Table of Contents

- Executive Summary
  - Summary of the Project and Overall Status
  - Deployed Application
  - Functionality
  - Challenges We Faced As A Group
  - Status of Current Deliverables
- Project Plan
- Individual Work
- Evaluation of Tools and Processes
    - Github
    - Zoom
    - Google Meets
    - Whatsapp
    - Flask
    - Jinja
    - SqlAlchemy
    - Swagger
    - unittest
    - Docker
    - Postman
    - AWS
  - APIs Used
    - Google Docs Geocoding
    - The Sports DB
    - Decathlon Developers
    - Bitcoin Price
    - OpenWeather
    - NameFake
    - Random User Generator
- Our Main Page
- API Documentations
    - Badge Related Endpoints
      - Adding a Badge to the System
      - Getting a Badge Point by Badge name
      - Getting All Badges
    - Event Related Endpoints
      - Adding an Event to the System:
      - Getting All Events From The System
        - Swagger Documentation

Related Functions
Endpoint Code
Unit Tests
Server part that uses the specified API
Sample frontend product:
Posting a New Interested (for an Event) Into System
Swagger Documentation
Endpoint Code
Getting an Event by its Id
Swagger Documentation:
Endpoint Code:
Unit tests, event_get_by_id_test.py
Frontend Part with Screenshots
Adding a new Spectator to an Event
Swagger Documentation
Endpoint Code:
Sport Related Endpoints
Basic Filtering of Sports
Discussion Related Endpoints
Equipment Related Endpoint
Authentication

Executive Summary

Summary of the Project and Overall Status

In this project, we built an app about amateur sports. Our app has two parts: one for general usage, that is, a front end that is run by a browser and can be viewed, and a bunch of RESTful APIs that serve for basic operations. These basic operations are elements such as creating a new event, viewing a badge with its name etc. We deployed our app into AWS and used Docker Hub. You can reach it through this url. In this milestone report, everything we did to create our practice-app and our approach to the problems we faced is documented.

Deployed Application

We adjust our docker-compose and Dockerfile. Afterwards, we created our image and tagged it as berkatil/practiceapp so that we can easily push this image on Berk's docker account. Then, we created an AWS configuration file which is shown below. Afterwards, we went to the AWS console and selected Elastic Beanstalk. Using the configuration file we deployed our application.

Configuration settings:

```
{ "AWSEBDockerrunVersion": "1",
  "Image": { "Name": "berkatil/practiceapp", "Update": "true" },
  "Ports": [{"ContainerPort": 5000,"HostPort": 8080}],
  "Logging": "/var/log/nginx"
}
```

Application URI: http://practiceapp6-env.eba-hx4ciydq.us-east-2.elasticbeanstalk.com

API URI: http://practiceapp6-env.eba-hx4ciydq.us-east-2.elasticbeanstalk.com/api/v1.0/
Requests to the endpoints can be sent to the API URI.

API Swagger Documentation:
http://practiceapp6-env.eba-hx4ciydq.us-east-2.elasticbeanstalk.com/apidocs/
API Wiki Documentation:
https://github.com/bounswe/2021SpringGroup6/wiki/Practice-app-API-Documentation

Functionality

Our project has two kinds of functionality. First one is basic user interface. In that part we provide a page that users can see current events, create a new event, see badges etc. Users can filter events and go to a specific event's page.

Actions that can be taken by a user include:

- Register, Login, Logout
- Create an event
- Get information of an event given its id
- Search events using a filter
- See interested users given the event id
- Post a discussion on an event
- Get discussions of an event
- Create a badge
- Get badge information
- Get points of badges
- Create a equipment
- Get sports information

Second one is our RESTful APIs. These APIs also provide service for our server part, we call these APIs to fetch data from database and render them in our server. Our APIs include `api/v1.0/equipments/` (post), `api/v1.0/events/` (get and post), `api/v1.0/<event_id>` (get), `api/v1.0/<event_id>/discussions` (get),

`api/v1.0/badges (post)`, `api/v1.0/badges/show_badge (get)`, `api/v1.0/badges/point (get)`, `api/v1.0/equipments (post)`, `api/v1.0/events/<event_id>/interesteds (post)`, `api/v1.0/sports (get)`, `api/v1.0/events/<event_id>/spectators (post)`. All endpoints return json objects as response. Also note that post methods expect json objects as input. Documentation of these endpoints are created using Swagger and can be found at “API Documentations” section.

Challenges We Faced As A Group

We learnt something about working as a group from the previous milestone project. In this project, our first challenge was that everybody started learning different tools from different sources. After that, when we came together, we had difficulty getting ourselves arranged to others’ work. This resulted in some leading practices and may have made some of us sad.

Second challenge actually comes from the first one. Because we learnt different practices from different resources, we had tough arguments in the early stage of our planning. We had to change the structure once. After that we thought more deeply and made no mistake.

Third challenge we faced was deadlines. We could not find a time to meet and put our work together, earlier than the final deadline. This resulted in missed deadlines and we were 1 day behind the desired program. After that, we freakend by the strict deadline policy and started working harder. This resolved our problem.

Fourth challenge is about building code and putting our code together. After we decided the form and general pattern of the project, we built some code, tested and wanted to merge them to the master. At that point, some difficulties arose:

- Some coding patterns did not match with the others. This resulted in some long conversations but at last they were resolved peacefully.
- New merges resulted in more conflicts in codes and this was annoying.
- Long conversations and more conflicts make some pull requests wait for a long time. This situation made some of us sad.

Afterall, all these processes are helpful and taught us about working as a group.

In addition to these points, the field about which we did the project was totally new for almost all of us, so we firstly tried to get familiar working for a web project which, definitely, has some differences from what we usually did until now. For instance, initially, we

conducted a small research about which language and frameworks to use in the project. Then we decided to use Python and Flask, both of which have ease of use. Also we had to learn about API's in general and specifically Restful API's. We've looked at the resources available on the internet and the ones who found any shared them with the rest of the group in order to avoid repetitive effort while dealing with the challenge of learning completely new topics.

Also providing a user interface was another point that we had to deal with since we had no serious experience in that, either. Nevertheless, each person has tried to learn about it and again by sharing resources and information, things became easier.

We couldn't be sure about what the desired point was for the unit testing part of the project. Even though we discussed the topics in the group we couldn't find a satisfactory solution, so we talked about it with our TA, which has made the picture far more clearer.

As a summary, learning totally new concepts combined with carrying out a project as a team have some difficulties; nonetheless, when we make more and more practice, we will be able to deal with challenges encountered more easily. In fact, this point is true both in the scope of what we've done and in general about life.

Status of Current Deliverables

Deliverable	Completion Time (Last Update)
Practice App Implementation	09.06.2021
Dockerize	09.06.2021
Deployment on AWS	10.06.2021
API Documentation	10.06.2021

Project Plan

The updates in the project plan after our work on the Practice App are given below:

15		**Meetings**	197 days?	**3/25/21 ...**	**12/27/2...**	
16		**Group Meetings**	52 days?	**3/25/21 ...**	**6/8/21 ...**	
17		Meeting 1	0 days	3/25/21 ...	3/25/21...	Musa Nuri Ihtiyar; Deniz Arda Budak;Elif Sema Balcioglu;Ibrahim Melih Aktas;Ekrem Yusuf Ekmekci;Can Atakan Ugur;Berk Atil;...
18		Meeting 2	0 days	4/2/21 6...	4/2/21 ...	Musa Nuri Ihtiyar; Deniz Arda Budak;Elif Sema Balcioglu;Ibrahim Melih Aktas;Ekrem Yusuf Ekmekci;Can Atakan Ugur;Berk Atil
19		Meeting 3	0 days	4/6/21 6...	4/6/21 ...	Musa Nuri Ihtiyar;Berk Atil;Omer Suve;Ibrahim Melih Aktas;Elif Sema Balcioglu
20		Meeting 4	0 days	4/13/21 ...	4/13/21...	Musa Nuri Ihtiyar; Deniz Arda Budak;Elif Sema Balcioglu;Ibrahim Melih Aktas;Ekrem Yusuf Ekmekci;Can Atakan Ugur;Berk Atil
21		Meeting 5.1	0 days	4/20/21 ...	4/20/21...	Musa Nuri Ihtiyar; Deniz Arda Budak;Elif Sema Balcioglu;Ibrahim Melih Aktas;Ekrem Yusuf Ekmekci;Can Atakan Ugur;Berk Atil;...
22		Meeting 5.2	0 days	4/22/21 ...	4/22/21...	Musa Nuri Ihtiyar; Deniz Arda Budak;Elif Sema Balcioglu;Ibrahim Melih Aktas;Ekrem Yusuf Ekmekci;Can Atakan Ugur;Berk Atil;...
23		Meeting 5.3	1 day	4/24/21 ...	4/26/21...	Musa Nuri Ihtiyar; Deniz Arda Budak;Elif Sema Balcioglu;Ibrahim Melih Aktas;Ekrem Yusuf Ekmekci;Can Atakan Ugur;Berk Atil;...
24		Meeting 6	0 days	4/27/21 ...	4/27/21...	Musa Nuri Ihtiyar; Deniz Arda Budak;Elif Sema Balcioglu;Ibrahim Melih Aktas;Ekrem Yusuf Ekmekci;Berk Atil;Omer Suve
25		Meeting 7.1	0 days	5/4/21 6...	5/4/21 ...	Musa Nuri Ihtiyar; Deniz Arda Budak;Elif Sema Balcioglu;Ibrahim Melih Aktas;Ekrem Yusuf Ekmekci;Can Atakan Ugur;Berk Atil;...
26		Meeting 7.2	1 day	5/8/21 6...	5/10/21...	Musa Nuri Ihtiyar; Deniz Arda Budak;Elif Sema Balcioglu;Ibrahim Melih Aktas;Ekrem Yusuf Ekmekci;Can Atakan Ugur;Berk Atil;...
27		Meeting 8	0 days?	5/21/21 ...	5/21/21...	Musa Nuri Ihtiyar; Deniz Arda Budak;Elif Sema Balcioglu;Ibrahim Melih Aktas;Ekrem Yusuf Ekmekci;Can Atakan Ugur;Berk Atil;...
28		Meeting 9.1	0 days?	5/25/21 ...	5/25/21...	Musa Nuri Ihtiyar; Deniz Arda Budak;Ibrahim Melih Aktas;Elif Sema Balcioglu;Ekrem Yusuf Ekmekci;Can Atakan Ugur;Berk Atil;...
29		Meeting 9.2	1 day?	5/30/21 ...	5/31/21...	Musa Nuri Ihtiyar; Deniz Arda Budak;Elif Sema Balcioglu;Ekrem Yusuf Ekmekci;Can Atakan Ugur;Berk Atil;Omer Suve;Ibrahim ...
30		Meeting 10	1 day?	6/5/21 8...	6/7/21 ...	Musa Nuri Ihtiyar; Deniz Arda Budak;Elif Sema Balcioglu;Ibrahim Melih Aktas;Ekrem Yusuf Ekmekci;Can Atakan Ugur;Berk Atil;...
31		Meeting 11	0 days?	6/8/21 8...	6/8/21 ...	Musa Nuri Ihtiyar; Deniz Arda Budak;Elif Sema Balcioglu;Ibrahim Melih Aktas;Ekrem Yusuf Ekmekci;Can Atakan Ugur;Berk Atil;...
32		**Extra Meetings for Practice App**	13 days?	**5/24/21 ...**	**6/9/21 ...**	
33		Back-End Meeting	1 day?	5/24/21 ...	5/24/21...	Elif Sema Balcioglu; Deniz Arda Budak;Berk Atil;Omer Suve
34		Front-End Meeting	0 days?	5/24/21 ...	5/24/21...	Musa Nuri Ihtiyar; Deniz Arda Budak;Elif Sema Balcioglu;Ibrahim Melih Aktas;Ekrem Yusuf Ekmekci;Can Atakan Ugur
35		Final Deployment Meeting	1 day?	6/9/21 8...	6/9/21 ...	Elif Sema Balcioglu;Ibrahim Melih Aktas;Berk Atil; Deniz Arda Budak

102		**PRACTICE - APP**	17 days?	**5/19/21 ...**	**6/10/21...**	
103		**Determination of Critical Sections**	5 days	**5/19/21 ...**	**5/25/21...**	**Musa Nuri Ihtiyar; Deniz Arda Budak;Elif Sema Balcioglu;Ibrahim Melih Aktas;Ekrem Yusuf Ekmekci;Can Atakan Ugur**
104		Questions to ask TA	5 days	5/19/21 ...	5/25/21...	
105		**Research & Pre-Implementation**	5 days?	**5/19/21 ...**	**5/25/21...**	**Musa Nuri Ihtiyar; Deniz Arda Budak;Elif Sema Balcioglu;Ibrahim Melih Aktas;Ekrem Yusuf Ekmekci;Can Atakan Ugur**
106		Research about public APIs	5 days?	5/19/21 ...	5/25/21...	
107		General research about web frameworks	5 days?	5/19/21 ...	5/25/21...	
108		Determining public APIs that can be used in the app	1 day?	5/25/21 ...	5/25/21...	
109		Determining which framework to use	1 day?	5/25/21 ...	5/25/21...	
110		Determining resources to follow for consistency	1 day?	5/25/21 ...	5/25/21...	
111		**Implementation Start**	1 day?	**5/26/21 ...**	**5/26/21...**	**Musa Nuri Ihtiyar; Deniz Arda Budak;Elif Sema Balcioglu;Ibrahim Melih Aktas;Ekrem Yusuf Ekmekci;Can Atakan Ugur**
112		**Front-End**	1 day?	**5/26/21 ...**	**5/26/21...**	
113		Everyone will code front-end for their endpoints	1 day?	5/26/21 ...	5/26/21...	
114		**Back-End**	0 days?	**5/26/21 ...**	**5/26/21...**	
115		Endpoints are divided across the team	0 days?	5/26/21 ...	5/26/21...	
116		**Back-End**	12 days?	**5/26/21 ...**	**6/10/21...**	
117		**Authorization**	12 days?	**5/26/21 ...**	**6/10/21...**	**Omer Suve**
118		login endpoint	12 days?	5/26/21 ...	6/10/21...	Omer Suve
119		logout endpoint	12 days?	5/26/21 ...	6/10/21...	Omer Suve
120		sign-up endpoint	12 days?	5/26/21 ...	6/10/21...	Omer Suve
121		**Event**	12 days?	**5/26/21 ...**	**6/10/21...**	
122		show event endpoint	12 days?	5/26/21 ...	6/10/21...	Deniz Arda Budak
123		event search endpoint	12 days?	5/26/21 ...	6/10/21...	Ekrem Yusuf Ekmekci
124		create event endpoint	12 days?	5/26/21 ...	6/10/21...	Elif Sema Balcioglu
125		**Equipment**	12 days?	**5/26/21 ...**	**6/10/21...**	
126		create equipment endpoint	12 days?	5/26/21 ...	6/10/21...	Omer Suve
127		**Badge**	12 days?	**5/26/21 ...**	**6/10/21...**	**Berk Atil**
128		show badge endpoint	12 days?	5/26/21 ...	6/10/21...	Berk Atil

129		create badge endpoint	12 days?	5/26/21 ...	6/10/21...	Berk Atil
130		**Sports**	12 days?	**5/26/21 ...**	**6/10/21...**	
131		show sport endpoint	12 days?	5/26/21 ...	6/10/21...	Elif Sema Balcioglu
132		**Spectators**	12 days?	**5/26/21 ...**	**6/10/21...**	
133		create spectators endpoint	12 days?	5/26/21 ...	6/10/21...	Deniz Arda Budak
134		**Interesteds**	12 days?	**5/26/21 ...**	**6/10/21...**	
135		create interested endpoint	12 days?	5/26/21 ...	6/10/21...	Ekrem Yusuf Ekmekci
136		**Testing**	12 days?	**5/26/21 ...**	**6/10/21...**	**Musa Nuri Ihtiyar; Deniz Arda Budak;Elif Sema Balcioglu;Ibrahim Melih Aktas;Ekrem Yusuf Ekmekci;Can Atakan Ugur**
137		Tests for functions used in api	12 days?	5/26/21 ...	6/10/21...	
138		**Discussions**	12 days?	**5/26/21 ...**	**6/10/21...**	
139		post a comment to a discussion endpoint	12 days?	5/26/21 ...	6/10/21...	Can Atakan Ugur
140		get discussions endpoint	12 days?	5/26/21 ...	6/10/21...	Musa Nuri Ihtiyar
141		**Front-End**	11 days?	**5/27/21 ...**	**6/10/21...**	**Musa Nuri Ihtiyar; Deniz Arda Budak;Elif Sema Balcioglu;Ibrahim Melih Aktas;Ekrem Yusuf Ekmekci;Can Atakan Ugur**
142		views	11 days?	5/27/21 ...	6/10/21...	
143		templates	11 days?	5/27/21 ...	6/10/21...	
144		**Dockerization**	2 days?	**6/9/21 8...**	**6/10/21...**	**Deniz Arda Budak;Elif Sema Balcioglu;Ibrahim Melih Aktas;Berk Atil**
145		AWS	2 days?	6/9/21 8...	6/10/21...	
146		**Documentation**	1 day?	**6/10/21 ...**	**6/10/21...**	**Musa Nuri Ihtiyar; Deniz Arda Budak;Elif Sema Balcioglu;Ibrahim Melih Aktas;Ekrem Yusuf Ekmekci;Can Atakan Ugur**
147		getting user doc	1 day?	6/10/21 ...	6/10/21...	
148		posting user doc	1 day?	6/10/21 ...	6/10/21...	
149		getting events doc	1 day?	6/10/21 ...	6/10/21...	
150		posting events doc	1 day?	6/10/21 ...	6/10/21...	
151		posting equipments doc	1 day?	6/10/21 ...	6/10/21...	
152		posting discussions for event doc	1 day?	6/10/21 ...	6/10/21...	
153		getting discussions for event	1 day?	6/10/21 ...	6/10/21...	
154		posting badges doc	1 day?	6/10/21 ...	6/10/21...	
155		getting badges doc	1 day?	6/10/21 ...	6/10/21...	

