the version control system and branches. We made great use of pull requests. When somebody finished their part for the app, they pushed to individual branches. Then, the pusher opened a pull request and asked for review via the pull request. This way we tracked the merges in a controlled way.

4.4 Google Docs and Sheets

We used Google docs & sheets to share our ideas and prepare drafts. We used these Tools especially when we brainstormed during meetings altogether

4.5 Docker

Docker: Docker is a widely used tool and we used it to containerize our application. Using docker, we packed our code into a container and isolated it. We used docker-compose to create two different services for django and for the database. We also dockerized the django application to deploy it safely and quickly.

4.6 Python

We used Python to implement our practice apps. It has many libraries and is easy to code.

4.7 Html & CSS

We were asked to create a front end for an end user for our practice apps. A simple interface was sufficient. So, we used Html and some of us made it more stylish with CSS and some other libraries.

4.8 Django & Django REST Framework

Django is a web application platform, one of the most popular ones. We chose to use it because it is easy to use and scalable. It was working with the database we chose, PostgreSQL. It provided an easy way to create multiple apps in one project, which was very useful in the practice-app task. REST framework allows us to send and receive data between the database and user interface. We believe it's a good choice and will be nice for the rest.

4.9 PostgreSQL

We chose to use PostgreSQL as our database as it's widely used and open source. It is secure, extensible. We believe it's a good choice and will be nice for the rest. Also we are glad we chose PostgreSQL because it is widely used.

4.10 PyCharm

Most of use used PyCharm as their IDE while developing the practice apps. PyCharm manages the tasks of creating the structure and files when a Django app is created.

4.11 Visual Studio Code

Visual Studio Code is a code editor with a lot of capabilities which are debugging, integrated terminal, formatting code, and git interface. Also, It has a wide community, and useful extensions. We used it for these reasons.

4.12 Postman

Postman allows us to test the RESTful APIs we have developed on our local. We used it to see if our APIs work fine and test the functionality. Postman allows us to see the responses clearly and easily.

4.13 AWS EC2

EC2 is a service of AWS. We created an instance and deployed our application to the server. We encountered some issues deciding on how to deploy the database and application, separately or together. We also had some problems after deployment, some apps were not working, the logs were so messy and we could not figure out the problem. Then we restarted and the issues were resolved.

4.14 Project Libre

We used ProjectLibre for our project plan for the report. It is challenging to use and not user friendly. For example, it does not allow you to complete a task on weekends. Also, it is hard to figure out how to do things

5 DELIVERABLES

5.1 Code Documentations

5.1.1 weatherCondition Code Documentation:

In the views.py file of the weatherCondition app, there are functions named: def weather(method,searchTown=None), def showWeather(request) and def weather_api(request). “weather” function sends requests to openweathermap api, retrieves information, checks the data and returns the current and the previous search data with appropriate status code as a json response to be fetched by either showWeather or weather_api function. In other words, showWeather and weather_api functions are the functions that fetch the requests and call weather function to do the job with type of the request and the town name if it is a post request and then simply return the response in requested form. showWeather returns the response as an html rendered with the response of weather function so showWeather function is used for the UI usage of the api. weather_api simply returns the the response of weather function and weather_api is used for retrieving json response.

weather function takes the type of the request method and searched town name. If a town name is not given, it is None as default (This is for the get request because when there is a get request, no town name is provided). First the function checks whether it is a get method. If it is a get method then a json response with empty dictionary and status code 200 is returned. If it is a post request, then initially the data of the previous search is retrieved from the database. If there is, the data is stored in the dictionary called “dictionaryOLD” via WeatherConditionSerializer. If there is no previous search, this dictionary is simply empty.

Then a request is made to the openweathermap api using the weather key that is hidden in the .env file. If the town name is a real town name then this api actually returns useful data and this is checked by the status code of the response. If the status code is not 200 then the “dictionaryNew” stores empty dictionary and the return status code is set to 404. If the status code is 200, the data is further processed. Data returned from the openweathermap api is stored in dictionaryNew and the validity of this data is checked via WeatherConditionSerializer. If the data fits the limits of the model (char length etc.) then a WeatherCondition object is created, saved into the database and the response code is determined as 201, else the object is not even created, dictionaryNew is emptied and the status code is stored as 400. Finally, the final data

which is called “dictionary” to be sent as the response is created using dictionaryOLD and dictionaryNew. The response object is created with its data as dictionary and status code that is determined earlier in the code. This response object is returned.

5.1.2 Core Code Documentation

In the views.py file of Core app, there are three functions named: def home(request), def signup(request), def showRandom(request).

Home function holds the number of registered users to the system and renders the home.html file, called homepage. In the homepage you can see log in, sign up, log out and random person links. You can’t see the log out link if you don’t login to the system. Signup function requests ‘UserCreationForm’ from ‘django.contrib.auth.forms’ and this form takes username and password for signing up to the system. After the password verification by entering the same password and username verification checking that the username already exists or not, this function registers the user to the system. If the form is not valid, the user sees the same form again.

The ShowRandom function is the function I created to take an API from somewhere. This function requests to https://randomuser.me/api/ and takes the tandom name, surname, gender and age. There are many other features like location, email, phone number but I chose the features above. The response from the URL is a json file and this function renders the random.html and sends the taken values to that html file. Random.html shows us the name, surname, age and gender of a random person. You can still see the other links like signup in that html file. There is no need to be registered to the system to use the showrandom function.

5.1.3 airQuality Code Documentation

get_pollution(request) is the main view of the API. It has only GET functionality. If ‘lat’ and ‘lon’ arguments are given, it calls AirVisual API and gets AQI from given coordinates. If user_id is given instead, it calls AirVisual API and gets AQI from given user’s last saved location. If user has no location history, it returns HTTP_404_NOT_FOUND. After getting the AQI, it categorizes it using AQI EPA standards and adds a name, a color and a description. It also adds recording datetime, city, state and country of given location and coordinates to the response and returns the response.

location_list(request) has GET and POST functionality. If the request is GET it returns all saved locations for all users. If the request is GET and user_id argument is provided, it only returns locations for that user. If user is not found it returns 404 error. If the request is POST, it checks if user_id, latitiude and longitude values are valid and if valid it saves them to the database. If not it

returns 400 error. View_page(request) is a basic view that returns the home page for user interface.

5.1.4 Musicapi Code Documentation

In general, this api returns a specific Spotify playlist's song names in a column.
In the views.py file, there are functions named musicapi and get_music.
musicapi function renders a basic home.html page. There is a text named "Fetch Music" in the homepage and it shows the list of songs when clicked. get_music function sends request to api.spotify.com and this url returns a json file and get_music function renders music.html.

```python
import requests
from django.shortcuts import render
from django.conf import settings
from rest_framework.views import APIView
from rest_framework.response import Response


def musicapi(request):
    return render(request, "musicapi/home.html")


def get_music(request):
    url = "https://api.spotify.com/v1/playlists/{}/tracks".format(
        settings.PLAYLIST_ID)
    headers = settings.TOKEN
    info = requests.get(url, headers=headers).json
    context = {
        "info": info
    }
    return render(request, "musicapi/music.html", context)


class GetMusicView(APIView):
    def get(self, request):
        url = "https://api.spotify.com/v1/playlists/{}/tracks".format(
            settings.PLAYLIST_ID)
        headers = settings.TOKEN
        r = requests.get(url, headers=headers)
        info = r.content
        json_response = r.json()
        repository = json_response['items']
        list = []
        for i in repository:
            list.append(i['track']['name'])
        return Response(list)
```

There should be two pieces of information which are PLAYLIST_ID and TOKEN. The PLAYLIST_ID is simply accessible by copying a Spotify playlist's url and selecting a particular part. For instance:
https://open.spotify.com/playlist/4JhDvULKIy5CXyzJRufHLt?si=14e994f8c07a4dc4
The underlined part is the PLAYLIST_ID. Therefore, OAuth Token can be reachable on https://developer.spotify.com/console/get-playlist-tracks/

After filling blanks with the "Fill Sample Data" button and arranging the limit as the number of the songs in the playlist, the OAuth token can easily be obtained by clicking the "Get Token" button. It will verify the user and give a 1-hour-valid OAuth token. I tried to extend this time

frame but I couldn't find any solution. Because of that my api returns the song names after I write a new token generated within 1 hour.

5.1.5 exchangeRateAPI Code Documentation

The API consists of two main API. First one consists of an internal API which serves posting and getting categories and event posts. The second one consists of an external API which gets exchange rates based on USD from https://openexchangerates.org/

I collected all functionalities on a templates/index.html page and views.py. I wrote codes part by part as a method on views.py. I used urls.py to manage URLS for get and post methods. In models.py, I created database diagrams like tables, fields and their restrictions.

5.1.6 nbaStats Code Documentation:

views.py

views.py class is used to handle api requests. It has one function stats_api(request) and the function runs in case of GET and POST requests.

Function has one parameter request. If request is GET function renders to base page. If request is POST, method process the request. First handle the status information. Status are explained in tests.py part. Then function creates a Player() object and creates players as needed.
In return part this function renders the player.html page with the status information.

```python
'''
    Author: Salih Bedirhan Eker
    Date: 07.06.2021
'''

from django.shortcuts import render
from django.http import HttpResponse
from rest_framework.decorators import api_view
import requests

from .models import Player


@api_view(['GET', 'POST'])
def stats_api(request):

    if request.method == 'GET':              # get method renders base page
        return render(request, 'nbaStats/base.html')

    if request.method == 'POST':            # post method renders player page
        if not request.data['player_name']:
            return HttpResponse(status=400)
        url = 'https://www.balldontlie.io/api/v1/players?search=' + \
            request.data['player_name']

        # gets the data and convert it into json
        try:
            data = requests.get(url).json()
        except requests.ConnectionError:
            return HttpResponse(status=503)

        match = True                        # is there any player with this name
        status = 200                                # true by default

        if len(data['data']) == 0:          # if there is no player
            match = False
            status = 404

        players = []                          # player list that will be send
        # for every player arrange their information
        for i in range(len(data['data'])):
            player = Player()
            player.first_name = data['data'][i]['first_name']
            player.last_name = data['data'][i]['last_name']
            player.position = data['data'][i]['position']
            player.team = data['data'][i]['team']['full_name']
            player.height = data['data'][i]['height_feet']
            player.weight = data['data'][i]['weight_pounds']
            players.append(player)

        # send the infomation and match flag
        return render(request, 'nbaStats/player.html', {'players': players, 'match': match}, status=status)
```

tests.py

In this class there are three test cases are provided.

test_no_match:

Test case is written in case there is no information provided on the interface. Status code is set HTTP_400_BAD_REQUEST because request is not properly handled. In order to API works there is a player name needed and when player name is not provided it cause an error.

test_valid_player_name:

Test case is written in order to test expected cases. When program has a proper player name information. Program search for the players information and returns it. It means all structure works well.

test_invalid_input:

Test case is written in case of player name is meaningless and there is nothing to show for the application. Request works fine but there is not matching answer for that request. Hence status code is set HTTP_400_NOT_FOUND

```python
from rest_framework import status
from rest_framework.test import APITestCase, APIClient


class nbaStats(APITestCase):

    # Tests if there is no player in this name
    def test_no_match(self):
        data = {"player_name": ""}
        client = APIClient()
        response = client.post("/nbaStats/", data)
        self.assertEqual(response.status_code, status.HTTP_400_BAD_REQUEST)

    # Tests if player name is valid and there will be match(es)
    def test_valid_player_name(self):
        data = {"player_name": "Davis"}
        client = APIClient()
        response = client.post("/nbaStats/", data)
        self.assertEqual(response.status_code, status.HTTP_200_OK)

    # Tests if player name is meaningles and there is no match
    def test_valid_input(self):
        data = {"player_name": "asdasdasdasdasdasdasd"}
        client = APIClient()
        response = client.post("/nbaStats/", data)
        self.assertEqual(response.status_code, status.HTTP_404_NOT_FOUND)
```

models.py

This class is written for saving the information of players that are searched for and this information will display the interface.

```python
from django.db import models

# Create your models here.


class Player:           #holds information of one player
    first_name: str
    last_name:str
    position:str
    team:str
    height:str
    weight:str
```

5.1.7 Daily Quote API Code Documentation

models.py

```
class DailyQuote(models.Model):
        quote_text = models.CharField(max_length=1000)  # quote text
        author = models.CharField(max_length=100)          # author of the quote
        date = models.DateField(unique=True)               # date of the quote
        points = models.IntegerField()                     # total points given
        ratings = models.IntegerField()                 # total ratings made
```

DailyQuote is a django.db model that the app uses. The meanings of the fields are explained in the comments. The model is registered in admin.py

serializer.py

```
class DailyQuoteSerializer(serializers.ModelSerializer):
        class Meta:
        model = DailyQuote
        fields = ['quote_text', 'author', 'date', 'points', 'ratings']
```

DailyQuoteSerializer is a serializer (inheriting from rest_framework.serializers) for the DailyQuote class.

views.py

```
@api_view(['GET', 'POST'])
def show_quote(request):
        if request.method == 'GET':
        res = quote().data
        return render(request, 'dailyQuote/base.html', res)
        else:
        res = rate(request.POST.get("Points", '')).data
        return render(request, 'dailyQuote/base.html', res)
```

```
@api_view(['GET', 'POST'])
def quote_api(request):
        if request.method == 'GET':
        res = quote()
        return res
        else:
        res = rate(request.POST.get("Points", ''))
        return res
```

There are two api_views. Both supports GET and POST methods and provide the same functionality. The only difference is that the show_quote function renders the response data with the HTML, whereas the quote_api function provides a plain api response. The GET method calls the quote() function and the POST method calls the rate() function with the input of “Points” in the request body. These functions are explained below.