156		getting badges point doc	1 day?	6/10/21 ...	6/10/21...	
157		posting interesteds doc	1 day?	6/10/21 ...	6/10/21...	
158		posting spectators doc	1 day?	6/10/21 ...	6/10/21...	

The full project plan with all the details can be found here.

Individual Work

Berk Atıl	- My core task was creating a post endpoint for Badges whose endpoint is /api/v1.0/badges. Normally, a badge has an id, name, symbol URL and point attributes. Name and symbol URL is asked to be provided by the users and id is determined automatically by our database management system. For the point field, I decided to use a bitcoin price at the time that the badge is created. I looked for a free and easy to use API for this and I found https://docs.coinapi.io/. After getting the bitcoin price, I extract the 4th decimal digit and assign it as a point of the badge - Also, I added the front-end part for the badge creation endpoint by using Bootstrap and Jinja templates so that we have a structured and consistent front-end part. - After we talked with our customer(TA), we realized that each team member should write at least 1 POST and 1 GET endpoint. Therefore, I thought about writing a GET endpoint. I wrote /api/v1.0/badges/point endpoint that returns the point of a badge given its name. - I wrote unit tests for the POST endpoint that I wrote. In order to achieve this, I researched how to write unit tests in a logical manner. - Since the documentation is a very important part of the creation of an API, I looked at Swagger which is suggested by Elif Sema. I learned how to add it to our project and added swagger documentations for both of the endpoints that I wrote. - I created a template for the tools evaluation part of this report and some of my friends made some contribution afterwards. - I tried to make code reviews as much as possible and more specific details about this can be found in my individual report. - I helped my teammates to solve the conflicts during the merge operation. - Lastly, I searched how to deploy our docker image to AWS and created an AWS account. After I learned the basics of this operation, we deployed it together with Elif Sema. - Since we used my AWS account, I had to deploy our application several times because of the small changes that we made on the last day.
Can Atakan Uğur	- I started by doing basic research on web frameworks. However, since I had never done anything web related before this assignment, it seemed complicated at the

	beginning; there were a multitude of different resources on the web. - After the initial structure was created by Elif Sema and we decided to use Flask, Bootstrap, Jinja, and SQLAlchemy in the project, I focused on those. - After we started, the authentication part was done very quickly by Ömer; this was crucial for me to understand the structure. - Despite the fact that I had a hard time getting used to the environment, I was able to get my fundamental code done thanks to a very beneficial video tutorial. - Also, we got divided into teams when we were finishing our parts. Related endpoints came together, for example, me and Musa were grouped together. He was also so helpful while we were both trying to code our endpoints about “Discussions” interactively. - So, my main responsibility was to create an endpoint which was supposed to allow users to post a comment in an event’s discussion page. - I used an external public API to get a random name for those who commented and show that name on the screen with the comment made. - Interestingly enough, the toughest part for me was using Github and Git for me. I tried to use Github actively by reviewing issues and pull requests of my friends but I spent most of my time on using Git. I had a hard time dealing with conflicts in my pull request, and even learning “what a branch is” basically. Berk was especially helpful in this period. - Since we all decided to use Swagger for documentation, I also used it for documenting my code. - Unfortunately since it was tough for me to learn everything and put pieces together at the very last minute, I was not able to participate in the meeting where my team deployed the application. However, I believe this will be beneficial for my future work and thus, I want to learn it whenever I am able to allocate some time. - Also, I wanted to update our project plan for our Milestone 2 Group Report. Me and Ömer were partnered for this assignment. - Ömer and I decided the outline of the updated project plan in our Zoom meeting. We both thought and confirmed that it was a good idea to update the meetings in our project report and decided what to add about our Practice App after an active discussion. - We both worked individually after we made the plan and shared the job and I completed updating the first half of the project plan whereas he did the second. - In addition to those, I reviewed the parts that are written by my friends in our Milestone 2 Group Report.

Deniz Arda Budak	- I did comprehensive research about web and web development, since I was not really familiar with web development before. It was almost a totally new area to me, however I grasped the fundamentals of the web and web development with a good research. I learnt the front-end and back-end developments concepts and how they are related with each other. - Since we had chosen Flask as our framework, I watched some Youtube videos about Flask and read the documentation of it. I did some practice with it before starting our project implementation. - Since we were expected to implement a RESTful API, I learnt the fundamentals and the use of it. - My main task was creating a GET endpoint to get an event given its id, in the endpoint api/v1.0/events/<event_id>, which was decided in our team meeting. However, after the teaching assistant warned us that every student should write one GET and one POST method, I also created a POST endpoint to add a Spectator to an event, in the endpoint api/v1.0/events/<event_id>/spectators. - I learnt how to use SQLAlchemy, since we decided to use it. I used it in functions of the endpoints I have implemented, to make a query to the database. - I searched for an external API that gives the weather data about a location, given its coordinates(longitude and latitude). I have found OpenWeather API, which was perfectly fitting for my need of getting weather information for a specific location. I have obtained an api key and tried sending different GET requests to the api on Postman, in order to test it with different parameters. - I helped Musa Nuri when he needed help about how to use the API functionality I have written(the GET functionality for getting an event by its id). Because he needed it in his own implementation(in the html file) about discussion of the event. - Documentation of an API is essential. As a team, we decided to use Swagger, which was suggested by Elif Sema. I learnt how an API is documented using the Swagger. I wrote Swagger documentations(.yml files) for both endpoints I have implemented. - I implemented the corresponding front end for showing the information of an event, given its id in the path(events/<event_id>). For that purpose, I wrote a simple HTML file, which extends base.html. - I wrote unit tests for the functions used inside the “GET an event by id” endpoint. - I opened my own issues on our repository and commented on them when it is needed. - In the pull requests I have made, my teammates did very nice comments and evaluations. When they wanted me to change some parts, I replied them and

	we had discussions on the pull request page. According to the final decisions we have arrived at, I did the necessary changes. I also reviewed some pull requests from teammates on GitHub. - I took part in the meeting where we deployed our docker image to AWS, with Elif Sema, Berk and Atakan. - I added the documentation for my endpoints on the related wiki page here.
Ekrem Yusuf Ekmekci	- I researched front-end development paradigms at the start of the project. I looked especially html and bootstrap from documentations - I could not attend meeting where we decided on the api structure. I researched REST api and gave criticism after the meeting notes were posted. - I suggested a new and more RESTful api structure that everybody builds APIs that have one of the following functionalities: getall / create new / get one / post one. I also suggested concepts for these apis such as getting all events, posting an event, getting a specific event etc. - After that point, the whole team started researching python, django, flask, and swagger. I created private video and reading lists and consumed them. Then we decided on using python with flask; html, bootstrap and jinja for frontend; sqlalchemy for database management; blueprint for layering system. My suggestion was that we should keep it simple and use flask instead of django. My biggest learning was that by using blueprint, we do not only keep it simple, but also keep it flexible and modular. - I read swagger ui and created a document with its online editor but it did not match with the group's decision. So, I changed my structure and added it to the project folder. All the documentation can be obtained from the application. - I built my part using the initial structure which was created by Elif Sema. Her contribution got more useful when Ömer added a sign-in feature to the project. I started with a living, simple application. I approved and appreciated their work. - My main part was about getting all events and providing some basic filtering operations. I first built backend part that serves basic html responses. At that point I assumed my server sends get requests to my future api and takes json object as documented earlier. For the frontend part I used jinja for variables to be passed. This decision was made earlier as mentioned above. I provided filtering by form post methods and sent the parameters to the api. - My API's base route is "api/v1.0/events". I took parameters as query parameters. I built an sql query

	based on these optional parameters and executed the query by the help of sqlalchemy. After fetching corresponding events, I converted them into json format and added “creator name” with an external api. My code calls get requests to an external api and takes random names for every event. I did it because our project’s scope is not database management but api handling. I could take the names from database joins but I did not prefer it. After adding names to json objects, I returned the obtained results. An array of json objects. - I also built an api that handles post requests. This API’s base route is “api/v1.0/events/<event_id>/interesteds”. I built it to gain knowledge about post requests. My code takes one input, user id and inserts event-user pairs into the database. Database collects people interested in specific events. I did not check for the validity of the event id. - I got help from my teammates and tried to help them also. We used whatsapp actively and consulted each other every time. For example, Berk taught me how to resolve conflicts so that I can edit my pull requests. I taught Atakan how to pass variables by using jinja. - I opened my own issues on github, commented on them and expected my teammates to review and close the issues. I also reviewed my teammates' self-assigned issues and closed them with thankful comments. - I reviewed more than 10 pull requests and commented on them. We used reviewing system actively and it helped us improve our code more efficiently. I really liked it. - I researched docker and aws before the deadline but could not attend the meeting because of an urgent situation. I did not contribute to the dockerizing process, I just reviewed it and saw no issue.
Elif Sema Balcıoğlu	- I created the initial structure with Blueprints to have a modular application. I showed it at one of our meetings and after everyone agreed, I created a pull request to merge to master. - I researched an easy to use library for Swagger to generate the documentation easily. I came across Flasgger and after trying I saw it was very easy to use and told my teammates, after they agree I created a template and created a pull request for it. - My main work was to create a POST endpoint for /api/v1.0/events, which is used to create a new event. I wanted for users to be able to add a address as they want and convert it to coordinates on back-end, so I found Google Maps Geocoding API. I generated a API key and used this API to handle this task. - I created the front-end for creating an event. I used

	Bootstrap and Jinja templates. While I was creating an event form, I realized having a sports list select would be useful for users to enter the related sport. I added the sports list but since there are a lot of sports, to add the functionality of searching, I added Bootstrap-select script to base.html. - I searched for an API that would give a list of sports and found The Sports DB. With a GET request I got the list of sports. Events were added to the database with ids that match the ids of sports on this API. - After discussion with my teammates, we decided to add a database table for sports to capture id-sport name information. I wrote the function which adds the sports to the database on database creation. - At this point I wrote a POST request and used two external APIs. But I wanted to learn how to write GET endpoints and use query parameters. And since our sport id-name pairs were added to the database, I created a GET endpoint on /api/v1.0/sports, that given a keyword, filters the sports which have the keyword in their names and returns a json containing the list. - I created a basic form endpoint for the sports list, which has a text input field and lists the sports using a table tag of Html. - I created .yml files for both of the endpoints I created to create Swagger documentation. - I wrote unit tests for the functions used inside the endpoints. - I tried to review and improve my teammates' codes on pull requests and wanted a review from them. - I researched Docker and learned how it is used, also looked for ways to make our database persistent using volumes. In one of our meetings we created the Dockerfile and docker-compose file. - With Berk, we deployed our application to AWS. He researched how to deploy and created an AWS account, after making changes to make database volume, we deployed our application. - I created the documentation page for the Wiki page and added the documentation for my endpoints. It can be found here.
İbrahim Melih Aktaş	- Firstly, I study what Rest API is and how we can use it because I hadn't even heard about it. I also study Flask and did some little practices with it. - In one of our meetings, Elif Sema created the general structure of our project. We all decided to follow that structure to build our APIs. - We discussed APIs that we can create in a meeting. APIs should have been related so we decided to brainstorm about it. What we understood about implementation assignment was meant to be one GET or one POST functionality for everyone so we started to share those APIs. My task was to create a GET API

	that shows all the badges from the database. Since we have to use one API from the outside, I decided to use an API that returns random cat pictures. My API was meant to get all the badges from the database and return them with the cat pictures so HTML can print them in an order that looks fine. - I have only used a basic html file that has a button on it until Elif Sema helped me to add bootstrap to it. Front end looks much better now. After the user clicks that button, it shows all the badges with cat photos instead of the symbol of the badges. Even though I haven't used many front-end features, I reviewed my friends' codes and understood how it can be done. - Since I needed a function in the views.py, I created the show_badge() function. If the button is clicked it goes to the API and prints all the badges by returning an appropriate render function. If the button is not clicked it returns the html file that shows only one button on the screen. - I created unit tests for a function that I use in the API. It checks if it returns the right number of urls and whether it returns a string. This unit test was a bit odd, however I could not think anything different from this so for experiencing the nature of unit tests I wrote these. - I researched the Swagger tool and created .yml files by checking Elif Sema's and Berk's documentations. - I reviewed some of our friends' pull requests and gave some comments about it. - I searched about docker and took part in the meeting where we created the necessary files. - I searched about amazon and took part in the meeting where we deployed our project.
Musa Nuri İhtiyar	- Web development was a totally new field for me, so I tried to learn the basics about it. I have looked at what back-end and front-end means. Also I tried to understand how they interact with each other. - Because Flask was selected as the framework to use, I tried to learn it by looking at related videos on the internet. - APIs, especially the concept of Restful API, was at the center of the project; therefore, I studied what they are and why they are useful, yet personally implementing was a very different experience from only learning theoretical parts about it. - I was responsible for implementing an endpoint which shows the discussions for a given event in the database. It's available in views.py - To do the previous part, I have implemented an API for this purpose and I have used it in my code. Its call can be done by using "/api/v1.0/events/<event_id>/discussions" - The documentation of the API was done by using

	swagger and writing .yml file describing the behavior of the API - We were also wanted to use an external API; consequently, I tried to find a free one. After putting some effort on this part, I have been able to find an API which gives information about sports. - All of the back-end related coding was commented too. - Front-end part for showing the discussions for an event was my responsibility too. I have used a simple structure in html, yet I think it was enough when the scope and aim of the project are taken into the account. - I have kept the meeting notes of our group. - Because almost all of the code I have written was either interacting with the database or making requests to an API, I couldn't find a meaningful part to test. Consequently, I have decided to write a new function which is partially related to the project topic and write unit tests for that. Then I have written unit tests for that one. It was useful for practicing unit test methodology. They can be found in "are_discussions_for_event_special_test.py" file, which is under test folder. - I have watched videos about what Docker is and why Dockerization is beneficial. - I tried to open issues for my personal tasks even though they weren't very long.

Ömer Süve	- Firstly, I studied the web development essentials. - I researched the python web development framework named Flask. I implemented basic tutorials and after that I started to implement code for the practice-app. - My part of the project was implementing the authentication part and implementing an endpoint for equipment posting. - I implemented the initial page of the application which was the authentication part. - I implemented the authentication part by making an api that includes three endpoints, which are sign-up, login and logout. - After implementing the authentication part, I started to build the operation of equipment posting to the database. - After spending a large amount of time searching for an appropriate api, I could not find a good api that gets an equipment name as a request. - I could not find any good api for fetching prices, therefore I found another api for fetching a number that corresponds arbitrarily for an equipment. - Then, I started writing by making an html code for the equipment page. I tried to write that with respect to the html pages of the other group members. - I used bootstrap for the UI, as all group members did. - My equipment model had to have a name and price. Besides that, since a model should have an id, I added that too. Thus, the equipment model has three parameters. - The input of the post request of the equipment to the database includes just their names, because I thought that an equipment needs only a name field in order to be posted. - In the backend part of the equipment api, since we have decided the initial route as a team as "/api/v1.0/equipments/", my endpoint for the equipment posting just corresponds to the current directory. - I implemented the documentation for the api I created using swagger. - Lastly, as all team members added, I implemented the frontend part in views.py. This part provides connection between the user and the frontend part which responds to the user. - I tried to write unit tests for the functions used inside the endpoints. Since I use only one independent function which finds the price for an equipment, I wrote only one test for the equipment part.

Evaluation of Tools and Processes

Github

We use Github as a version control system. Tracking our progress is easy and convenient thanks to its issue features.

- We can easily see who is doing what and their status. It also provides custom labels that makes our job easier.
- The wiki pages are easily editable and creatable so we can document our work in a structured way. Also, it is very easy to see the date and acting user of revisions and also different versions of them can be analyzed.
- Each team member wrote their codes in different branches and then we merged our codes. Thanks to this process, tracking the code changes was easy and avoiding different kinds of problems due to team work on the same files was achieved.
- Code reviewing is comfortable. Users can see the differences, comment and discuss them, start a review and check files as reviewed. All these features make information to flow easily and permanently.

To sum up, we can document and develop our code easily via Github.

Zoom

Because of the pandemic, we cannot meet face to face. Therefore, we use Zoom to meet online and it is easy to use. However, its 40 minutes time limitation is annoying.

Google Meets

We also sometimes use google meets especially to meet with our customer due to the time limit of Zoom.

Google Docs

While writing the Milestone 1 and Milestone 2 documents, we wanted for everyone to be able to work on it at the same time, so we used Google Docs for collaborative editing. For the M2, we also used an add-on called Code Blocks to make sure our code is correctly indented and every code has the same format. In general Google Docs was very useful and made our writing jobs easier.

Whatsapp

We use it to communicate with each other. We share all the ideas and plans via whatsapp. We also arrange our meetings, ask our quick questions etc. via whatsapp.

Flask

It is an easy to learn and use lightweight web development framework for Python. We used different layers in our architecture which Flask allows.

In the model layer we represent our database. Thanks to the flexibility of Flask, you can freely use SQL commands but you do not have to. Because our database is simple and small, this part is also simple and small compared to other parts.

We have our view layer which provides the front-end operations of our application. We decided routes and builded code accordingly. For example, ??/events route displays the latest events. Users can also filter them. Note that to do this filtering, our app should support post methods for this route. Our structure is a mixture of just get, just post, both get and post pages. It includes pages for creating an event, displaying badges etc.

We also have templates that adjust the visual part of our application. In this part "**Jinja2**" helped us. See document for Jinja. See Jinja's part below.

For arranging layers we used flask's blueprint objects. They allow us to define routes and their corresponding files. Thanks to that, we built our app more distributed and small-sized.

Jinja

Jinja is a templating engine. It allows python code to be executed in an html file. By the help of it, we have smaller and flexible html files. For example, base.html is imported by all files and it handles the top part of the app's page. When there is a new button, we just need to change base.html, not all the html files that represent the main structure.

We also used jinja for transferring our variables to the front end. Our variables are passed, rendered by jinja, and finalized as flexible html files.

Bootstrap

Bootstrap is an open source CSS framework. Using classes on Html tags, it creates responsive and visually pleasing front-end pages. For this project we wanted to use an easy to use framework that would create a pleasing end result since we had limited time. After

some research, we decided to use Bootstrap since it did not require much knowledge and has a rich documentation. With Jinja explained above, we created our front-end smoothly and we were able to focus on the API implementation.

SqlAlchemy

It is an open source object-relational mapper. It provides both power and flexibility of SQL. Its usefulness is discussed by our friends (Ekrem) but we saw that it is a common practice. We used sqlalchemy to read from and write into the database. It also helps in modeling the database.

Swagger

It is an easy to use API documentation tool. We added swagger support for all of our endpoints. Its UI is very convenient. It helped us to understand the details of endpoints that are written by other team members.

We should point out that it is really easy to implement. We just built the main structure of the api and swagger provides an entire documentation page with a nice view.

For the integration with our Flask application we used the Flasgger library. Using Flasgger our documentation is generated from the .yml files we wrote and shown on the /apidocs/ endpoint.

unittest

Python unit test framework helps us build test cases for our code. We all built test cases and tested our code accordingly.

Docker

Docker is the container on which our application is working. It is a very flexible tool that wraps all of the requirements and dependence of the application so that you can run it everywhere.

Postman

It is a collaboration platform during the API development process. It allows us to send requests for endpoints and its UI is very easy to use and convenient. We can say that it is our hands-on testing tool.

AWS

We used AWS EC2 to deploy our application. At first we tried to deploy using a Youtube video but some errors occurred. Therefore we tried to deploy from the Docker Hub. After pushing our Docker image to the Docker Hub, it was very easy to deploy our application using a configuration file, its content can be found on Deployed Application Section.

APIs Used

Google Docs Geocoding

URI: 'https://maps.googleapis.com/maps/api/geocode/json'
Description: Given an address in basic English, after making a GET request to the uri above, detailed information on the address is returned. On this project among the different fields of the response, coordinates and formatted address is used on Adding an Event to the System functionality. Coordinates are then used to find the weather information of the event, formatted address is used to give clear information to users looking for events.

The Sports DB

URI: 'https://www.thesportsdb.com/api/v1/json/1/all_sports.php'
Description: This api is used to get the list of sport types. Also ids given on the response are used as the ids on our database. At the creation of the database, they are added to the Sports table.

Decathlon Developers

URI: 'https://sports.api.decathlon.com/sports/'

Description: When this API is called, it returns a number of points such as description and related images for all of the sports supported by it. In my personal API, I am using the description part of the corresponding sport type in the discussions of an event.

Bitcoin Price

URI: 'https://rest.coinapi.io/v1/'
Description: When you execute a GET request to the uri above with the necessary query parameters, it returns a current requested crypto currency price in terms of requested currency. In this project, bitcoin price in terms of dollars are collected from here to assign points to each badge.

OpenWeather

URI: "https://api.openweathermap.org/data/2.5/weather"
Description:
When a GET request is sent to the URI above, a detailed current weather data of the location is returned. The body request consists of the latitude and the longitude of the location(coordinates.). The coordinates of an event, which are obtained by Google Docs Geocoding, are available in the event data, with the help of Google Docs Geocoding, which is utilized in the POST operation of a new event. Hence, the coordinates are directly taken from the event gotten.

NameFake

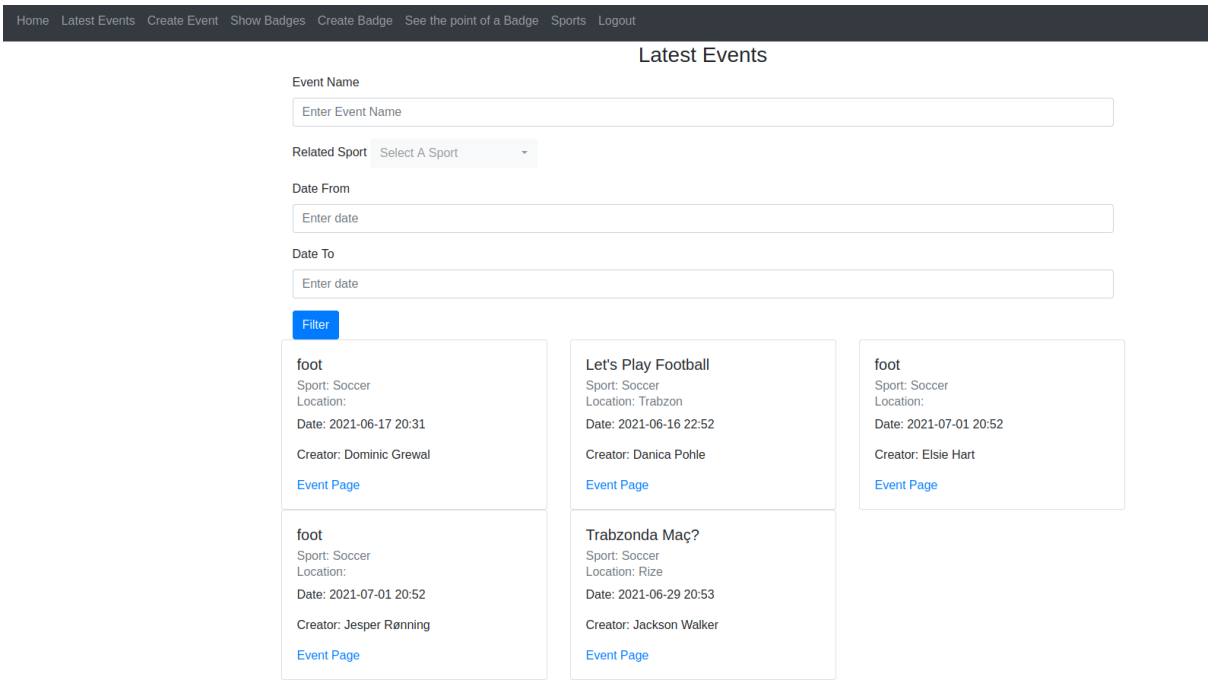
URI: "https://api.namefake.com/english-united-states/random/"
Description:
NameFake is an online service to generate fake names according to the requests. For example, the request that is used in our project asks NameFake to return a random American name.

Random User Generator

URI: "https://randomuser.me/api/?inc=name"
Description:
Since our project was simple and small, we could not have many users. This situation makes me think of an external api that gives me random celebrity names so that our app look like

celebrities use it. However, I could not find an api that returns random celebrities. Instead I found web pages that generate such information. Because we need to implement api calls, I decided to use an api that returns random names. This brings me to “Random User Generator”. It has simple and useful documentation. I used it easily. When I used it I take random names and assigned them every time a user calls events get api.

Our Main Page



API Documentations

1. Badge Related Endpoints

1.1.

1.2. Adding a Badge to the System

Endpoint: /badges
Method: Post
Authorization: Not Needed
Author: Berk Atıl
Parameters: {
“name”: Name of the badge
“symbol”: Symbol URL of the badge
}

Expected Response When No Error:

```
{
  “badgeId”: Id of the badge
  “name”: Name of the badge
  “symbol”: Symbol URL of the badge
  “point”: Point of the badge extracted from a bitcoin price
}
```

Responses:

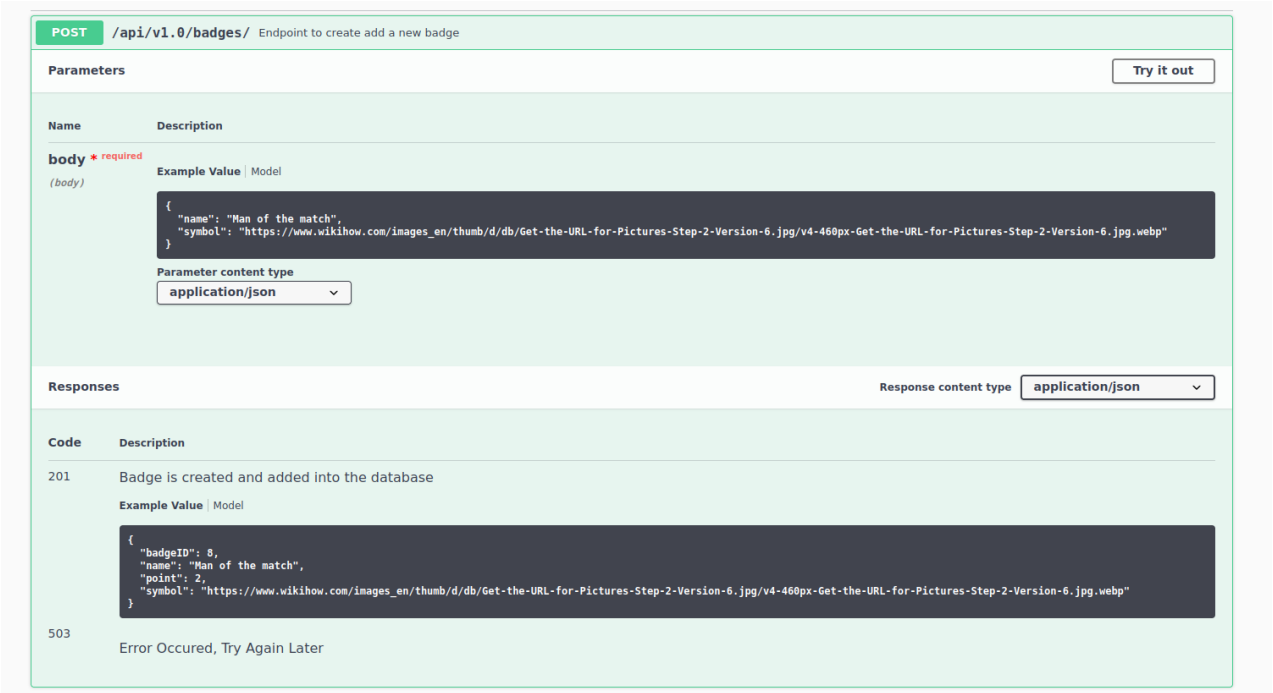
201: `"Badge added."`

Example Response:

```
{
 "badgeId": 1,
 "name": "Name",
 "symbol": "picture.com",
 "point": 7
}
```

503: `"Try Later."`

Swagger Documentation:



Related functions:

This function gets a bitcoin price(float) as a parameter and then extracts the 4th decimal point of this price.

```
def extract_point(price):
    return str(price).split('.')[1][3]
```

This function is used to create a “random” badge point. It basically handles the bitcoin price

request. In order to do this, it initially sends a get request to https://rest.coinapi.io/v1/exchangerate/BTC/USD URI to get the current bitcoin price in terms of dollars. Afterwards, it sends the price to the “extract_point” function to get the exact digit and then returns this value.

```
def create_badge_point():
    """
    This is a function to assign a point to the badge.
    It uses a Bitcoin Price API and get a number from that price.
    """
    url = 'https://rest.coinapi.io/v1/exchangerate/BTC/USD'
    headers = {'X-CoinAPI-Key' : os.environ.get("API_KEY_BADGE")}
    response = requests.get(url, headers=headers)

    return extract_point(response.json()['rate'])
```

This function is used to create a badge object given name and symbol attributes. Basically, it takes a dictionary that contains the name and symbol URL of a badge. Then it creates a badge object with this information and returns the badge object.

```
def prepare_badge(data):
    badge_name = data['name']
    badge_symbol = data['symbol']
    badge = Badge(name=badge_name, symbol=badge_symbol)

    return badge
```

This is actually the endpoint code to handle POST requests to api/v1.0/badges. Firstly, it gets the badge object by calling “prepare_badge” function. After that, in order to determine the point of a badge it calls “create_badge_point” function and assigns that value to point field of a badge object. Afterwards, it tries to insert this badge information into the database. If no problem is encountered during this operation, 201 response code with the json version of the badge object is returned. However, there could be some problems during the database operation so if this is the case 503 response code is returned with “Try Later” message.

```
@badges.route('/',methods = ['POST'])
@swag_from('doc/badges_POST.yml', methods = ['POST'])
def add_badge():
    """
    This is an only POST endpoint for adding a badge.
    In order to add a badge "name" and "symbol" which is a URL for a symbol
are needed.
    If the badge is added successfully, it returns 201 response code.
    If there is a problem during the addition of the badge, it returns 503
response code.
    """

    badge = prepare_badge(request.json)
```

```
    badge.point = create_badge_point()
    try:
        db.session.add(badge)
        db.session.commit()
    except exc.SQLAlchemyError as e:
        db.session.rollback()
        return "Try Later", 503

    return jsonify(badge.serialize()), 201
```

Unit tests:
This is a setup function for the badge post endpoint unittests. That is, it creates some mock data which will be used in tests. More specifically, “data” corresponds to a request body of the endpoint and “bitcoin_price” represents the bitcoin price coming from the third party API.

```
def setUp(self):
        self.data = {'name':'mvp', 'symbol':'trialpicture.com'}
        self.bitcoin_price = 32.4523478
```

This function tests if the badge object is created correctly. That is, it calls the “prepare_badge” function that creates the badge object given the name and symbol fields. This function checks if the fields of the badge object matches with the parameters in “data” which is a request body.

```
    def test_badges(self):
        expected = Badge(name=self.data['name'], symbol=self.data['symbol'])
        badge = prepare_badge(self.data)

        self.assertEqual(expected.name, badge.name)
        self.assertEqual(expected.symbol, badge.symbol)
```

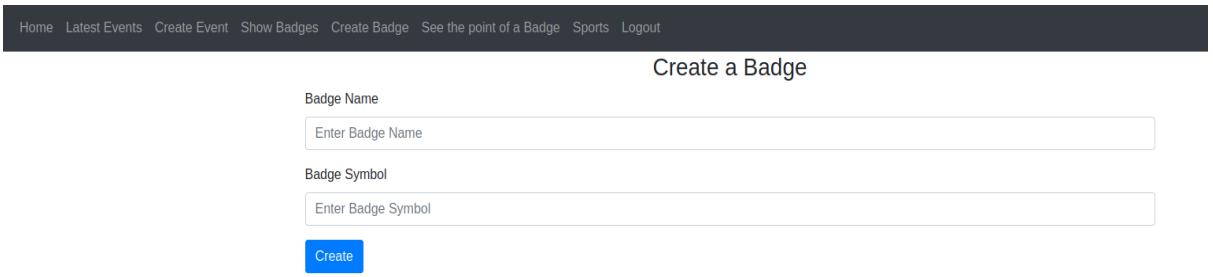
This function tests if the correct decimal digit is extracted from the bitcoin price. That is, it calls the “extract_point” function that returns a digit. Afterwards, it checks if this digit matches the 4th decimal digit of the “bitcoin_price”.

```
def test_badge_point(self):
        expected = extract_point(self.bitcoin_price)
        self.assertEqual(expected,'3') # 3 is the 4th digit after decimal
point
```

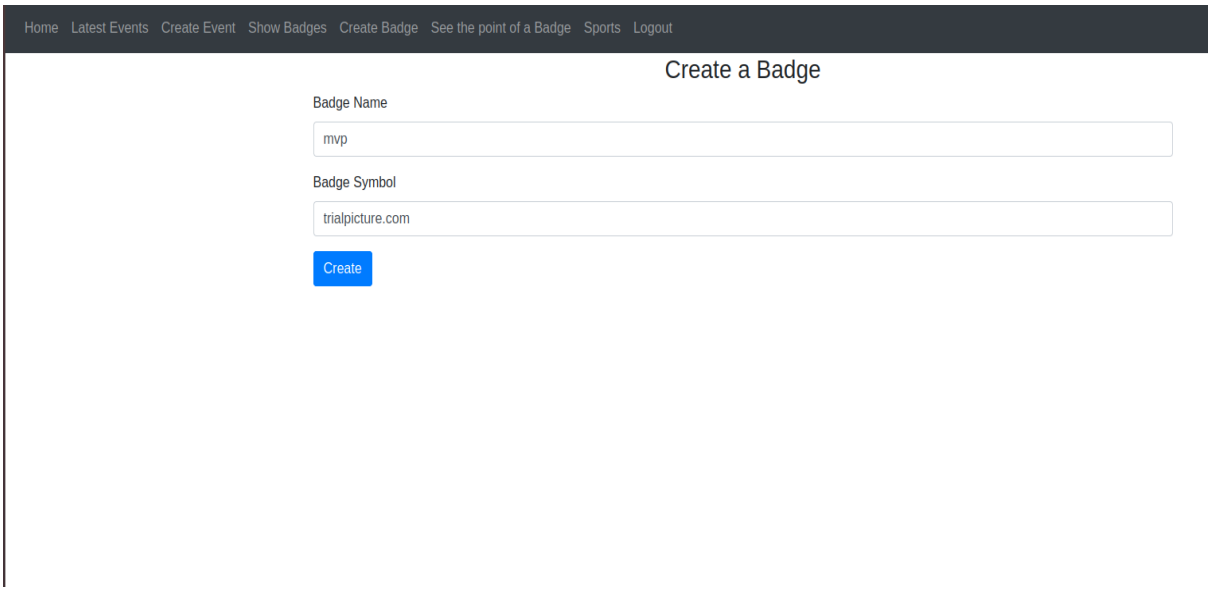
Frontend Part with Screenshots:
This function handles the frontend part of the badge post endpoint. If a GET request is retrieved it just shows the badge creation page which provides some text boxes to enter the necessary fields. If a POST request is retrieved which is done by clicking a button on badge creation page, it creates a request body for the api/v1.0/badges endpoint. Then, it sends a POST request to this path. Afterwards, it checks the status code of the response. If it is 201, it redirects to the home page with a “Badge added” message by labeling it as “success”. If another response code is retrieved, it redirects to the home page with a “Error Occured, Try Again Later” message by labeling it as “error”.

```python
@views.route('badge/', methods=['POST','GET'])
def badge():
    """
    This prepares the front-end side of the badge addition functionality.
    It can be used as a POST and GET. When it is used as a GET, badge
addition page is shown to provide related fields for it.
    When it is used as a POST, the provided fields for a badge is posted and
badge post endpoint is requested.
    """
    if request.method == "POST":
        badge = {
            "name" : request.form.get("name"),
            "symbol": request.form.get("symbol")
        }

        req = BASE_URL+"/api/v1.0/badges"
        headers = {'Content-type': 'application/json'}
        response = requests.post(req, data=json.dumps(badge),
headers=headers)

        if response.status_code == 201:
            flash('Badge Added', category='success')
        else:
            flash('Error Occured, Try Again Later', category='error')

        return redirect(url_for('views.home'))

    return render_template("badge.html", user= current_user)
```