Below, the quote() function is given in 2 parts. The quote function basically returns a response with a body and a status. It is the code that runs when a GET request is made.
views.py

```
def quote():
        today = str(datetime.utcnow().date())
        points, ratings, value = 0, 0 , 0

        # get the quote with todays date from the database, if exists
        try:
        quote_in_db = DailyQuoteSerializer(DailyQuote.objects.filter(date=today)[0]).data
        except:
        quote_in_db = None

        # if the quote of today has been saved before, take the points and ratings
        if quote_in_db is not None:
        points = quote_in_db["points"]
        ratings = quote_in_db["ratings"]

        # send get request to the remote api
        api = 'https://quotes.rest/qod?category=sports&language=en'
        response = requests.get(api)

        # if the response is successful, process the response of the remote api
        if response.status_code == 200:
        res = response.json()["contents"]["quotes"][0]
        res["quote_text"] = res['quote']
        # due to request limit per hour, the response may return 429, too many requests
        # if so, process the response of the database
        else:
        res = quote_in_db
```

The general logical flow is as follows: The aim is to provide the quote of the day, along with its points and number of ratings. So, it first checks the database and tries to get the row with today's date. If a row is found, this means somebody has made a GET request before and the quote is saved to the database. Maybe some POST request has been made, so the points and number of ratings are read from the database response. Then, a request to the remote API is send.

```
    # make sure that the res is not None
       # it can be None if the remote api did not return 200 and the database is empty
       if res is not None:
       quote_text = res["quote_text"]
       author = res["author"]
       date = res["date"]
       # if there is not a row in the database for today, create a new object and save it to
the db
       if quote_in_db is None:
              new_quote = DailyQuote(quote_text=quote_text, author=author, date=date,
                                     points=0, ratings=0)
              new_quote.save()
              status = s.HTTP_201_CREATED

       # if there exists a row in the database for today, then calculate the value to be
shown
       else:
              if not ratings == 0:
              value = points * 1.0 / ratings
              value = "{:.1f}".format(value)
              status = s.HTTP_200_OK

       # send quote text, author, calculated value and number of ratings as the response
       dictionary = {"quote_text": quote_text, "author": author, "value": value, "ratings":
ratings}

       # if res is None (the remote api did not return 200 and the database is empty),
return 404
       else:
       dictionary = False
       status = s.HTTP_404_NOT_FOUND

       return Response({"response": dictionary}, status=status)
```

The remote API returns the sports quote of the day along with many other information, but the app just needs the quote and the author. Sometimes the remote API does not return due to too many requests. If so, it uses the database's return value to reach the quote and author. Either way, unless a very unlikely error occurs, the code reaches a quote and author. If the database does not contain a row for today, then the quote has 0 points and 0 ratings. It is saved to the database. If so, the return status becomes 201. If a new row for today hasn't been created, it returns with HTTP 200. If a quote from neither of the database and remote API could be reached, it returns with HTTP 404. If there is a successful return, the response has the keys quote_text, author, value and ratings. Value is the total points divided by number of ratings rounded to have 1 digit after decimal.

views.py

```
def rate(points):

        # make sure that the input is valid, if not, return 400
        if points is None or (not str(points).isdecimal()) or (not int(points) in [0, 1, 2,
3, 4, 5]):
        dictionary = False
        status = s.HTTP_400_BAD_REQUEST
        return Response({"response": dictionary}, status=status)

        today = str(datetime.utcnow().date())
        # if somebody has rated, the quote they liked is today's quote, which is in the
database
        try:
        quote_in_db = DailyQuoteSerializer(DailyQuote.objects.filter(date=today)[0]).data
        except:
        quote_in_db = None

        # if we could get the row from the database successfully, process it
        if quote_in_db is not None:
        points = int(points)
        # calculate points, ratings and value
        points_total = quote_in_db["points"] + points
        ratings_total = quote_in_db["ratings"] + 1
        value = points_total * 1.0 / ratings_total
        value = "{:.1f}".format(value)
        dictionary = {"quote_text": quote_in_db["quote_text"],
                      "author": quote_in_db["author"],
                      "value": value,
                      "ratings": ratings_total}
        DailyQuote.objects.filter(date=quote_in_db["date"]).update(points=points_total)
        DailyQuote.objects.filter(date=quote_in_db["date"]).update(ratings=ratings_total)
        status = s.HTTP_200_OK

        # if we could not get the row from the database successfully (unlikely) return 404
        else:
        dictionary = False
        status = s.HTTP_404_NOT_FOUND

        return Response({"response": dictionary}, status=status)
```

Above, the rate() function is given. The rate function basically returns a response with a body and a status. It is the code that runs when a POST request is made. Rows are explained in detail in the comments. The general logical flow is as follows: First the input is validated to be 0,1,2,3,4 or 5. If not, HTTP 400 is returned. If the input is valid, the row with today's date is read from the database. The points and ratings are updated and updated in the database. Then, a success response is returned with the quote, author, value and ratings. If due to some unlikely error, the data could not be read from the database, HTTP 404 is returned.

5.1.8 sportRelation API Code Documentation

```
class Sport(models.Model):
    link = models.CharField(max_length=2083, null=True)
    name = models.CharField(max_length=255, null=True)
    description = models.TextField(null=True)
    id = models.IntegerField(primary_key=True)
    slug = models.CharField(max_length=255, null=True)
    icon = models.CharField(max_length=2083, null=True)
```

Sport is an Django model which will be used to create a database table with column given as a field. It has necessary fields for a sport type.

```
class SportDetail(mixins.RetrieveModelMixin, generics.GenericAPIView):
    queryset = Sport.objects.all()
    serializer_class = SportSerializer

    def get(self, request, *args, **kwargs):
        return self.retrieve(request, *args, **kwargs)
```

`SportDetail` **is an API View with** `/sportRelation/api/sports/int:pk` **endpoint. It has only one get method and it retrieve data from Sport model.**

```
def get_related_sports(pk):
    url = base_url + str(pk)
    try:
        sports = requests.get(url).json()
        # if there is not any related sport (in this case decathlon api will not return a list for related field)
        if not isinstance(sports['data']['relationships']['related'], list):
            return []

        return [{
                **SportSerializer(Sport.objects.get(pk=x['data']['id'])).data,
                'weight': float(x['data']['weight'])
                } for x in sports['data']['relationships']['related']]
    except:  # If pk is not valid, return False
        return False
```

`get_related_sports(pk)` **is a helper function which will be used for similar and suggest endpoints. It takes a p.k. and fetch related sports of a sport from**

Decathlon API. After, synchronize it with `Sport` model with help of `SportSerializer`. If there is not any sport for given `pk` value. It returns False.

```
class SimilarSport(APIView):  # Will return at most five similar sport with given pk

    def get(self, request, pk):
        related_sports = get_related_sports(pk)

        if related_sports == False:  # If pk is not valid, return Not Found
            return Response({"detail": "Not found."}, status=status.HTTP_404_NOT_FOUND)

        if len(related_sports) > 5:  # If there are more than five related sport return only five of them
            return Response(related_sports[:5])

        return Response(related_sports)
```

`SimilarSport` is an API View with `/sportRelation/api/similar/int:pk` endpoint. It has only one get method and it returns at most five similar sport with given sport id.

```
class SuggestSport(APIView):

    def get(self, request, arg):
        pks = arg.split('-')
        suggestions = {}

        for pk in pks:  # Fetch similar sports one by one and add results to suggestions dictionary

            related_sports = get_related_sports(pk)

            if related_sports == False:  # If pk is not valid, return Not Found
                return Response({"detail": "Not found."}, status=status.HTTP_404_NOT_FOUND)

            for sport in related_sports:

                if sport['id'] in pks:  # we are looking for sports which are not choosen by user
                    continue

                # if we encounter with that sport before, update its weight
                if suggestions.get(sport['id']):
                    suggestions[sport['id']]['weight'] += sport['weight']
                else:  # if it is the first time we are encountering with that sport
                    suggestions[sport['id']] = sport

        suggestion = {}
        if suggestions:  # if suggestions dict is not an empty dict, return a suggestion with maximum weight
            suggestion = max(suggestions.values(), key=lambda x: x['weight'])

        return Response(suggestion)
```