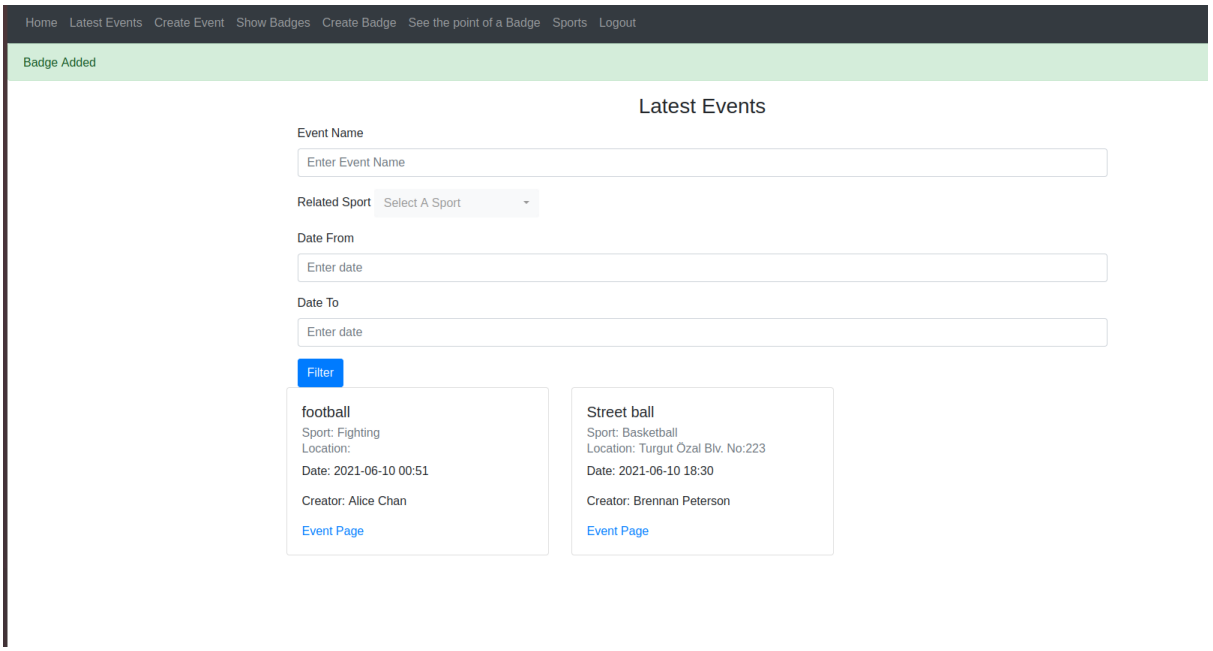
Badge Creation Screen:



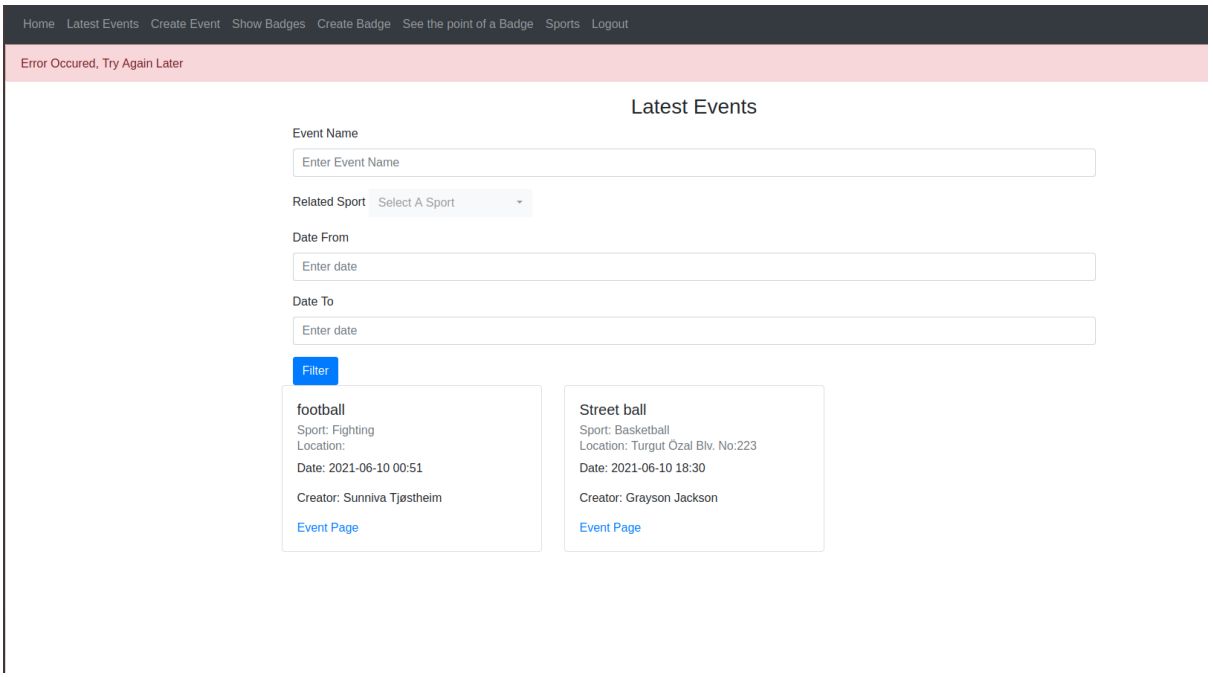
Badge Creation Submission:



Successful Badge Creation:



Unsuccessful Badge Creation:



1.3. Getting a Badge Point by Badge name

Endpoint: /badges/point?name=\<name\>
Method: Get
Authorization: Not Needed
Author: Berk Atıl
Expected Response When No Error:

```
{
  “name”: Name of the badge
  “point”: Point of the badge extracted from a bitcoin price
}
```

Responses:

200: `"Badge point is retrieved."`

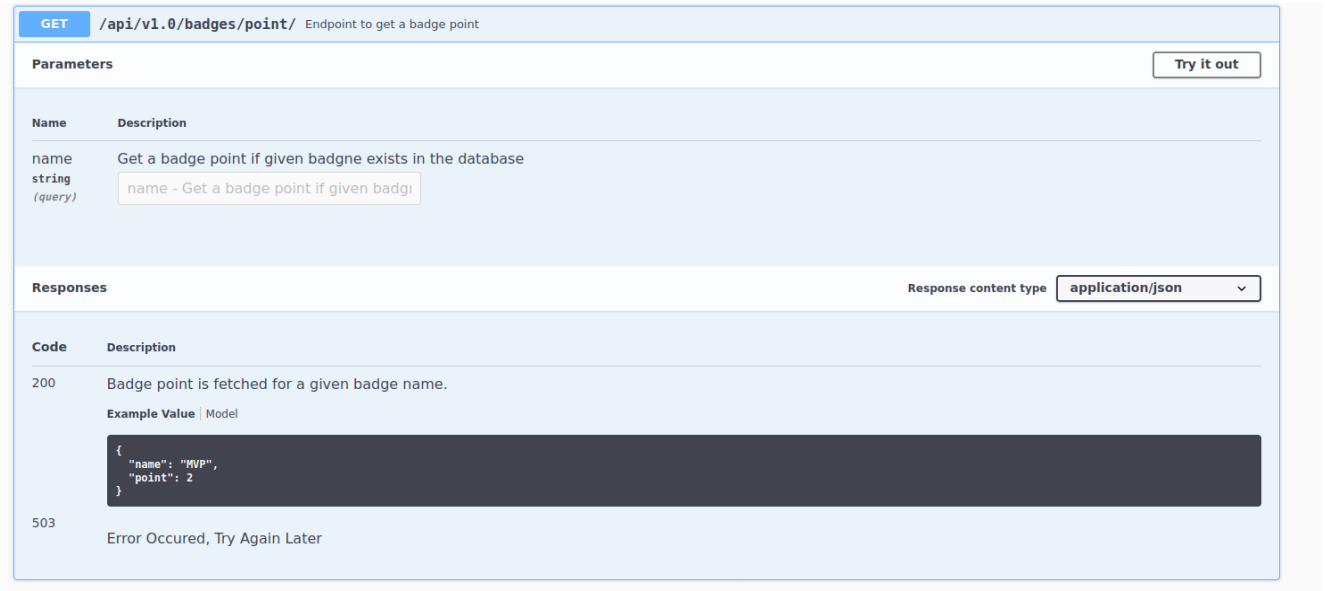
Example Response:

```
{
  "name": "Name",
  "point": 7
}
```

400: `"No badge with this name."`

503: `"Error Occured, Try Again Later"`

Swagger Documentation:



Related functions:

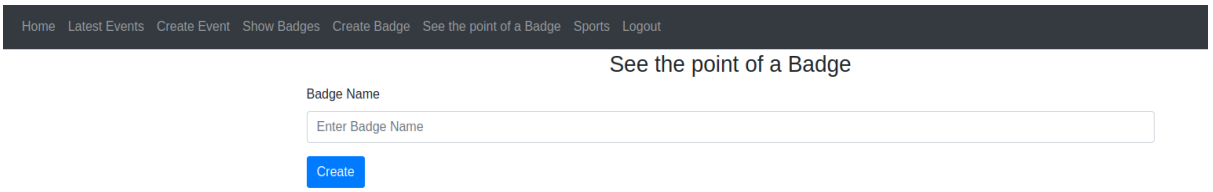
This is actually the endpoint code to handle GET requests to api/v1.0/badges/point. Firstly, it gets the name of the badge whose point is wanted to be shown. Afterwards, it tries to get the badge point by badge name from the database. If any error occurs during this operation, 503 response code with “Error Occured, Try Again Later” message is returned. If no badge is retrieved for a given badge name, 400 response code with “No badge with this Name” message is returned. Lastly, if everything is successful, 200 response code with badge name and badge point are returned.

```
@badges.route('/point/',methods = ['GET'])
@swag_from('doc/badges_point_get.yml', methods = ['GET'])
def get_badge_point():
    """
    This is an only GET endpoint for getting the point of a badge by name.
    Badge name should be provided.
    If the badge point is fetched successfully, it returns 201 response code
    If there is no badge with the given name it returns 400 response code.
    If there is a problem during the fetch, it returns 503 response code.
    """
    badge_name = request.args.get('name')
    try:
        badge =
db.session.query(Badge.point).filter(Badge.name==badge_name).first()
    except exc.SQLAlchemyError as e:
        db.session.rollback()
        return "Error Occured, Try Again Later", 503

    if badge is None:
        return "No badge with this Name", 400
    return {'name': badge_name, 'point':badge[0]}, 200
```

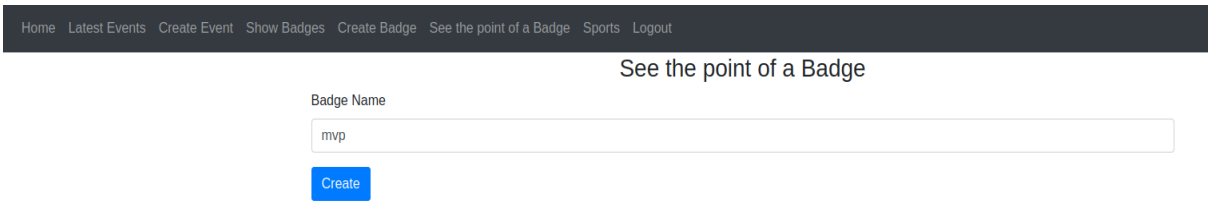
Frontend Part with Screenshots:
This function handles the frontend part of the badge point get endpoint. If a GET request is retrieved it just shows the badge point retrieval page which provides a text box to enter the badge name. If a POST request is retrieved which is done by clicking a button on badge creation page, it creates a query parameter for the api/v1.0/badges/point endpoint. Then, it sends a GET request to this path. Afterwards, it checks the status code of the response. If it is not 200, it redirects to the home page with an error message by labelling it as “error”. Otherwise, it renders the badge point page to show the point of a badge.

```
@views.route('point_badge/', methods=['POST','GET'])
def point_badge():
   """
   This prepares the front-end side of the badge point retrieval.
   It can be used as a POST and GET. When it is used as a GET, badge point
retrieval page is shown.
   When it is used as a POST, badge point for a given badge name is shown.
   """
   if request.method == "POST":

       badge_name = request.form.get('name')
       req = f'{BASE_URL}/api/v1.0/badges/point?name={badge_name}'
       headers = {'Content-type': 'application/json'}
       response = requests.get(req, headers=headers)

       if response.status_code != 200:
           flash(response.text, category='error')
           return redirect(url_for('views.home'))

       return render_template("show_badge_point.html", user=current_user,
badge=response.json())

   return render_template("point_badge.html", user=current_user)
```

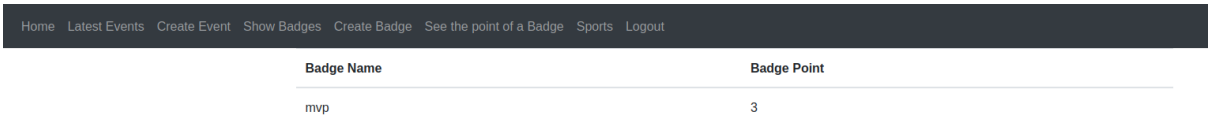
Badge Point Retrieval Screen:



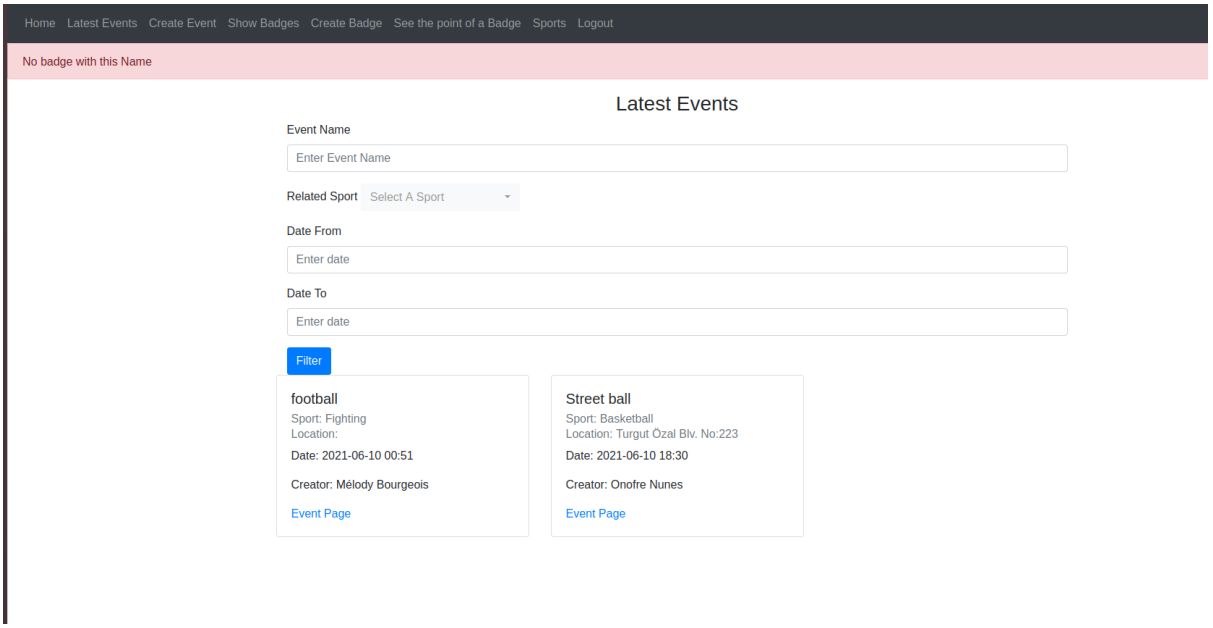
Badge Point Retrieval Submission:



Successful Badge Point Retrieval:



Unsuccessful Badge Point Retrieval:



1.4. Getting All Badges

Endpoint: /badges/show_badge
Method: Get
Authorization: Not Needed
Author: İbrahim Melih Aktaş
Parameters: -
Expected Response When No Error:

{
"badgeID": Name of the badge.,
"control": Decides whether HTML should show badges or not,
"dog_photos": Cat photo from the RandomCat API,
"name": Name of the badge.,

"point": Point of the badge,
"symbol": It is a URL for the symbol picture of the badge.
}

Responses:

200: "All badges are retrieved."