`SuggestSport` is an API View with `/sportRelation/api/suggest/int:pk` endpoint. It has only one get method and it takes three p.k. and get their similar sports. After, it combine them and returns sport which has maximum total weight.

```python
class SaveSportListScript(APIView):
    permission_classes = [permissions.IsAdminUser]

    def post(self, request):
        url = base_url
        response = requests.get(url)  # fetch all sports
        sportlist = response.json()['data']

        sportlist = [{  # filter fields of sports
            'link': x['links']['self'],
            'name': x['attributes']['name'],
            'description': x['attributes']['description'],
            'id': x['id'],
            'slug': x['attributes']['slug'],
            'icon': x['attributes']['icon'],
        } for x in sportlist]

        ids = []

        for sport in sportlist:  # save fetched sports to datavase
            serializer = SportSerializer(data=sport)
            if serializer.is_valid():
                serializer.save()
                ids.append(sport['id'])
            else:  # if there is an array while saving the database, return HTTP_400
                return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

        # if all save operations are successfull, return their ids, with HTTP_201
        return Response({'AcceptedIds': ids}, status=status.HTTP_201_CREATED)
```

`SaveSportListScript` is an API View with `/sportRelation/api/save-sport-list-script` endpoint. It has only one post method and it is for only one time run. It can only be run by super admin to avoid possible security bug. It will fill the database with sports which are fetched from Decathlon API, and will filter necessary fields.

```python
def index(request):  # Render home page
    sportlist = Sport.objects.order_by('name')

    context = {
        'sportlist': sportlist
    }

    return render(request, 'sport_relation/index.html', context)

def similar(request):  # Render similar sports page
    api_url = get_api_url(request.build_absolute_uri())
    pk = request.GET.get('sportlist', False)
    url = api_url + "similar/" + pk

    similar_sports = requests.get(url).json()
    root_sport = Sport.objects.get(pk=pk)

    context = {
        "similar_sports": similar_sports,
        "root_sport": root_sport
    }

    return render(request, 'sport_relation/similar.html', context)

def suggest(request):  # Render suggested sport page
    api_url = get_api_url(request.build_absolute_uri())
    pks = [request.GET.get('sportlist1', False), request.GET.get(
        'sportlist2', False), request.GET.get('sportlist3', False)]
    url = api_url + "suggest/" + "-".join(pks)

    suggestion = requests.get(url).json()
    root_sports = [Sport.objects.get(pk=pk) for pk in pks]

    context = {
        "suggestion": suggestion,
        "root_sports": root_sports
    }

    return render(request, 'sport_relation/suggest.html', context)
```

`index` `similar` and `suggest` are three view for user interface. They have similar jobs, they take a request fetch the data with given pk or pks, and give the data to html template in order to render the html to user.

5.1.9 randomActivity API Code Documentation

```python
from django.shortcuts import render

# Create your views here.

# views.py
from rest_framework import viewsets

from .serializers import ActivitySerializer
from .models import Activity
import requests

#Send get method to retrieve data from boredAPI
#It returns data with JSON format
class ActivityViewSet(viewsets.ModelViewSet):
    queryset = Activity.objects.all().order_by('activity')
    serializer_class = ActivitySerializer

def showActivity(request):
    response = requests.get('http://www.boredapi.com/api/activity/')
    activityDetail = response.json()
    return render(request, 'activity/home.html', {
        'activity': activityDetail ['activity'],
        'type' : activityDetail ['type'],
        'participants' : activityDetail ['participants']

    })
```

```python
from django.test import TestCase
from randomActivity.models import Activity
from rest_framework import status
from rest_framework.test import APITestCase, APIClient
# Create your tests here.

class RandomActivity(APITestCase):

    # Tests if the get request works
    def test_get_method(self):
        client=APIClient()
        response=client.get("/randomActivity/api/")
        self.assertEqual(response.status_code,status.HTTP_200_OK)
```

5.2 API Documentations

5.2.1 WeatherCondition API documentation

weatherCondition API description

For the API service functionality

GET functionality

It returns an empty json with status code 200.

POST functionality

When the user gives a json in this format: `{ Town : town_name }` it returns the weather condition of the town with the data of the previous search.

For the UI of the API

GET functionality

It returns a form to write a town name to get its weather condition.

POST functionality

When the user has typed a town name, it returns the weather condition of the town with the data of the previous search as two lists. If the current search is not appropriate, it says "Not valid town name" and if there is no previous search it says "No previous search"

Functions

def weather_api(request)

It returns json response for the get and post requests. If the request is a GET, then `weather(method,searchTown=None)` is called with the parameter request.method which is "GET" and not any town information is sent to the method `weather(method,searchTown=None)` since this is a GET request.
If the request is POST, then `weather(method,searchTown=None)` is called with the parameter request.method which is "POST" and request.POST.get("Town") which is the given town name to search the weather condition.
`weather(method,searchTown=None)` returns json response for both type of requests and these responses are also returned by the `def weather_api(request)` function.

def weather(method,searchTown=None)

method: Stores the type of the request
searchTown: Stores the searched town name for POST requests. It is None if the request type is GET.

For method==GET:
Return value: `Response({},status=status.HTTP_200_OK)`

For method==POST:

Weather condition of a town is returned from openweathermap API via this URI:
`http://api.openweathermap.org/data/2.5/weather?q={}&appid={}&units=metric`

Return value: Data of the previous search is also provided. If openweathermap API returns with status code 200, the validity of the data is checked. If the data fits the limits of the model (If the length of the town and country names are less than or equal to 15 and the length

Return value: Data of the previous search is also provided. If openweathermap API returns with status code 200, the validity of the data is checked. If the data fits the limits of the model (If the length of the town and country names are less than or equal to 15 and the length of description is less than or equal to 50) then the data is saved in the database, the previous one and new weather conditions are returned with the status code 201. If the data doesn't fit the limits of the model, then only the previous data ise sent with the status code 400.

def showWeather(request)

It renders the data in the response returned by the `weather(method,searchTown=None)` function with html files. For get requests an html page where the user can type the name of town shows up, for post requests a view page where the weather conditions of the current and the previous page are listed is shown.

API key

It can be found in the .env file of the group5_api folder.

Example API requests and responses

Get Request:
`None` given

Response:
`{}`

Status code: 200

Post Request 1:

Post Request 1:
`{"Town" : "London"}`
Response:


```
    "new": {
        "country": "GB",
        "town": "London",
        "x": 51.5085,
        "y": -0.1257,
        "description": "broken clouds",
        "degrees": 24.12,
        "pressure": 1023,
        "humidity": 49,
        "speed": 3.02,
        "degreeOfWind": 248
    },
    "old": {
        "id": 1,
        "country": "TR",
        "town": "Istanbul",
        "x": 41.0351,
        "y": 28.9833,
        "description": "clear sky",
        "degrees": 22.28,
        "pressure": 1016.0,
        "humidity": 53.0,
        "speed": 1.71,
        "degreeOfWind": 162.0
    }
}
```

Status code: 201

Post Request 2:

`{"Town" : "shoe" }`

Response:


```
    "new": {},
    "old": {
        "id": 1,
        "country": "TR",
        "town": "Istanbul",
        "x": 41.0351,
        "y": 28.9833,
        "description": "clear sky",
        "degrees": 22.28,
        "pressure": 1016.0,
        "humidity": 53.0,
        "speed": 1.71,
        "degreeOfWind": 162.0
    }
}
```


Status Code: 404

Post Request 3:

`{"Town" : "llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch"}` Response:


```
    "new": {},
    "old": {
        "id": 1,
```



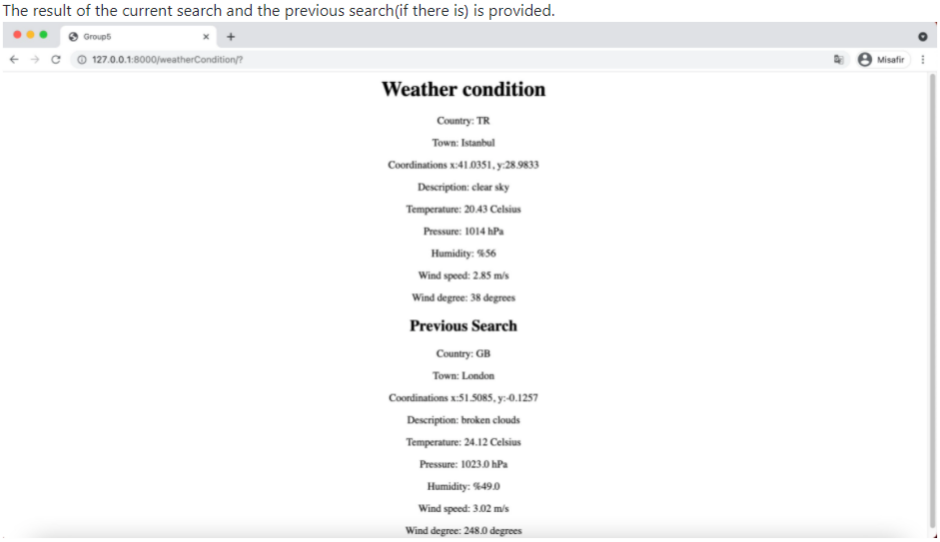
```
    "old": {
        "id": 1,
        "country": "TR",
        "town": "Istanbul",
        "x": 41.0351,
        "y": 28.9833,
        "description": "clear sky",
        "degrees": 22.28,
        "pressure": 1016.0,
        "humidity": 53.0,
        "speed": 1.71,
        "degreeOfWind": 162.0
    }
}
```


Status Code: 400

Example UI images of the API

1. Type a town name

2. The result of the current search and the previous search(if there is) is provided.



5.2.2 Core API Documentation

Authentication and Random Person API

Welcome to my Authentication and Random User API. This API is written in Django and Python. The main goal of the API that learn GET, POST methods and database operations like insert or select.

In this API; you can add user, log in and log out and see how many registered users are there in that API and also you can get a random person's name, gender and age.

You can access my API from main link : http://127.0.0.1:8000/authentication/

How to run the app:

- Clone the repository
- Install the requirements
- Run the following command to create the tables in the database:

```
python3 ./manage.py migrate
```

- Run the following command to start the app:

```
python3 ./manage.py runserver
```

- Open the browser (http://127.0.0.1:8000/authentication/)
- Quit the app using ctrl+c

5.2.3 AirQuality API Documentation

AirQuality API description

This is a simple API which returns air quality index and corresponding risk category for a specified location. If provided with a user_id it can store locations for a user and later use them to return air quality data.

This API uses IQ Air AirVisual API to get air quality index data.

API Routes

For air pollution services:

/airQuality/api/pollution/

For location history services:

/airQuality/api/location/

API functionality

GET /airQuality/api/pollution/?lat={{LATITUDE}}&lon={{LONGITUDE}}

Returns air quality, classification of air quality, description, city and country of given coordinates.

Arguments:

- lat: Latitude of the coordinate
- lon: Longitude of the coordinate

It returns a json.

GET /airQuality/api/pollution/?user_id={{USER_ID}}

Returns air quality, classification of air quality, description, city and country; based on latest recorded location of the user with user_id

Arguments:

- user_id: Id of the user

It returns a json.

GET /airQuality/api/location/

Returns all saved locations for all users, sorted by date added.

It returns a json.

GET /airQuality/api/location/?user_id={{USER_ID}}

GET /airQuality/api/location/?user_id={{USER_ID}}

Returns all saved locations for a user, sorted by date added.

Arguments:

- user_id: Id of the user

It returns a json.

POST /airQuality/api/location/

When the user gives a json in this format:

`{'user_id': '5', 'latitude': '39.7667', 'longitude': '30.5256'}`

It saves the coordinates and user_id.

UI of the API

This is a stub UI for the API. It can be accessed from /airQuality

GET /airQuality/?lat={{LATITUDE}}&lon={{LONGITUDE}}

GET /airQuality/?user_id={{USER_ID}}

This calls GET methods from /airQuality/api/pollution with according arguments and shows the result on a basic template.

I didn't have much time creating an interface and this only uses get methods, it can be improved by adding some basic buttons. It can be accessed from this address: http://3.127.142.97:8000/airQuality/ I am providing more input and outputs below.

Input and Output

POST /airQuality/api/locations/

Create a location for user id 61 Using data below:

```
{
        "user_id": 61,
        "latitude": 41.0027,
        "longitude": 39.7168
}
```

Output:

```
{
        "created": "2021-06-09T22:08:46.766111Z",
        "user_id": 61,
        "latitude": 41.0027,
        "longitude": 39.7168
```

GET /airQuality/api/locations/?user_id=61

Get saved location of user id 61 Output:


```
[
        {
                "created": "2021-06-09T22:08:46.766111Z",
                "user_id": 61,
                "latitude": 41.0027,
                "longitude": 39.7168
        }
]
```


GET /airQuality/api/pollution/?user_id=61

Find pollution on user's latest recorded location Output:


```
{
    "aqi": 69,
    "name": "Moderate",
    "color": "#faff29",
    "description": "Air quality is acceptable. However, there may be a risk for some people, particularly those who are unusually sen
    "polluiton_ts": "2021-06-09T22:00:00.000Z",
    "city": "Trabzon",
    "state": "Trabzon",
```

```
    "state": "Trabzon",
    "country": "Turkey",
    "coordinates": [
        39.731671906746335,
        41.00458548106175
    ]
}
```


GET /airQuality/api/pollution/?lat=39.7667&lon=30.5256

Get pollution of a location directly Output:


```
{
    "aqi": 48,
    "name": "Good",
    "color": "#47e60e",
    "description": "Air quality is satisfactory, and air pollution poses little or no risk.",
    "polluiton_ts": "2021-06-09T22:00:00.000Z",
    "city": "Odunpazari",
    "state": "Eskisehir",
    "country": "Turkey",
    "coordinates": [
        30.53538,
        39.76821
    ]
}
```


5.2.4 MusicApi API Documentation

Musicapi API Description

Welcome to the Music API.

Musicapi provides the song names of a particular sport playlist. Music API is developed with Django framework and it only supports GET method.