Example Response:

{
"badgeID": 5,
"control": true,
"dog_photos": "https://i.thatcopy.pw/cat/H20brn4.jpg",
"name": "Best Scorer",
"point": 3,
"Symbol":
"https://www.wikihow.com/images_en/thumb/d/db/Get-the-URL-for-Pictures-Step-2-Version-6.jpg/v4-460px-Get-the-URL-for-Pictures-Step-2-Version-6.jpg.webp"
}

Related Functions:
get_cat_pictures function takes a list and returns the right number of urls that contain cat pictures.
show_badge function is the get endpoint. After the user clicks the button in the front end, this endpoint returns badges, cat pictures and a control boolean. Badges contain all the badges in the database. Cat pictures is the value that the get_cat_pictures function returns. Control button is for the html, so it can show the appropriate table.

```
def get_cat_pictures(badges):
    """This is the function that returns a picture for all
    elements in the parameter list."""

    url = 'https://thatcopy.pw/catapi/rest/'

    return [requests.get(url).json()['url'] for _ in badges]


@badges.route('show_badge/',methods = ['GET','POST'])
@swag_from('doc/badges_GET.yml', methods=['POST'])
def show_badge():
    """
    This function is the that shows all the badges after the user
    clicks the button. It uses RandomCat API to show random cat
pictures.
    Since it starts working with a POST request from the server, first
    thing it does is checking request method.
    Then it gets all the badges from the database and return a json that
contains cat photos.
```

```
    control variable provides that if the button is not clicked yet, it
returns False and
    there will be no table. If the button is clicked control is True and
there will be a
    table on the html file.
    """
    if request.method == 'GET':
        badges = Badge.query.all()
        badges_serialized = [i.serialize() for i in badges]

        cat_photos = get_cat_pictures(badges)

        control = True
        result = {"badges":badges_serialized, "dog_photos":cat_photos,
"control":control}
        return jsonify(result), 200

    else:
        control = False
        result = {"control":control}
        return jsonify(result),200
```

Unit Tests:
These tests are written for the get_cat_pictures function which takes the badges parameter that is a list and returns the same number of urls that comes from an external API. First we set up its parameters. Badges list's content does not matter because we only care for its length. If it's length is three we expect it to return 3 urls. If its length is 0, we expect it to return an empty list. On these unit tests, I set up variables firstly. test_pictures_length checks whether it contains the right number of items which is 3 in the case or not. test_empty checks whether it returns an empty list or not when the parameter is an empty list. test_pictures_type checks whether it returns string or not because the urls are string in that case.

```
import unittest
from ..api.badge import get_cat_pictures

class Test_Cat_Pictures(unittest.TestCase):
      def setUp(self):
      self.badges = ["Badge1", "Badge2", "Badge3"]
      self.empty = []

      def test_pictures_length(self):
      pictures_result = get_cat_pictures(self.badges)
```

```
        self.assertEqual(len(self.badges), len(pictures_result))

        def test_empty(self):
        empty_result = get_cat_pictures(self.empty)
        self.assertEqual(len(self.empty), len(empty_result))

        def test_pictures_type(self):
        pictures_result = get_cat_pictures(self.badges)
        self.assertEqual(type(""), type(pictures_result[0]))
        self.assertEqual(type(""), type(pictures_result[1]))
        self.assertEqual(type(""), type(pictures_result[2]))
```

2. Event Related Endpoints

2.1. Adding an Event to the System:

Endpoint: /events
Method: POST
Authorization: Not Needed
Author: Elif Sema Balcıoğlu
Parameters: {
"name" : Name of the event, title. required
"creator_user" : Id of the user creating the event. Id must be registered to a user. required
"location" : Address of the event, given using basic English. required
"sport" : Id of the sport. Between 102-120. required
"date" : Date of the event. Format is "YYYY-MM-DDTHH:MM required
}
Expected Response When No Error:
{
"creator_user": Id of the user creating the event.
"date": Date of the event.
"entered_address": Address of the event, given using basic English. Same as location on parameters.
"formatted_address": Formatted address by Google Maps API.
"id": Id of the event.
"latitude": Latitude of the location.
"longitude": Longitude of the location.
"name": Name of the event, title.,
"sport": Id of the sport.
}
Responses:
201: `"Event created and added to database."`

Example Response:

```
{
        "creator_user": 1,
        "date": "10.12.2021",
        "entered_address": "Trabzon",
        "formatted_address": "Trabzon, Ortahisar/Trabzon, Turkey",
        "id": 2,
        "latitude": 41.0026969,
        "longitude": 39.7167633,
        "name": "abc",
        "sport": 103
}
```

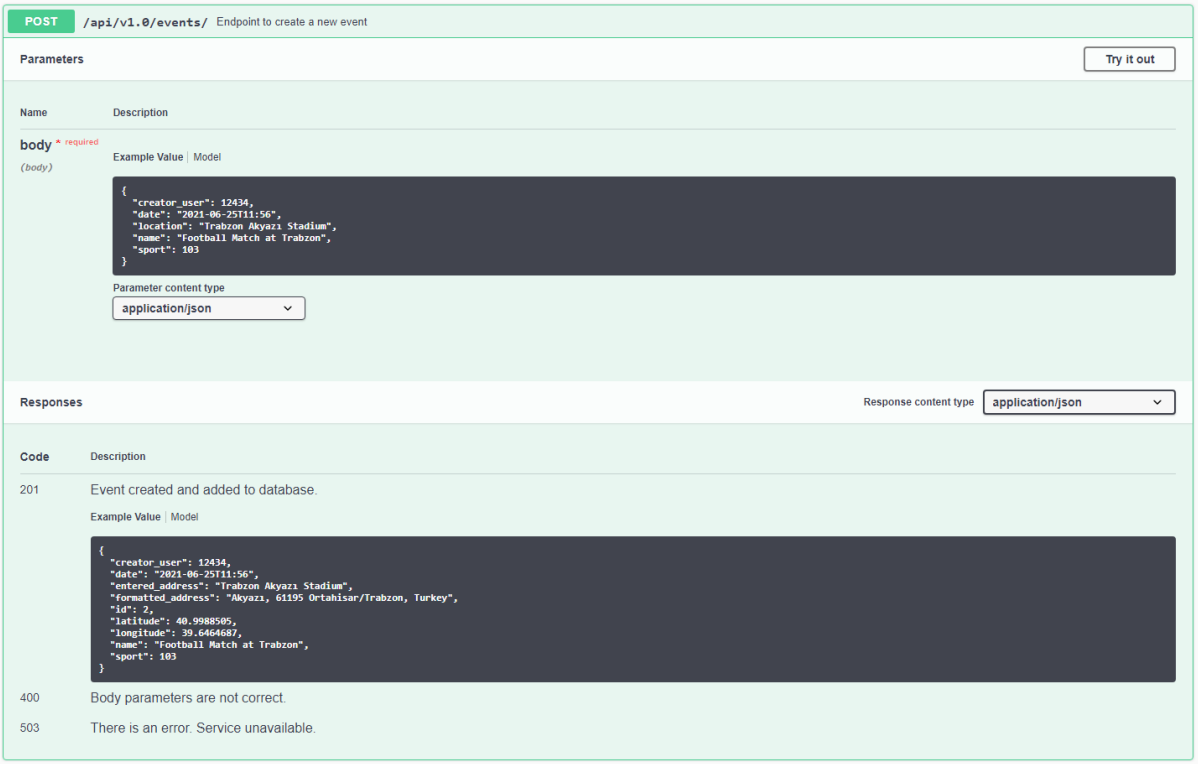
400: "Body parameters are not correct."

```
{
        "error": “Parameters are not correct”
}
```

503: "There is an error, try later."

```
{
        "error": “Service unavailable”
}
```

Swagger Documentation:



Related Functions:

This function is used to check if the id of sport of a given event is in between the range of sport ids that is stored on the Sport database table. New_event is a Event object and has a

field named sport which contains the sport id of the event. Sport ids start from 102 and ends at 120, if the given event has a id in this range returns True, otherwise returns False.

```
"""
    Check the validity of the sport field of event
    parameters:
        new_event: Event
    return:
        True if valid False otherwise
"""
def check_event_sport(new_event):
    try:
        # sport Ids between 102-120
        if int(new_event.sport) < 102 or int(new_event.sport) > 120:
            return False
        return True
    except:
        # sport string cannot be changed to integer
        return False
```

This function checks the format of the date field of the new_event parameter. new_event is an Event object type and its date field contains the date parameter given on the body of the request. Format should be in the form “YYYY-MM-DDTHH:MM”. Regular expressions are used to check the validity. If the date field matches the regex, returns True, returns False otherwise.

```
"""
    Check the format of the date field of event
    Format should match YYYY-MM-DDTHH:MM

    parameters:
        new_event: Event
    return:
        True if valid False otherwise
"""
def check_event_date(new_event):
    # Date format YYYY-MM-DDTHH:MM
    date_regex =
"^(20[0-9][0-9])-(0[1-9]|1[0-2])-(0[1-9]|1[0-9]|2[0-9]|3[0-1])T(0[0-9]|1[0-9
]|2[0-3]):(0[0-9]|1[0-9]|2[0-9]|3[0-9]|4[0-9]|5[0-9])$"
    if not re.match(date_regex, new_event.date):
        return False
    return True
```

Function getCoordinates handles the request to the Google Maps Geocoding API. Takes the address string for the address given by the user, sends a GET request to the uri 'https://maps.googleapis.com/maps/api/geocode/json'. Response is then turned to json format. If the status code of the response is 200, that means no error occurred asking for the

information. From the response json, coordinates and the formatted address is returned with the code “OK”. Formatted address is a more detailed and accurate form of the address entered by the user. If the response from the API has the status “ZERO_RESULTS” or “INVALID_REQUEST”, that means address parameter is not correct and “Address Not Valid” error is returned. If other responses are returned, they are caused by either the network problems or related to the quota of the API_KEY, therefore a “Try Later” error is returned.

```
"""
    Get coordinates using Google Maps API

    parameters:
        address: "address of the location"
    return:
        True if valid False otherwise
"""
def getCoordinates(address):

    # Make GET request to the uri using API_KEY
    parameters = {'key': API_KEY , 'address': address}
    uri = 'https://maps.googleapis.com/maps/api/geocode/json'

    r = requests.get(uri, params=parameters)

    result = r.json()

    # If there is no error return information from response.
    if r.status_code == 200:
        if result['status'] == "OK":
            formatted_address = result['results'][0]['formatted_address']
            longitude = result['results'][0]['geometry']['location']['lng']
            latitude = result['results'][0]['geometry']['location']['lat']
            return formatted_address, longitude, latitude, "OK"

    # Address not valid, error messages from API docs
    if result['status'] == "ZERO_RESULTS" or result['status'] ==
"INVALID_REQUEST":
        return "",0,0,"Address Not Valid"
    # Cannot make a request
    elif result['status'] == "OVER_DAILY_LIMIT" or result['status'] ==
"OVER_QUERY_LIMIT" or result['status'] == "REQUEST_DENIED" or
result['status'] == "UNKNOWN_ERROR":
        return "",0,0,"Try Later"
    # Remaining Errors
    else:
        return "",0,0,"Try Later"
```

Endpoint Code:

This is the handler for the POST request sent to the /api/v1.0/events. At first it is checked if all body parameters (name, creator_user, location, sport, date) are given since they are all required. If there is a missing parameter, error is returned with code 400 for bad request.
If all parameters are given, at first using the getCoordinates(address) function explained above, a coordinate and the formatted address of the address parameter is taken. Then it is checked if that function returned a valid response, if not, 400 code is returned with the error message if the address parameter was not valid or 503 code with message “Service Unavailable” is returned if there was another problem.
An event object is then created with the parameters given and address information. Using the creator_user parameter, it is checked if the database contains a user with that id, if not 400 code with error message "User Not Registered" is returned to prevent non existing users from creating events.
Sport id and date checks are performed using the functions explained above check_event_date(new_event) and check_event_sport(new_event). If there is one that returns False, 400 code is returned with corresponding error message "Date Format Not Correct" or "Sport Id Is Not Correct".
If all checks are correct, the event is added to the database. If no connection problem occurs and the event gets added to the database without error, response with 201 code and json format of the event is returned. If an error occurs, “Service Unavailable” message with error code 503 is returned.

```
@events.route('/', methods = ['GET','POST'])
@swag_from('doc/events_POST.yml', methods=['POST'])
def event():
    if request.method == 'POST':
        # All parameters must be present.
        if not request.json or not 'name' in request.json or not
'creator_user'  in request.json or not 'location' in request.json or not
'sport' in request.json or not 'date' in request.json:
            return jsonify({"error":"Parameters not correct"}), 400

        # Get coordinates using Google Maps API.
        formatted_address, longitude, latitude, error =
getCoordinates(request.json['location'])

        # Return error if fetch was not correct.
        if error != "OK":
            if error != "Try Later":
                return jsonify({"error": error}), 400
            else:
                return jsonify({"error": "Service Unavailable"}), 503

        # Create database model
        new_event = Event(
            name = request.json['name'],
            date = request.json['date'],
            formatted_address = formatted_address,
            entered_address = request.json['location'],
            longitude = longitude,
```

```
            latitude = latitude,
            creator_user = request.json['creator_user'],
            sport = request.json['sport']
        )

        # Check if creator_user is valid and there is a user registered with
that id.
        user = User.query.get(request.json['creator_user'])
        if not user:
            return jsonify({"error":"User Not Registered"}), 400

        # Check sport id.
        if not check_event_sport(new_event):
            return jsonify({"error":"Sport Id Is Not Correct"}), 400

        # Check date format.
        if not check_event_date(new_event):
            return jsonify({"error":"Date Format Not Correct"}), 400


        try:
            # Add Event to database.
            db.session.add(new_event)
            db.session.commit()
        except exc.SQLAlchemyError as e:
            # In case of error, return error message
            db.session.rollback()
            return jsonify({"error": "Service Unavailable"}), 503

        # No error, return new event information
        return jsonify(new_event.serialize()), 201
```

Unit tests event_test.py:

To be consistent with the format of the input field of the html form, date should have a specific format, it is checked on the check_event_date function. Following tests are written for that function:

Correct Format: Given an event with the format that matches the documentation, it is expected for check_event_date(event) to return True.

```
def test_date_correct_format(self):
        """
            Given a string with correct format as YYYY-MM-DD, returns True
        """
        event = Event(date = "2021-12-30T12:30")

        is_date_correct = check_event_date(event)
```

```
        self.assertEqual(is_date_correct, True)
```

Invalid Month: Given an event with the correct format but with a invalid month such as 13, it is expected for check_event_date(event) to return False.

```
def test_date_invalid_month(self):
        """
            Given a string with a non existing month such as 13 returns
False
        """
        event = Event(date="2021-13-30T12:30")

        is_date_correct = check_event_date(event)

        self.assertEqual(is_date_correct, False)
```

Invalid Day: Given an event with the correct format but with a invalid day such as 45, it is expected for check_event_date(event) to return False.

```
def test_date_invalid_day(self):
        """
            Given a string with a non existing day such as 45 returns False
        """
        event = Event(date="2021-10-45T12:30")

        is_date_correct = check_event_date(event)

        self.assertEqual(is_date_correct, False)
```

Invalid Hour: Given an event with the correct format but with a invalid hour such as 25, it is expected for check_event_date(event) to return False.

```
def test_date_invalid_hour(self):
        """
            Given a string with a non existing hour such as 25 returns False
        """
        event = Event(date = "2021-11-30T25:30")

        is_date_correct = check_event_date(event)

        self.assertEqual(is_date_correct, False)
```

Invalid Minute: Given an event with the correct format but with a invalid minute such as 61, it is expected for check_event_date(event) to return False.

```
def test_date_invalid_minute(self):
        """
            Given a string with a non existing minute such as 61 returns
False
        """
        event = Event(date = "2021-11-30T23:61")

        is_date_correct = check_event_date(event)

        self.assertEqual(is_date_correct, False)
```

Invalid Format: Given an event with the incorrect format than the one specified, it is expected for check_event_date(event) to return False.

```
def test_date_invalid_format(self):
        """
            Given a string with a different format returns False
        """
        event = Event(date = "2021-11-30")

        is_date_correct = check_event_date(event)

        self.assertEqual(is_date_correct, False)
```

To make sure that the sport id parameter is correct, check_event_sport function is written. Following tests are written for that function:

Correct Id: Given an event with the id that is between 102-120, it is expected for check_event_sport(event) to return True.

```
def test_sport_correct(self):
        """
            Given sport id between 102-120 returns True
        """
        event = Event(sport = "104")

        is_sport_correct = check_event_sport(event)

        self.assertEqual(is_sport_correct, True)
```

Out Of Range: Given an event with the id that is not between 102-120, it is expected for check_event_sport(event) to return False.

```
def test_sport_out_of_range(self):
        """
```

```
            Given sport id not between 102-120 returns False
        """
        event = Event(sport = "5")

        is_sport_correct = check_event_sport(event)

        self.assertEqual(is_sport_correct, False)
```

Invalid Format: Given an event with the id that can not be changed to integer, it is expected for check_event_sport(event) to return False.

```
def test_sport_incorrect_type(self):
        """
            Given sport id as a string that can be changed to a integer
returns False
        """
        event = Event(sport = "five")

        is_sport_correct = check_event_sport(event)


        self.assertEqual(is_sport_correct, False)
```

Integer: Given an event with the id that is an integer and in range, it is expected for check_event_sport(event) to return True.

```
def test_sport_integer(self):
        """
            Given sport id not between 102-120 returns False
        """
        event = Event(sport = 103)

        is_sport_correct = check_event_sport(event)

        self.assertEqual(is_sport_correct, True)
```

Frontend Part with Screenshots:

Get_sport_names is a helper function for the front-end part of creating an event. It gets all the sport id-name pairs from the database and returns a dictionary with id as key and sport name as value. Used to create a dropdown select for users to select the correct sport.

```
"""
    Get sport id-name pairs.
    return:
```

```
        Dictionary with id key, sport name value
"""

def get_sport_names():
    result = Sport.query.all()

    sports={}

    for sport in result:
        sports[sport.id] = sport.sport
    return sports
```

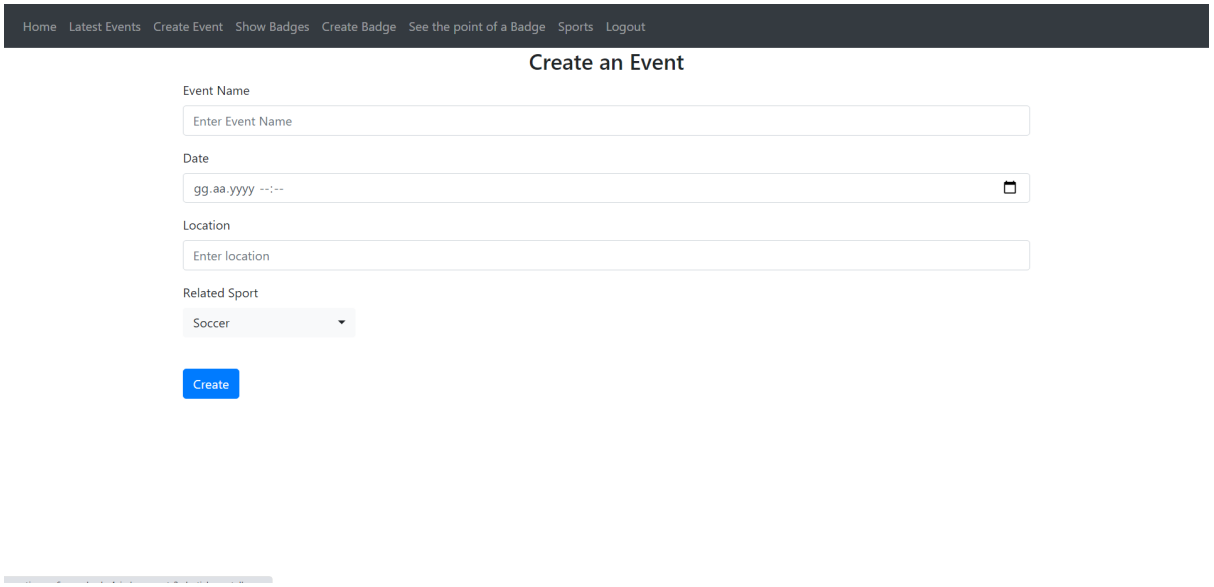
This is the main handler for the url create_event. Login is required to reach this page. User is shown a form which includes fields for the name, date, location and port fields. Current_user id is taken with the help of flask-login library. Then a POST request is made the the /api/v1.0/events/ endpoint to create the event and add to the database. Depending on the response, if an error occurred, a flash with an error statement is shown or created event information with a link to reach the event page is shown.

```
"""
    Front-end to create event.

"""

@views.route('create_event/', methods=['POST','GET'])
@login_required
def create_event():
    # Get id - sport name pairs
    sports = get_sport_names()
    if request.method == 'POST':
        # Get form information
        event = {
            "name" : request.form.get('name'),
            "date" : request.form.get('date'),
            "location" : request.form.get('location'),
            "sport" : request.form.get('sport'),
            "creator_user" : current_user.id
        }

        # Make request to back-end API
        req = BASE_URL+"/api/v1.0/events"
        headers = {'Content-type': 'application/json'}
        response = requests.post(req, data=json.dumps(event),
headers=headers)
        event = response.json()

        # No error, show event information
```

```
        if response.status_code == 201:
            flash('Event Created', category='success')
            return render_template("create_event.html", user= current_user,
sports=sports, created = True, event = event)
        # Incorrect information
        elif response.status_code == 400 :
            flash('Check Information Entered', category='error')
        # Some other error
        else:
            flash('Error Occurred, Try Again Later', category='error')

    return render_template("create_event.html", user= current_user,
sports=sports, created = False, event = {})
```

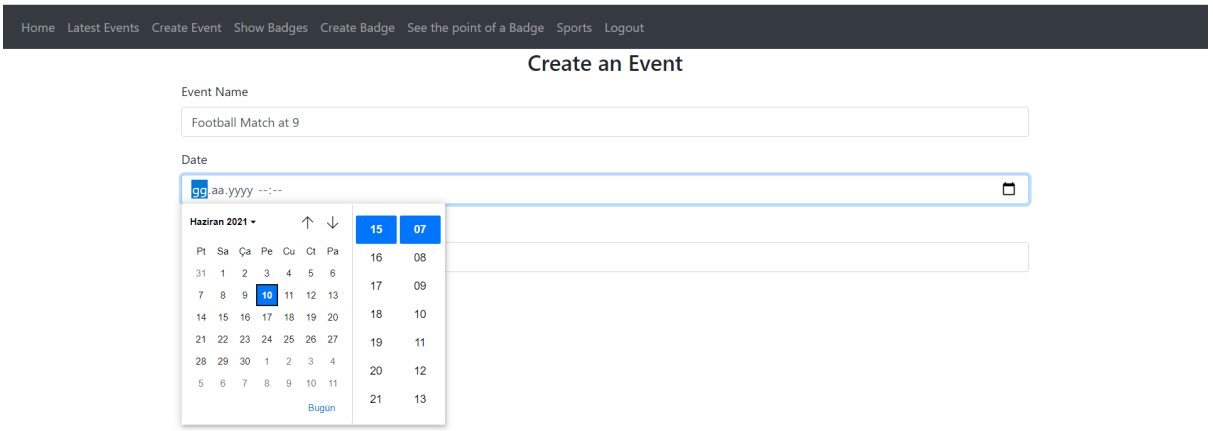
Screenshots:

First Screen on /create_event/

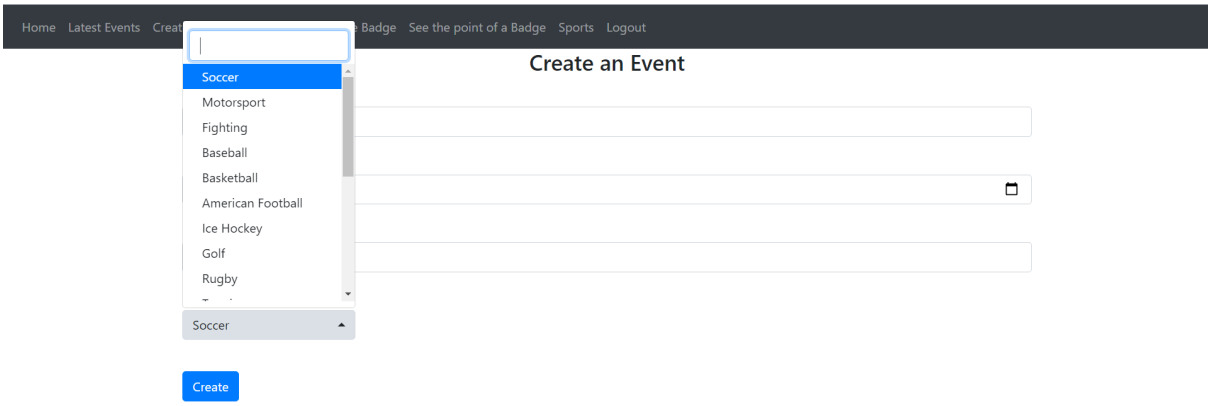


Date selection

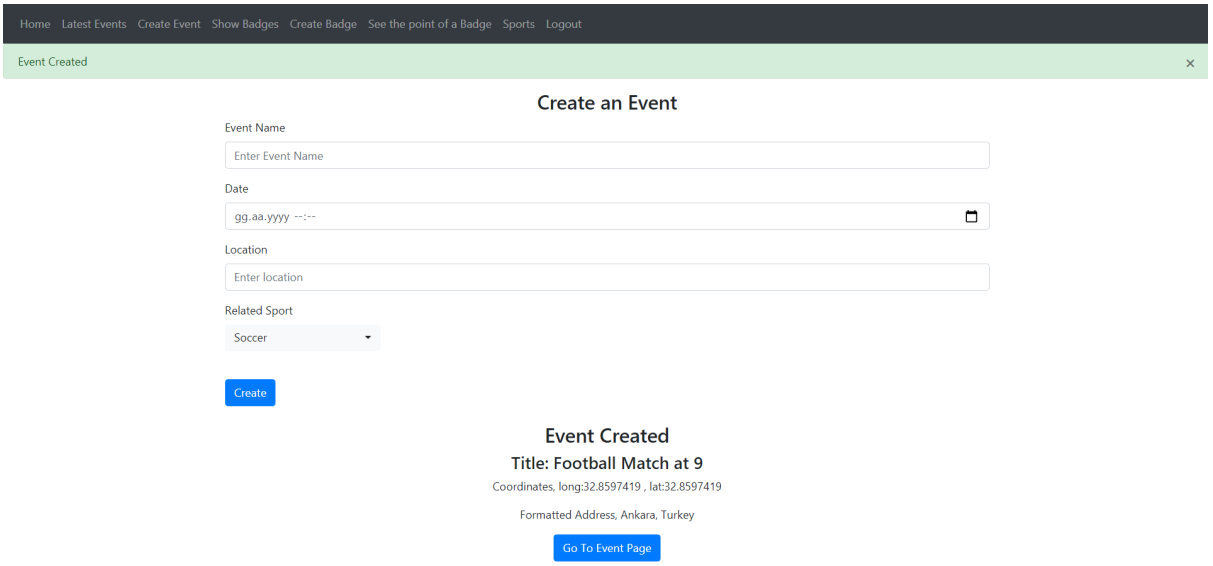
Note: Some browsers do not show the date selection field, using chrome is recommended. If date selection does not show date should be entered with the format "YYYY-MM-DDTHH:MM"



Sport selection



After Submit:
With the Go To Event Page button, user can be redirected to the page of the corresponding event.



2.2. Getting All Events From The System

Endpoint: /events
Method: Get
Description: Returns all events in the database. Supports some basic filterings. All parameters are optional and query parameters.
Authorization: Not Needed
Author: Ekrem Yusuf Ekmekci

Parameters:

"name": String that is searched in the event title
"sport": Sport id that represents sport type. Between 102-120
"date_from": Date string. Filters events before the given date. Format: YYYY-MM-DDTHH:HH
"date_to": Date string. Filters events after the given date. Format: YYYY-MM-DDTHH:HH

Responses:

200: "Events are fetched from database"
Returns an array of json objects.
[{
"Creator_name": (string) Full name of the creator. Random value for every call.
"creator_user": (integer) User id of the creator.
"date": (string) Date of the event. Format is YYYY-MM-DDTH:HH
"Entered_address": (string) Address entered by the creator.
"formatted_address": (string) Formatted address returned by google api. Takes entered address as input.
"id" : (integer) Id of the event
"latitude" : (number) Latitude of the event. Based on the formatted address.

“longitude” : (number) Longitude of the event. Based on the formatted address.

“name” : (string) Event description entered by the creator.

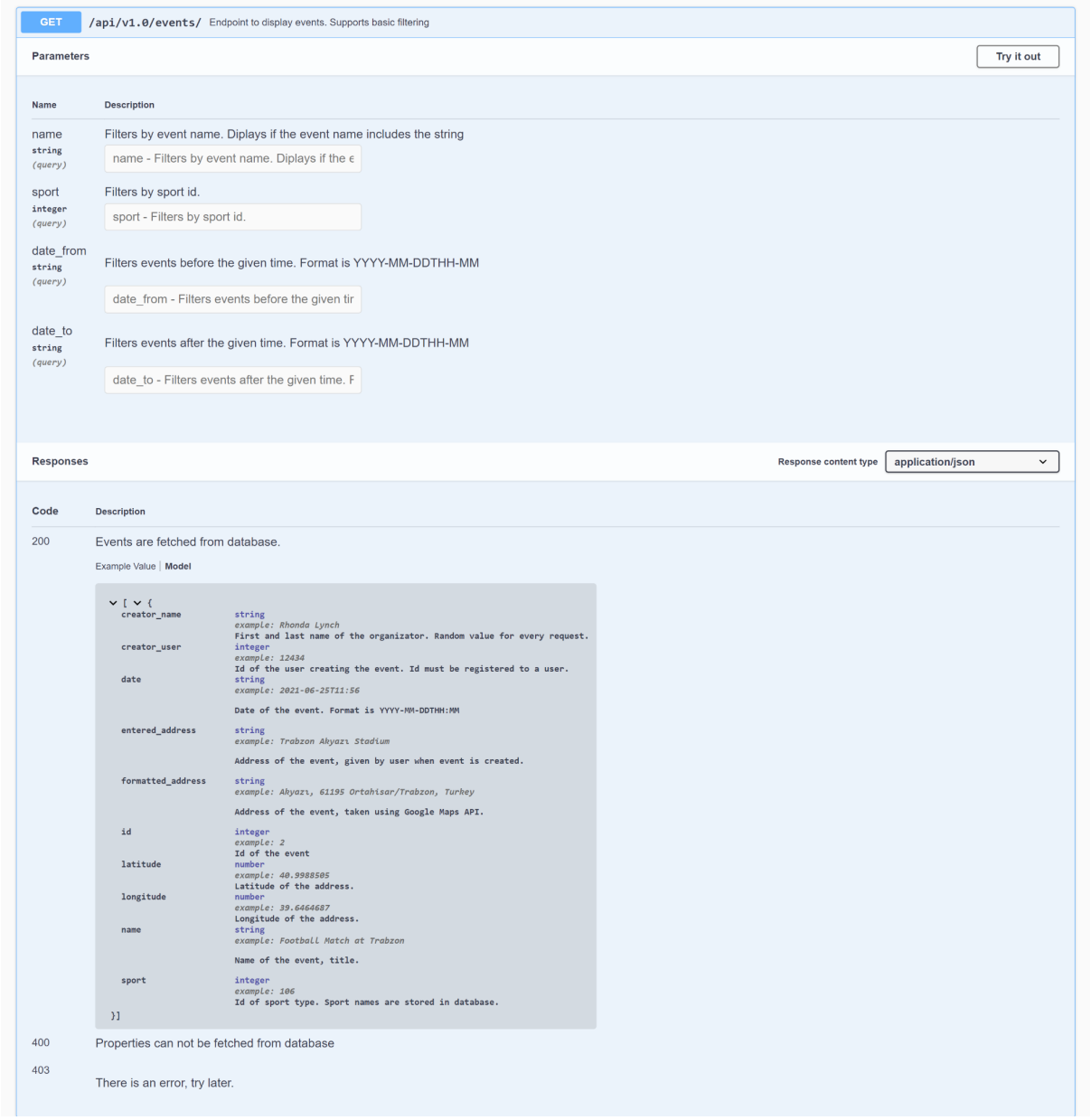
“sport” : (integer) Sport type represented as a number. Corresponds to a string in another table.

}]

400: Error. Single generic message
“Properties can not be fetched from database.”

503: Unexpected. Single generıc message
“There is an error. Try later.”

Swagger Documentation



GET /api/v1.0/events/ Endpoint to display events. Supports basic filtering

Parameters | Try it out

Name	Description
name string (query)	Filters by event name. Diplays if the event name includes the string name - Filters by event name. Diplays if the e
sport integer (query)	Filters by sport id. sport - Filters by sport id.
date_from string (query)	Filters events before the given time. Format is YYYY-MM-DDTHH-MM date_from - Filters events before the given tir
date_to string (query)	Filters events after the given time. Format is YYYY-MM-DDTHH-MM date_to - Filters events after the given time. F

Responses | Response content type: application/json

Code	Description
200	Events are fetched from database. Example Value | Model
400	Properties can not be fetched from database
403	There is an error, try later.

```
[ {
  creator_name        string
                      example: Rhonda Lynch
                      First and last name of the organizator. Random value for every request.
  creator_user        integer
                      example: 12434
                      Id of the user creating the event. Id must be registered to a user.
  date                string
                      example: 2021-06-25T11:56

                      Date of the event. Format is YYYY-MM-DDTHH:MM

  entered_address     string
                      example: Trabzon Akyazı Stadium

                      Address of the event, given by user when event is created.

  formatted_address   string
                      example: Akyazı, 61195 Ortahisar/Trabzon, Turkey

                      Address of the event, taken using Google Maps API.

  id                  integer
                      example: 2
                      Id of the event
  latitude            number
                      example: 40.9988505
                      Latitude of the address.
  longitude           number
                      example: 39.6464687
                      Longitude of the address.
  name                string
                      example: Football Match at Trabzon

                      Name of the event, title.

  sport               integer
                      example: 106
                      Id of sport type. Sport names are stored in database.
}]
```

Related Functions

This function takes query parameters of the api and converts them into a database query. Checks for every option and adds filters if any exists. Then function returns the final query to be used in the api.
Parameters: all optional parameters from the api call.
Returns: Final query of Event table.

```
#helper method for querying
def query_handler_events(name, sport, date_from, date_to):
    query = 'SELECT * FROM Event WHERE'
    filters = []
    # adds filter for name
    if name:
        filters.append(' name LIKE \'%' + name + '%\' AND')
    # adds filter for sport name
    if sport:
        filters.append(' sport = ' + str(sport) + ' AND')
    # adds filter for initial date
    if date_from:
        filters.append(' date >= \'' + date_from + '\' AND')
    # adds filter for final date
    if date_to:
        filters.append(' date <= \'' + date_to + '\' AND')

    # checks if there is any filter added
    if filters:
        for filt in filters:
            query += filt
        query = query[:-4] + ';'
    else:
        query = query[:-6] + ';'
    return query
```

Endpoint Code

Summary: First fetches parameters, converts them into database queries, then executes the query. Takes the result, converts it into a json object array, and adds additional fields with the help of an external api. Then returns the result. Input and output format are specified on the header.

External API takes the number of events as parameter and returns that number of random names in json array format. “Inc” parameter specifies that the returned object includes only names of the random users.

```
@events.route('/', methods = ['GET','POST'])
@swag_from('doc/events_GET.yml', methods=['GET'])
def event():
```

```
    # handles get request
    if request.method == 'GET':
        # fethes body parameters
        query_parameters = request.args
        name = query_parameters.get('name')
        sport = query_parameters.get('sport')
        date_from = query_parameters.get('date_from')
        date_to = query_parameters.get('date_to')

        # sets up queries
        query = query_handler_events(name, sport, date_from, date_to)

        # executes query and converts to json format
        eventList = db.engine.execute(query)
        eventList = json.dumps([dict(event_item) for event_item in
eventList])

        # temporary parameter for adding additional info into result
        temp = json.loads(eventList)

        # external api for assigning random names for event creators
        multi = requests.get('https://randomuser.me/api/?inc=name&results='
+ str(len(temp))).json()['results']

        # adds external info into the result
        for index, item in enumerate(temp):
            name = multi[index]['name']
            item['creator_name'] = name['first'] + ' ' + name['last']

        # returns the result in json format
        return jsonify(temp)
```

Unit Tests

Helper function is tested with unit tests. Its format and inputs are specified above. Unit tests checks for the type and content. Returned output is expected to be consistent with the ideal output.