API Routes

/musicapi/api

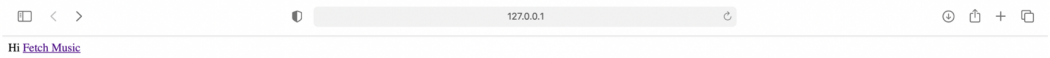
API Functionality

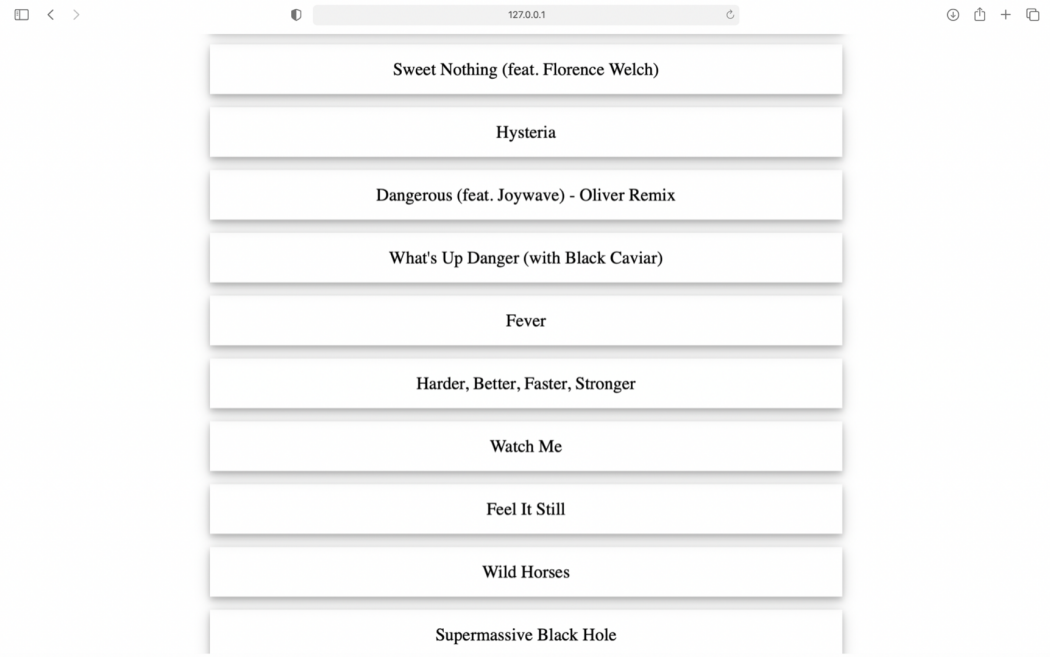
GET /musicapi/api

returns name of the songs in json format.

There should be two input in the code, PLAYLIST_ID and TOKEN. https://open.spotify.com/playlist/4JhDvULKIy5CXyzJRufHLt?si=14e994f8c07a4dc4 PLAYLIST_ID: 4JhDvULKIy5CXyzJRufHLt OAuth Token can be reachable on https://developer.spotify.com/console/get-playlist-tracks/ OAuth is valid for 1 hour. I couldn't find a way to extend this time period. Because of that I should renew it to get the list of song names.

Here is some screenshots of the API.





5.2.5 DailyQuote API Documentation

The Daily Quote API is an API that provides you daily quotes about sports, in English. The Daily Quote API is written in Django and is connected with a PostgreSQL database. The Daily Quote API is a RESTful API and it uses a remote API, They Said So Quotes API.

Each day's quote, author and date are saved to the database if it has not been saved before. The Daily Quote API adds a rating functionality to the remote API. It supports GET and POST methods. When a user makes a GET request to the API, they see the average rating of the quote and the number of ratings made so far, along with the quote and the author. Users can rate the quote with a point of their wish. Allowed points are 0,1,2,3,4 and 5. When a POST request is made with a valid point, the total points of the quote is incremented by the given point and the total ratings of the quote is incremented by one. These values are updated in the database. Then, the new average value and number of ratings are returned in the response.

The Daily Quote API uses a remote API named They Said So Quotes API. The remote API provides many functionalities like quote search, quote images, quote of the day, etc. The Daily Quote API uses the quote of the day functionality and filters the quote category as sports and the quote language as English by making a get request to the remote API on URI https://quotes.rest/qod?category=sports&language=en.


```
{
        "success": { "total": 1},
        "contents": {
        "quotes": [
                {
                "quote": "Bodybuilding is much like any other sport. To be successful, you
must dedicate yourself 100% to your training, diet and mental approach.",
                "length": "136",
                "author": "Arnold Schwarzenegger",
                  "tags": ["diet", "sports"],
                "category": "sports",
                "language": "en",
                "date": "2021-06-08",
                "permalink":
"https://theysaidso.com/quote/arnold-schwarzenegger-bodybuilding-is-much-like-any-other-spor
t-to-be-successful",
                "id": "LflBg5X1AMVlVOsgoso6aweF",
                "background": "https://theysaidso.com/img/qod/qod-sports.jpg",
                "title": "Sports Quote of the day"
                }
        ]
        },
        "baseurl": "https://theysaidso.com",
        "copyright": {"year": 2023, "url": "https://theysaidso.com"
        }
}
```


For more information on the remote API: https://quotes.rest/

Endpoint

```
GET http://localhost:8000/dailyQuote/api
```

This endpoint retrieves the sports quote of today, it's author, the average of ratings and the number of ratings of that quote.

Request Parameters

None

Response

```
Response({"response": <dictionary>}, status=<HTTP status>)
```

Sample Call

```
{
      "response": {
      "quote_text": "Bodybuilding is much like any other sport. To be
successful, you must dedicate yourself 100% to your training, diet and mental
approach.",
      "author": "Arnold Schwarzenegger",
      "value": "5.0",
      "ratings": 1
      }
}
```

This is a successful response with status code 200.

Endpoint

```
POST http://localhost:8000/dailyQuote/api/
```

This endpoint allows rating the quote between 0 and 5. The response is the quote of, it's author, the new average of ratings and the new number of ratings of that quote.

Request Parameters

Points: Taken as form parameter. Stands for the points the user gives to the quote. The API only allows values 0,1,2,3,4 and 5 as legal points.

Response

```
Response({"response": <dictionary>}, status=<HTTP status>)
```

Sample Call (Points = 4)

```
{
      "response": {
      "quote_text": "Bodybuilding is much like any other sport. To be
successful, you must dedicate yourself 100% to your training, diet and mental
approach.",
      "author": "Arnold Schwarzenegger",
      "value": "4.5",
      "ratings": 2
      }
}
```

This is a successful response with status code 200.

Sample Call (Points = 6)

```
{"response": False}
```

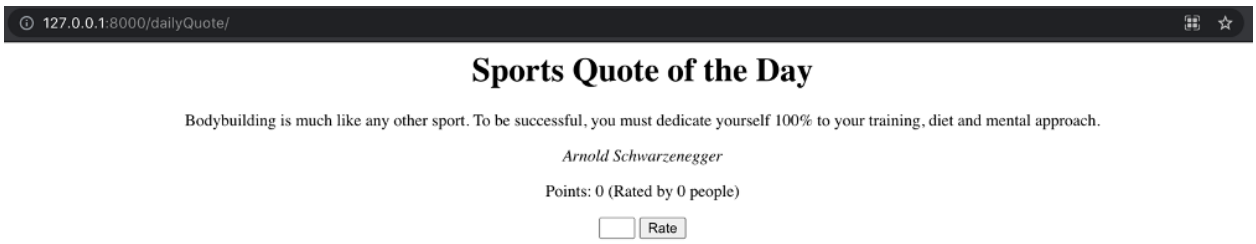
This is an unsuccessful response with status code 400 as the value is greater than 5.