Case that no valid parameter is passed tested:

```
# test for no filter
def test_no_filter(self):
    query_expected = "SELECT * FROM Event;"
    query_method = query_handler_events(None, None, None, None)
    self.assertEqual(query_expected, query_method)
```

Case that only name argument is not null:

```
# test for filter by name
def test_filter_by_name(self):
    query_expected = "SELECT * FROM Event WHERE name LIKE '%street%';"
    query_method = query_handler_events("street", None, None, None)
    self.assertEqual(query_expected, query_method)
```

Case that only sport argument is not null:

```
# test for filter by sport
def test_filter_by_sport(self):
    query_expected = "SELECT * FROM Event WHERE sport = 106;"
    query_method = query_handler_events(None, 106, None, None)
    self.assertEqual(query_expected, query_method)
```

Case that only date_from argument is not null:

```
# test for filter by date_from
def test_filter_by_date_from(self):
    query_expected = "SELECT * FROM Event WHERE date >= '2021-06-08T18:00';"
    query_method = query_handler_events(None, None, '2021-06-08T18:00',
None)
    self.assertEqual(query_expected, query_method)
```

Case that only date_to argument is not null:

```
# test for filter by date_to
def test_filter_by_date_to(self):
    query_expected = "SELECT * FROM Event WHERE date <= '2021-06-09T18:00';"
    query_method = query_handler_events(None, None, None,
'2021-06-09T18:00')
    self.assertEqual(query_expected, query_method)
```

```
# test for filter by sport and date_from
def test_filter_by_sport_and_date_from(self):
    query_expected = "SELECT * FROM Event WHERE sport = 106 AND date >=
'2021-06-08T18:00';"
    query_method = query_handler_events(None, 106, '2021-06-08T18:00', None)
    self.assertEqual(query_expected, query_method)
```

Case that only date_from and date_to arguments are passed:

```
# test for filter by date_from and date_to
def test_filter_by_date_from_to(self):
    query_expected = "SELECT * FROM Event WHERE date >= '2021-06-08T18:00'
AND date <= '2021-06-09T18:00';"
    query_method = query_handler_events(None, None, '2021-06-08T18:00',
'2021-06-09T18:00')
```

```
    self.assertEqual(query_expected, query_method)
```

Case that only name and sport arguments are passed:

```
# test for filter by name and sport
def test_filter_by_name_and_sport(self):
    query_expected = "SELECT * FROM Event WHERE name LIKE '%street%' AND
sport = 106;"
    query_method = query_handler_events("street", 106, None, None)
    self.assertEqual(query_expected, query_method)
```

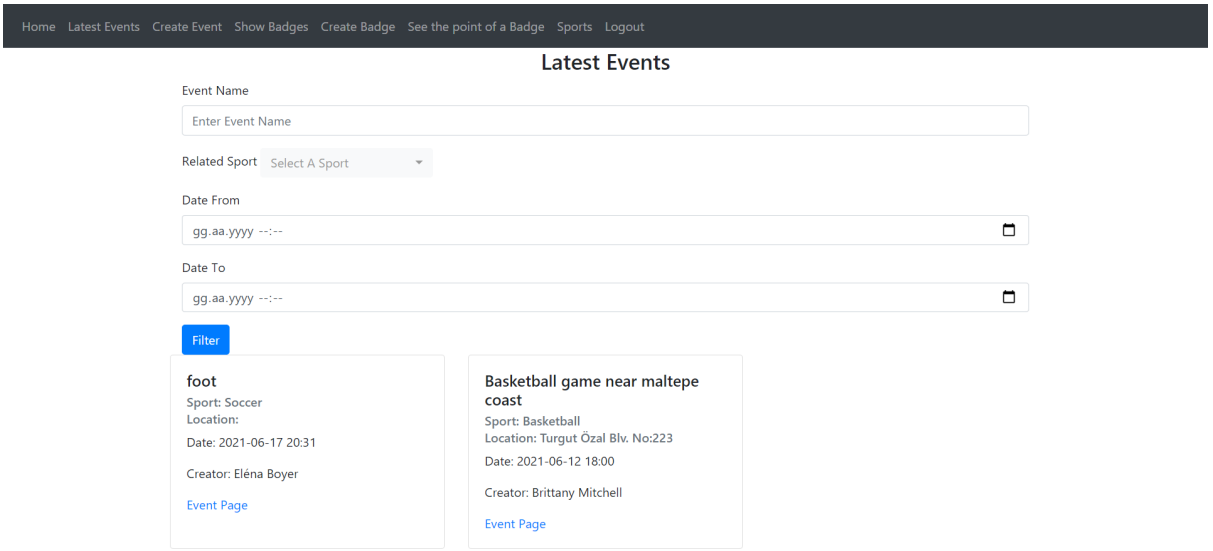
Server part that uses the specified API

In the server layer, page that displays events takes parameters from form and passes them to the api. Its actual function is kind of a mediary. It takes inputs, passes them, takes the result, renders it and sends to the frontend part.

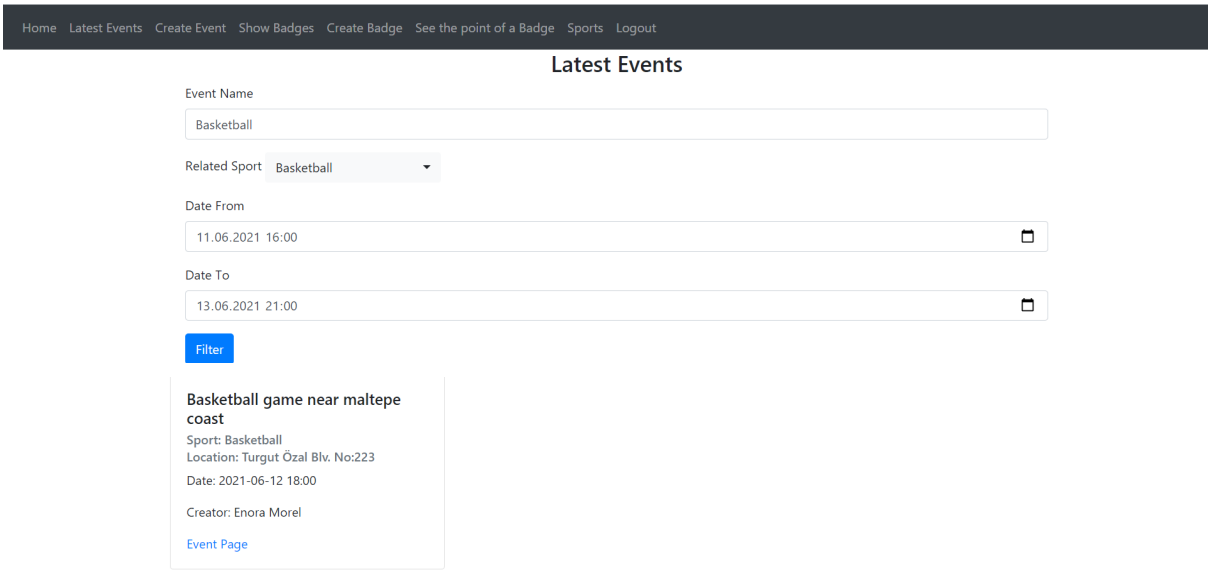
```
@views.route('events/', methods=['POST', 'GET'])
@login_required
def event_search():
    # gets sport names and ids
    sports = get_sport_names()

    # initialize request url
    req = BASE_URL+"/api/v1.0/events"

    # handle post request from frontend. in form input format
    if request.method == 'POST':

        print(request.form.get('date_from'))
        # change url according to posted parameters
        req = url_handler_events(req, request.form.get('name'),
request.form.get('sport'), request.form.get('date_from'),
request.form.get('date_to'))
        print(req)

    # call api from event.py file
    headers = {'Content-type': 'application/json'}
    response = requests.get(req, headers = headers)

    # return rendered page
    if response.status_code == 200:
        #flash('Events are fetched successfully', category='success')
        # TODO: When event page implemented, redirect to it.
        return render_template("event_search.html", user= current_user,
sports=sports, events=response.json())
    elif response.status_code == 400 :
```

```
        return "<h1>Check Information Entered</h1>"
    else:
        return "<h1>Error Occured, Try Again Later</h1>"
```

Sample frontend product:



Note: all parameters are consistent with the api parameters. Related sport string has integer value in form.



2.3. Posting a New Interested (for an Event) Into System

Endpoint: /events/<event_id>/interested
Method: Post
Description: API for declaring interest into an event. Very basic and simple. Built for educational purposes. Path parameter needs to be an integer. Body parameter is described below. Both parameters are required.

Authorization: Not Needed
Author: Ekrem Yusuf Ekmekci
Parameters:

Path Parameter:

<event_id>: (integer) Id of the event that the user is interested in.

Body:

{ “user_id” : (integer) Id of the user that declares their interest }

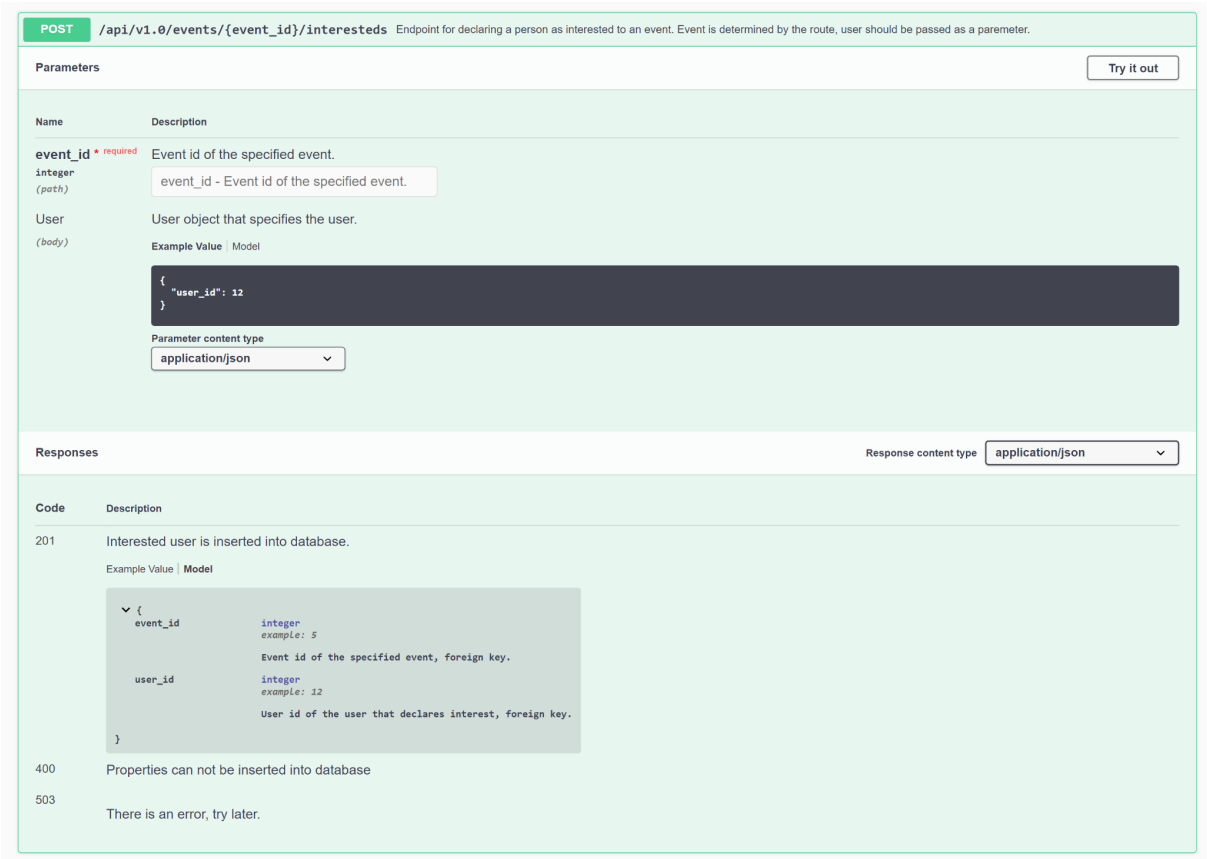
Responses:

201: “Interested user is inserted into database”
Returns the pair that was inserted into the database.
{
“event_id” : (integer) Id of the event that the user is interested in.
“user_id” : (integer) Id of the user that declares their interest.
}

400: Error. Single generic message
“Properties can not be inserted into database.”

503: Unexpected. Single generic message
“There is an error. Try later.”

Swagger Documentation



Endpoint Code

Below code fetches parameters from path and body, creates db object from them, and tries to insert the object into the database. If it could insert, returns the inserted pair.

```
# endpoint for declaring a person as interested to an event
# event is determined by the route, user should be passed as a paremeter
@interesteds.route('/<event_id>/interesteds', methods = ['POST'])
@swag_from('doc/interesteds_post.yml', methods = ['POST'])
def declare_interest(event_id):
    if request.method == 'POST':
        # basic checks
        if not request.json:
            return {'error': 'Request is not in JSON format'}, 400
        if not 'user_id' in request.json:
            return {'error': "No user id provided"}, 400
        interested = Interesteds(
            event_id = event_id,
            user_id = request.json['user_id'] #current_user
        )

    # insert into database
    try:
        db.session.add(interested)
        db.session.commit()
    except exc.NoReferenceError as e:
        db.session.rollback()
        return "Bad request", 400
    except exc.SQLAlchemyError as e:
        db.session.rollback()
        return "Server problem", 503

    return interested.serialize(), 201
```

Because the API is built for educational purposes only, it has a simple structure and it has no additional algorithm in it. It only takes the parameters and tries to insert them into the database directly. So, there is no part to be tested.

Note: This is the end of this API's documentation. It proves that the author can implement RESTful APIs that handle get and post methods.

2.4. Getting an Event by its Id

Endpoint: /events/<event_id>
Method: GET
Authorization: Not Needed

Author: Deniz Arda Budak
Parameters: Path parameter:
event_id (required)

Expected Response When No Error:

{
"event_id": Id of the user creating the event, obtained from the path
"id": oId of the user creating the event, obtained from database
"name": Latitude of the location.
"date": Date of the event.
"entered_address": Address of the event, given using basic English. Same as location on parameters.
"Formatted_address": Address of the event, taken using Google Maps API
"latitude": Latitude of the address
"longitude": Id of the event.
"creator_user": ID of the user creating the event.
"sport": Longitude of the location.
"hour": The hour at which the event will take place.
"weather": The description of the weather, taken using OpenWeatherMap API.
"weather_icon": The weather icon id, taken using OpenWeatherMap API.
}

Responses:

200: "Event with the corresponding ID is fetched."

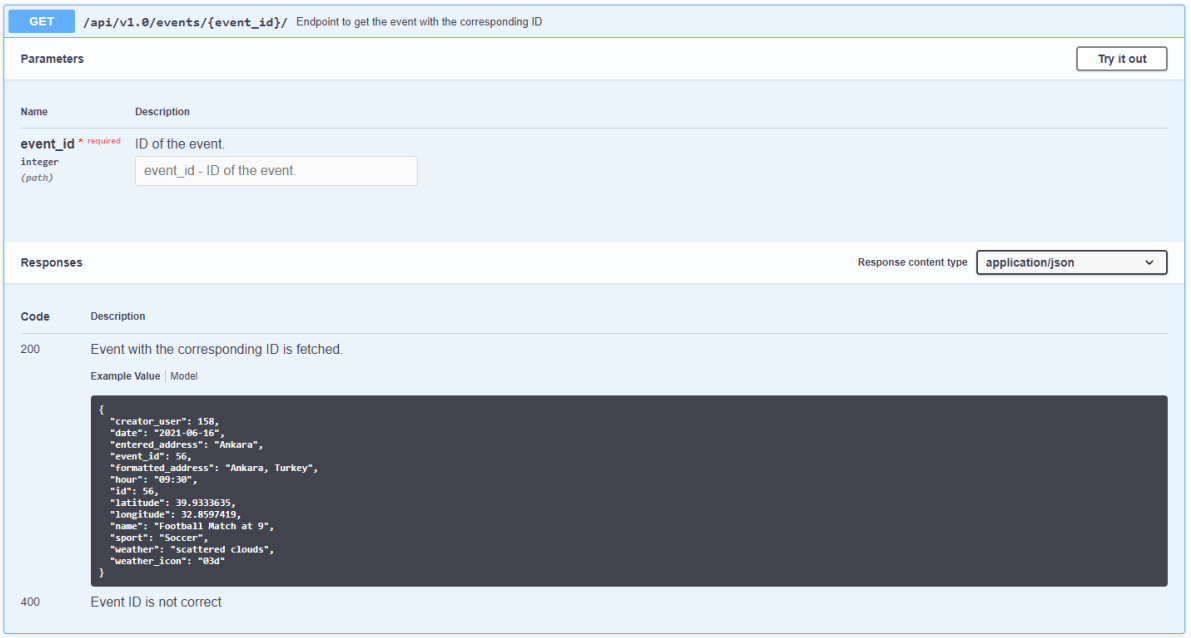
Example Response:

```
{
"event_id": 56
"id": 56
"name": "Football Match at 9"
"date": "2021-06-16"
"entered_address": "Ankara"
"formatted_address": "Ankara, Turkey"
"latitude": 39.9333365
"Longitude": 32.885654
"creator_user": 160
"sport": "Soccer"
"hour": "19:00"
"weather": "scattered clouds"
"weather_icon": "03d"
}
```

400: "Event ID is not correct."

```
{
"error": "Event ID is not correct"
}
```

Swagger Documentation:



Related Functions:

get_weather() function takes two parameters, latitude and longitude of the event. A get request is sent to the uri(OpenWeather API). The response is converted to json and the description of the weather and the weather icon id are returned.

```
def get_weather(latitude, longitude):
    """
    Get weather description using OpenWeather API.

    parameters:
        latitude : "latitude of the location"
        longitude : "longitude
    return:
        The weather description and the weather icon id
    """

    # GET request to the uri using API_KEY2
    parameters = {'lat': latitude, 'lon' : longitude, 'appid': API_KEY2 }
    uri = 'https://api.openweathermap.org/data/2.5/weather'
    r = requests.get(uri, params=parameters)
    r = r.json()
    # return the weather description and the weather icon id
    return r["weather"][0]["description"], r["weather"][0]["icon"]
```

check_event_id(new_event) function takes an event as a parameter. Then the event is serialized, to get its id. If id is equal or smaller than 0, False is returned, otherwise True.

```python
def check_event_id(new_event):
    """
    Check if the given event id is valid
    parameters:
        new_event: Event
    return:
        True if valid false otherwise
    """
    event = new_event.serialize()
    id = event["id"]
    if(id <= 0):
        return False
    return True
```

check_weather_icon(weather_icon_id) takes an weather icon id as a parameter, which is obtained from OpenWeather API. If the weather_icon_id does not match with the weather-icon_id_regex, False is returned. If it matches, True is returned.

```python
def check_weather_icon(weather_icon_id):
    """
    Check the validity of a given weather icon id
    parameters:
        weather_icon_id: String
    return:
        True if valid false otherwise
    """
    # create the corresponding regex
    weather_icon_id_regex = "[0-9][0-9][dn]"
    if not re.match(weather_icon_id_regex, weather_icon_id):
        return False
    return True
```