Sample Call (Points = 'bad_request')

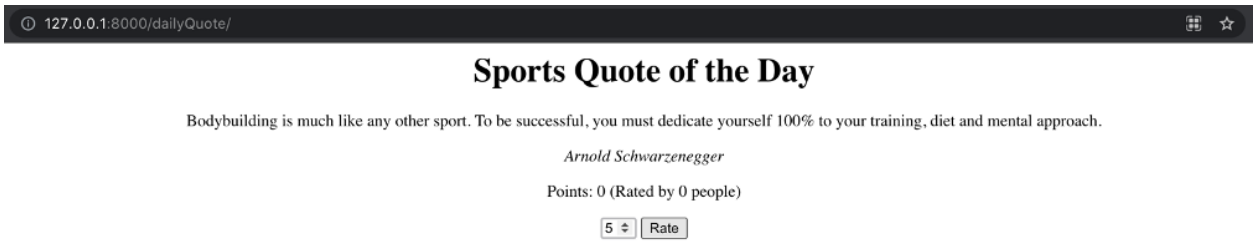
```
{"response": False}
```

This is an unsuccessful response with status code 400 as the value is not an integer.

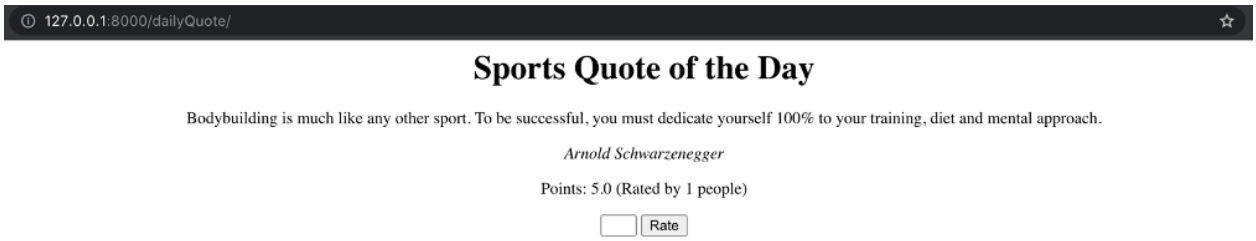
The API functionality can be tested via the user interface or tools like Postman. Below, some screenshots are attached that show the app works logically.



(the sports quote of the day is shown as a result of the GET request)



(the form input is filled)



(when the rate button is clicked, a POST request is made, and the page is updated)

5.2.6 nbaStats API Documentation

nbaStats API Description

Welcome to nbaStats API. You can search for NBA players by their name (and/or surname) and you can get some information about them. API is developed with Django Framework. Also API uses 3rd party API [balldontlie].(https://www.balldontlie.io/api/v1/players)

API supports GET and POST requests. Frontend is implemented using HTML. HTML files are located in templates file. Also API is provided tests. Command in order to run the tests:

```
python manage.py test nbaStats
```

Methods:

API supports GET and POST methods. GET and POST requests are in the same function. The requests are in the same function because GET methods practically does nothing. It just renders the base page. POST part is the main part. First it handles status information. Then this part creates a `Player()` object according to information that POST request got. Finally Method renders the data and status information to the page.

Possible Error Codes:

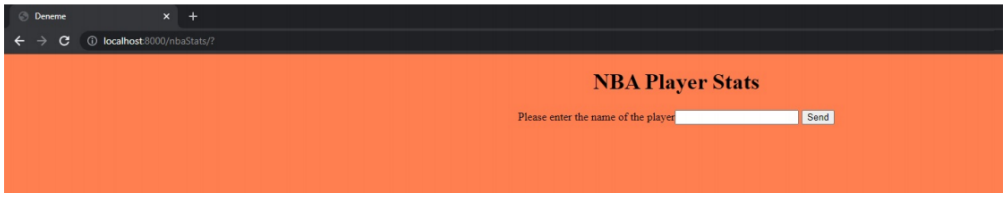
If there is no Player: Error code `HTTP_400_BAD_REQUEST`

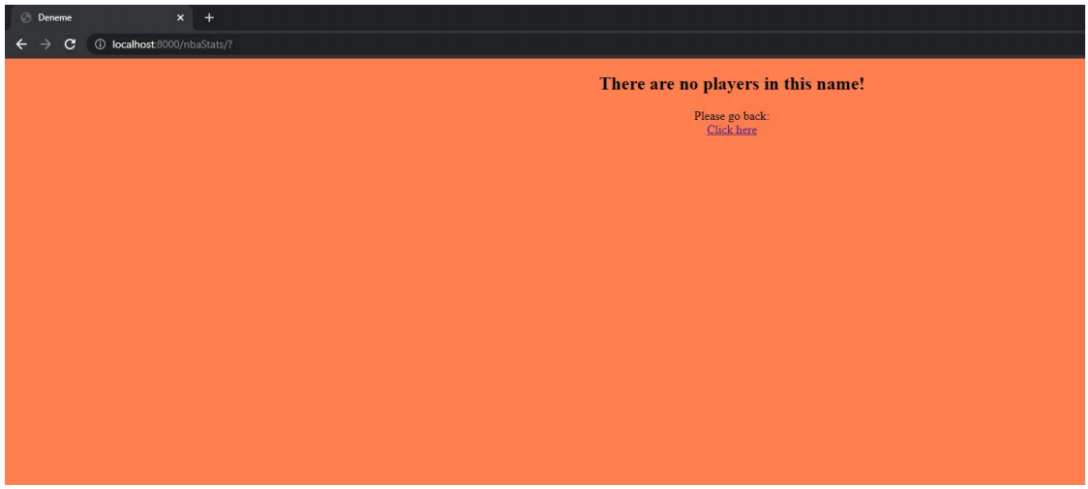
If everything works `HTTP_200_OK`

If there is no match according to request information `HTTP_404_NOT_FOUND`

Details are on the code documentation

Screenshots of Interface:





How to run the app:

- Clone the repository
- Install the requirements
- Set up a PostgreSQL database named `group5db`
- Run the following command to create the tables in the database:

  `python3 ./manage.py migrate`

- Run the following command to start the app: `python3 ./manage.py runserver`
  - Quit the app using Ctrl+C

You can run the tests with following command:

```
python3 ./manage.py test
```

5.2.7 sportRelation API Documentation

Sport Relation API Documentation

Sport Detail

Endpoint: `GET /sportRelation/api/sports/:pk`

This endpoint retrieves a single sport. The pk parameter is a unique primary key for a sport.

Request Parameters

The only accepted (and required) parameter for this endpoint is the sport `pk`

Example for : `GET /sportRelation/api/sports/1`

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "id": 1,
    "name": "Duathlon",
    "description": "Because running and cycling are two perfectly complementary sports, duathlon will make you work on different musc
    "link": "/sports/1",
    "slug": "duathlon",
    "icon": null
```

Similar Sport

Endpoint: `GET /sportRelation/api/similar/:pk`

This endpoint retrieves at most five sports which are most similar sports with given sport in order. The pk parameter is a unique primary key for a sport.

Request Parameters

The only accepted (and required) parameter for this endpoint is the sport `pk`

Example for : `GET /sportRelation/api/similar/1`

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 391,
        "name": "Triathlon",
        "description": "Combining three endurance races, in order, swimming, cycling and running, triathlon is practiced over various
        "link": "/sports/391",
        "slug": "triathlon",
```

```
    },
    {
        "id": 900,
        "name": "Aquathlon",
        "description": "Running and swimming are the center of your sporting life? Why don't you take part in an aquathlon race? This
        "link": "/sports/900",
        "slug": "aquathlon",
        "icon": null,
        "weight": 0.25225590317391
    },
    {
        "id": 899,
        "name": "Winter triathlon",
        "description": null,
        "link": "/sports/899",
        "slug": "winter-triathlon",
        "icon": null,
        "weight": 0.238386339769658
    },
    {
        "id": 511,
        "name": "Long-distance running",
        "description": null,
        "link": "/sports/511",
        "slug": "long-distance-running",
        "icon": null,
        "weight": 0.215339218561191
    },
    {
        "id": 508
```

```
    },
    {
        "id": 508,
        "name": "Adventure running",
        "description": null,
        "link": "/sports/508",
        "slug": "adventure-running",
        "icon": null,
        "weight": 0.215339218561191
    }
]
```


Suggest Sport

Endpoint: `GET /sportRelation/api/suggest/:pk1-pk2-pk3`

This endpoint retrieves a single sport which are the suggestion of the system (sport which has maximum total weight) for given three sport. The pk1-pk2-pk3 parameter is a string which contains three unique primary keys for three sports.

Request Parameters

The only accepted (and required) parameter for this endpoint is the `pk1-pk2-pk3`

Example for : `GET /sportRelation/api/suggest/1-4-5`

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
```


```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "id": 511,
    "name": "Long-distance running",
    "description": null,
    "link": "/sports/511",
    "slug": "long-distance-running",
    "icon": null,
    "weight": 0.790303418869382
}
```


Save Sport List Script

Endpoint: `POST /sportRelation/api/save-sport-list-script`

This endpoint saves necessary fields of all sports which retrieved from Decathlon API in order to fill database for future usages. It can only be called by super admin.

Example for : `POST /sportRelation/api/save-sport-list-script`

```
HTTP 201 Created
Allow: POST, OPTIONS
```

This endpoint saves necessary fields of all sports which retrieved from Decathlon API in order to fill database for future usages. It can only be called by super admin.

Example for : `POST /sportRelation/api/save-sport-list-script`

```
HTTP 201 Created
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "AcceptedIds": [
        282,
        284,
        285,
        224,
        286,
                                    ...
                ]
}
```

5.2.8 randomActivity API Documentation

Random Activity Generator API

Brief explanation of the API

Random activity generator API is an application programming interface which shows different activities every time it asked. It generates different types of activities with different number of people. Target user of the API is bored people as it is written at the top of the html page. For generating random activities, as a third party API, boredAPI (https://www.boredapi.com/) is used.

How to Run

- Create a virtual environment
- Install requirements of the API.(requirements.txt)
- From python run the command: python3 ./manage.py migrate
- From python run the command: python3 ./manage.py sunserver
- From browser open the link: http://127.0.0.1:8000/

Request and response of boredAPI

It requests with get method via http://www.boredapi.com/api/activity/.

It requests with get method via http://www.boredapi.com/api/activity/.

Result of this request is in JSON format. An example output is like that:

{

"activity": "Learn Express.js",

"accessibility": 0.25,

"type": "education",

"participants": 1,

"price": 0.1,

"link": "https://expressjs.com/",

"key": "3943506"

}

Used Tech

In the process of producing the API, Django and Django Rest Framework are used. So we used python.

Other than backend usage, HTML is used to show output of the API.

In the process of producing the API, Django and Django Rest Framework are used. So we used python.

Other than backend usage, HTML is used to show output of the API.

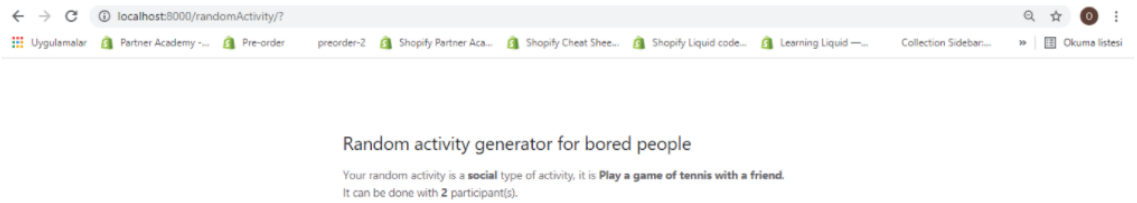
As database it is decided to use PostgreSQL. For the entire project of practise-API we used PostgreSQL as a team.

Function

def showActivity(request)

This function communicate with the other API. It requests Get from http://www.boredapi.com/api/activity/. And in return renders the input as activity, type, and participants. It returns JSON response as an output.

User Interface of the API



5.2.9 exchangeRateAPI API Documentation

1. POST /add/category/

Creating a new sport category into database

Parameters:

- *category_name:* String the name of the category
- *category_description:* String the description of the category
- *max_player:* Integer max player can play that category

Response:
Response in JSON. With the following information:

- *responseCode:* The status code of the response
- *responseText:* The status message of the response

2. GET /category/ or /category/all/

Returns all categories which saved into database

Response:
Response in JSON. With the following information:

- *category_name:* String the name of the category
- *category_description:* String the description of the category
- *max_player:* Integer max player can play that category

3. GET /category/<str:category_name>/

Returns spesifik category fields by category name

Parameters:

- *category_name:* String the name of the category

Response:
Response in JSON. With the following information:

- *category_name*
- *category_description*
- *max_player*

4. POST /add/eventPost/

Creating a event post into database

Parameters:

- *event_post_id:* Integer event post id which can be added manually or incrementally
- *date_time:* Datetime date of the event
- *participation_limit:* Integer limit of the participation
- *spectator_limi:* Integer limit of the spectator
- *rule:* String the rule of event which determined by creator
- *equipment_requirement:* String event creator can determine equipment requirements for that event
- *status: String the status of the event like online, face-to-face*
- *location: String the location of the event, it can be address or choosing from the map*
- *contact_info: String the information of the creator of event*
- *skill_requirement: String event creator can determine skill requirements of the participation*
- *repating_frequency: Integer event creator can create event automatically for next weeks while using that field*

Response:
Response in JSON. With the following information:

- *responseCode:* The status code of the response
- *responseText:* The status message of the response

5. GET /eventPost/ or /eventPost/all/

Returns all event posts which saved into database

Response:
Response in JSON. With the following information:

- *event_post_id*
- *date_time*
- *participation_limit*
- *spectator_limit*
- *rule*
- *equipment_requirement*
- *status*
- *location*
- *contact_info*
- *skill_requirement*

- *repating_frequency*

6. GET /eventPost/<int:post_id>/

Returns spesifik event post fields by event post id

Parameters:

- *event_post_id:* Integer event post id

Response:

Response in JSON. With the following information:

- *event_post_id*
- *date_time*
- *participation_limit*
- *spectator_limit*
- *rule*
- *equipment_requirement*
- *status*
- *location*
- *contact_info*
- *skill_requirement*
- *repating_frequency*

7. GET /currency/ or /currency/all/

Returns all currencies rates from external API which is http://openexchangerates.org/

Response:

Response in JSON. With the following information:

- *disclaimer: String the term link*
- *licence: String the licence link*
- *timestamp: Timestamp the response time for API*
- *base: String the base currency which is USD*
- *rates Array it returns all currency rates based on USD*

8. GET /currency/<str:target_currency>/

Returns spesifik a currency rate from external API

Parameters:

- *target_currency:* String the target currency which learned exchange rate based on USD

Response:
Response in TEXT. With the following information:

- *responseCode:* The status code of the response
- *responseText:* The status message of the response

9. POST /add/currentCurrency/
Calculates any amount of USD to target exchange rate and add inputs and result into database

Parameters:

- *base_total:* Integer total amount of currency to exchange another rate
- *target_currency:* String the target currency which exchange to it

Response:
Response in JSON. With the following information:

- *responseCode:* The status code of the response
- *responseText:* The status message of the response (if successful, the calculation is shown here)