Endpoint Code:

This function is the handler for the GET request sent to the /api/v1.0/events/<event_id>. First, the method of the request is checked if it is GET or not. If it is a GET request, the event is fetched from the database, using the query.get function of SQLAlchemy. If the event is None, which means that there is no such event in the database, an error is returned with the 400 response status code. After that, if the id of the event is invalid(equal or smaller than 0), an error is returned with the 400 response status code. If there is no error, the event is serialized, to be able add the weather data to it. The event_id key is added, which is the

event_id in the path. The “hour” and “date” keys are added, storing the hour and date. The weather information is obtained by calling the helper function get_weather(). The weather description is added to the event with the key “weather”. The weather icon is checked if it is valid. If it is valid, it is added to the event with the key “weather_icon”. Then “sport” of the event is modified so that it contains not the sport id, but the sport name. Finally, event_with_weather is returned in json format, with the help of jsonify() function, with the response status code 200.

```
@events.route('/<event_id>/', methods = ['GET'])
@swag_from('doc/event_GET.yml', methods=['GET'])
def get_event_by_id(event_id):
    """
            Used to get the event with the corresponding ID
            event_id is a path parameter and it is required
            Endpoint description:
                ./api/v1.0/events/<event_id>
                'GET':
                    Response Example : {
                                            "event_id": 24,
                                            "id": 24,
                                            "name": "Basketball Match at 9",
                                            "date": "2021-06-16",
                                            "formatted_address": "Artvin,
Merkez/Artvin, Turkey",
                                            "entered_address": "Artvin",
                                            "longitude": 41.87645,
                                            "latitude": 43.54367,
                                            "creator_user" : 5748
                                            "sport": "Basketball"
                                            "hour" : "09:00"
                                            "weather" : "scattered clouds"
                                            "weatcher_icon" : "03d"
                                        }
                    Status Codes:
                        200: "Event has been fetched."
                        400: "Event ID is not correct."

        """

    if request.method == 'GET':
        event = Event.query.get(event_id)

        # if there exists no such event return error.
        if(event is None):
            return jsonify({"error":"Event ID is not correct"}), 400
        # if event id is not valid return error
        elif(not check_event_id(event)):
            return jsonify({"error":"Event ID is not correct"}), 400
```

```
        else:
            event_with_weather = event.serialize()
            event_with_weather['event_id'] = event_id
            # Split the hour and the date.
            event_with_weather["hour"] =  event_with_weather["date"][11:]
            event_with_weather["date"] = event_with_weather["date"][:10]
            #get the weather and the weather_icon, sending request to
OpenWeather API
            weather, weather_icon =
get_weather(event_with_weather["latitude"], event_with_weather["longitude"])
            event_with_weather["weather"] = weather
            #check if weather_icon is valid.
            if(check_weather_icon(weather_icon)):
                event_with_weather["weather_icon"] = weather_icon
            sport_names = get_sport_names()
            # Change the sport id to the corresponding sport name
            event_with_weather["sport"] =
sport_names[event_with_weather["sport"]]
            return jsonify(event_with_weather), 200
```

Unit tests, event_get_by_id_test.py

This function checks the validness of the id of a given Event. If the id is larger than 0, it means that the id of the event is valid

```
def test_valid_event_id(self):
        """
            Check if the event id is valid, i.e id is larger than 0
        """
        event = Event(id = 51)
        is_id_valid = check_event_id(event)
        self.assertEqual(is_id_valid , True)
```

This function checks the invalidness of the id of a given Event. If the id is smaller than or equal to 0, it means that the id of the event is invalid

```
def test_invalid_event_id(self):
        """
            Check if the event id is invalid, i.e id is smaller than 1
        """
```

```
        event = Event(id = -15)
        is_id_valid = check_event_id(event)
        self.assertEqual(is_id_valid , False)
```

This function checks the validness of the weather_icon of a given event. If the weather icon id is in the form of “\d\d[dn]” regex, it means that the weather icon id is valid.

```
def test_valid_weather_icon_id(self):
        """
            Check if the weather icon id is valid
        """
        weather_icon_id = "03d"
        is_weather_icon_valid = check_weather_icon(weather_icon_id)
        self.assertEqual(is_weather_icon_valid , True)
```

This function checks the invalidness of the weather_icon of a given event. If the weather icon id is not in the form of “\d\d[dn]” regex, it means that the weather icon id is invalid.

```
    def test_invalid_weather_icon_id(self):
        """
            Check if the weather icon id is invalid
        """
        weather_icon_id = "07x"
        is_weather_icon_valid = check_weather_icon(weather_icon_id)
        self.assertEqual(is_weather_icon_valid , False)
```

Frontend Part with Screenshots

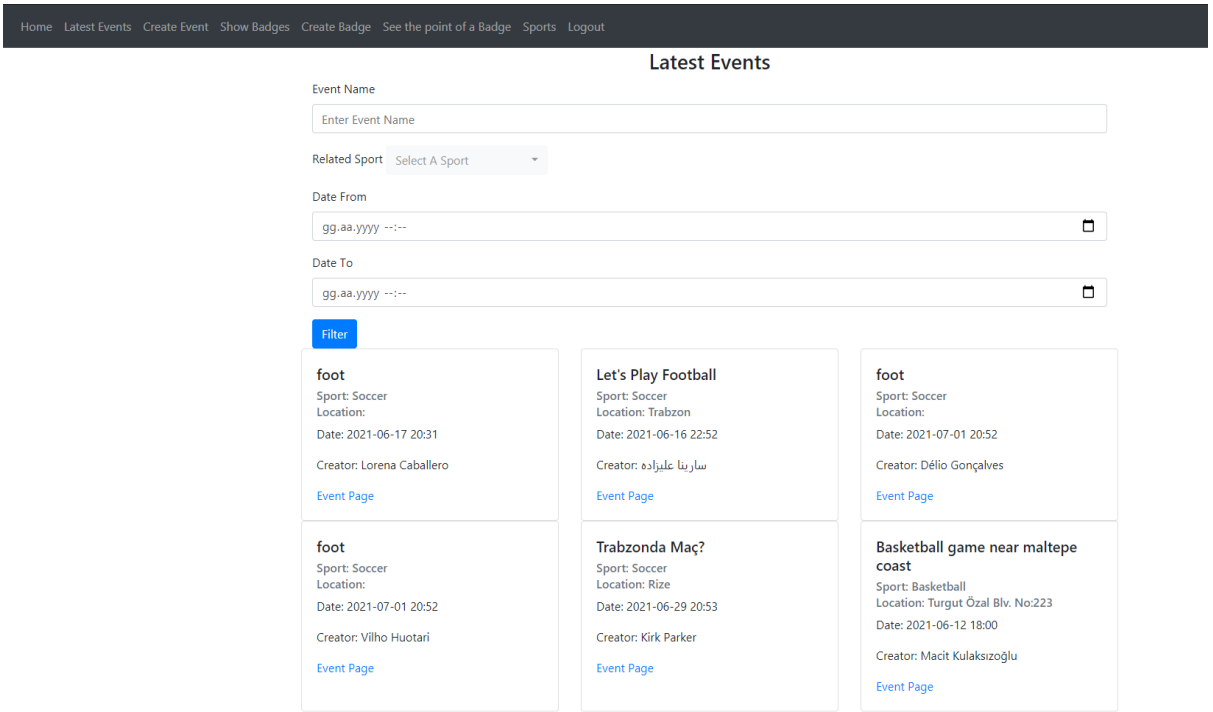
This is the main handler for the url events/<event_id>. This url shows the information about an event, given its id in the path. The information that are showed are the name of the event, date of the event, hour of the event, sport type of the event, the weather description of the event, an icon representing the weather description and a hyperlink to the Discussion Page of the event.
In the function, firstly it is checked it is a GET request. Then, a GET request is sent to the API endpoint which gives the event, given its id. If the status code of the response of the API endpoint is 400, an error is returned on the page, stating that the ID is invalid. If the status code of the response of the API endpoint is 404, an error is returned on the page, stating that the ID is invalid. Else, the necessary information about the event are showed on the page.

```
@views.route('events/<event_id>/', methods=['GET'])
def view_event(event_id):
    """
```

```
    This is the front-end side of the show event by id functionality.
    Shows the event information for the event with event_id and also
    the corresponding weather icon, according to the weather description in
    the event location.
    """
    if request.method == 'GET':

        uri = f"{BASE_URL}/api/v1.0/events/{event_id}/"
        res = requests.get(uri)
        event = res.json()
        # Incorrect information
        if res.status_code == 400:
            return "<h1>ERROR : The ID is invalid</h1>"
        # There exists no such event with the corresponding ID
        elif res.status_code == 404 :
            return "<h1>ERROR : There is no such event</h1>"
        else:
            icon = event["weather_icon"]
            weather_icon_url =
f"http://openweathermap.org/img/wn/{icon}@2x.png"
            url = f"/events/{event_id}/discussions"
            return render_template("event_page_by_id.html", user =
current_user, event = event, weather_icon_url = weather_icon_url, url=url)
```

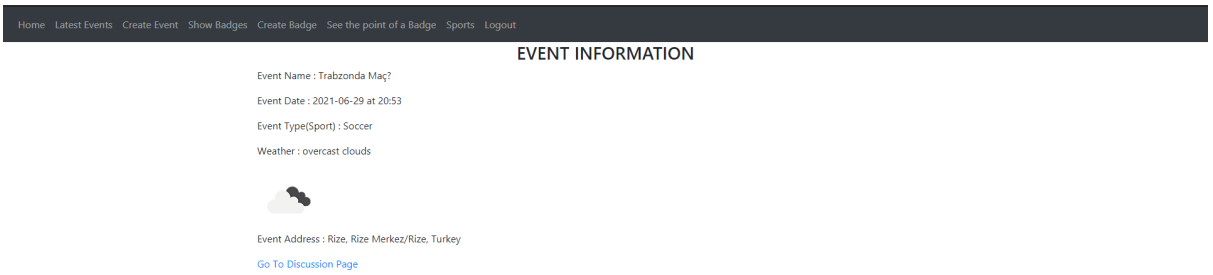
Screenshots:

The events can be seen in the Latest Events.

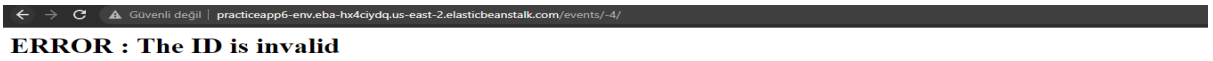


When the hyperlink “Event Page”, which is blue, is clicked by the user, the user is redirected to the event page of the event. Below is the event information page, which was implemented

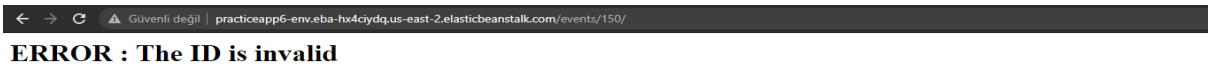
by me, as I have explained above. The event name, event date, event type, weather description of the event, the icon representing the weather, teh event address and a hyperlink to the discussion page of the event are available.



When the <event_id> variable in the path is an invalid id,is(the event_id of the event that is wanted to be shown is -4 in this case), an error is printed on the page.



When the <event_id> variable in the path is the id of an event that does not exist(the event_id of the event that is wanted to be shown is 150 in this case), an error is printed on the page.



2.5. Adding a new Spectator to an Event

Endpoint: /events/<event_id>/interested

Method: Post
Description: API for adding a new Spectator to an event. The path parameter is required to be an integer. Body parameter is described below. Both parameters are required.
Authorization: Not Needed
Author: Deniz Arda Budak
Parameters:

Path Parameter:

<event_id>: (integer) Id of the event that the user that wants to be a spectator of the event.

Body:

{ “user_id” : (integer) Id of the user that wants to be a spectator of the event }

Responses:

201: “`Spectator user is inserted into database.`”

Returns the pair that was inserted into the database.

{

“event_id” : (integer) Id of the event that the user is interested in.

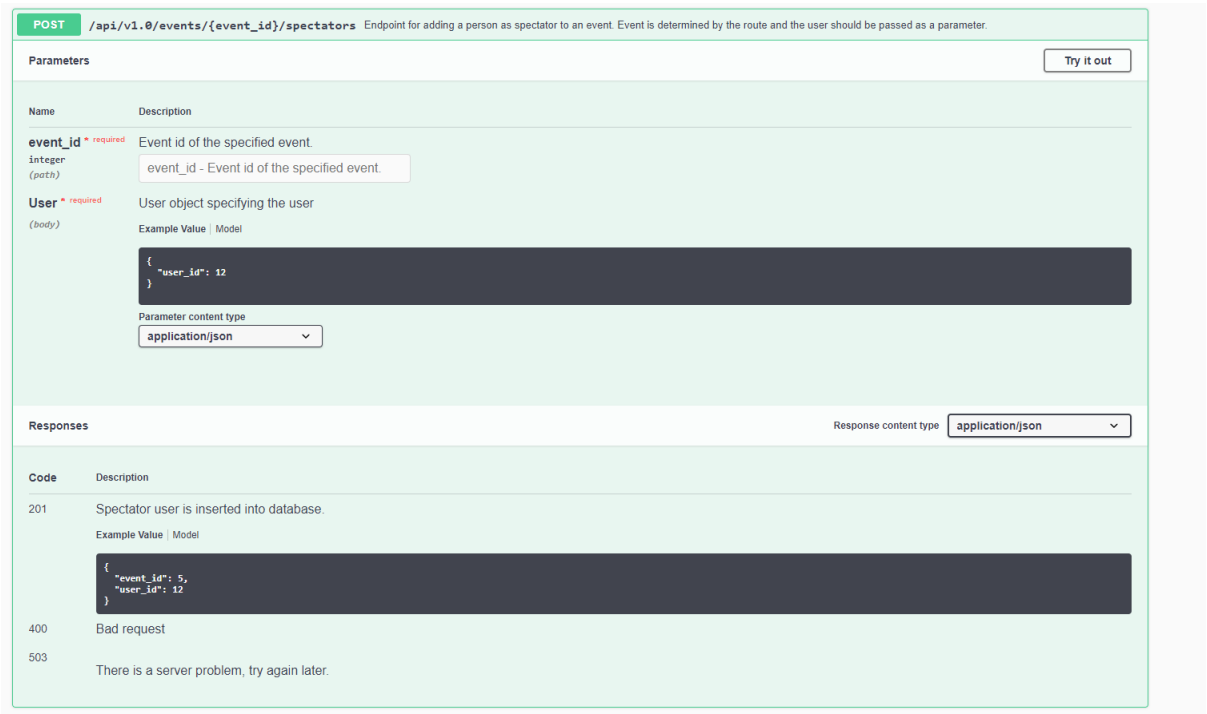
“user_id” : (integer) Id of the user that declares their interest.

}

400: `"Bad request"`

503: `"There is a server problem, try again later."`

Swagger Documentation



Endpoint Code:

This function is the handler for the POST request sent to the /api/v1.0/events/<event_id>/spectators, which adds a user as a Spectator to an event. First,

it is checked if the body is in JSON format. If it is not, an error with 400 status code is returned. Then it is checked if the key “user_id” is the body. If it is not, an error with 400 status code is returned. If there is no error, a new spectator is created to be inserted into the database. If encountered with the exc.noReferenceError exception, an error with 400 status code is returned. If encountered with the exc.SQLAlchemy exception, an error with 503 status code is returned. If the insertion into the database is successful, the spectator is returned in JSON format with the response status code 201.

```
@spectators.route('/<event_id>/spectators', methods = ['POST'])
@swag_from('doc/spectators_post.yml', methods = ['POST'])
def add_as_spectator_to_event(event_id):

    """
    Endpoint for adding a user as a spectator to an event.
    event_id is a path parameter and the user_id should be in the request
body.
    Endpoint description:
        ./api/v1.0/events/<event_id>/spectators
        'GET':
            JSON Request Body Format :      {
                                                "user_id" = id of the user
who will be added as spectator to the event, required.
                                            }
            Response Example : {
                                    "event_id": 45,
                                    "user_id": 1764
                                }
            Status Codes:
                201: "The user has been added to the event as spectator"
                400: "Bad request"
                503: "Server problem"
    """
    if request.method == 'POST':
        # check if the request body is in json format
        if not request.json:
            return jsonify({'error': 'request body is not in JSON format'}),
400
        if not 'user_id' in request.json:
            return {'error': "user id is not given"}, 400
        # create the spectator which will be inserted into the database
        spectator = Spectators(
            event_id = event_id,
            user_id = request.json['user_id']
        )

        # insert into the database
        try:
            db.session.add(spectator)
            db.session.commit()
```

```
        except exc.NoReferenceError as e:
            db.session.rollback()
            return "Bad request", 400
        except exc.SQLAlchemyError as e:
            db.session.rollback()
            return "Server problem", 503

        return jsonify(spectator.serialize()), 201
```

3. Sport Related Endpoints

3.1. Basic Filtering of Sports

Endpoint: /sports?keyword=<keyword>
Method: GET
Authorization: Not Needed
Author: Elif Sema Balcıoğlu
Expected Response When No Error:

```
{
    "sports": List of sports which contain the keyword in their names.
    [
        {
            "id": Id of the sport.
            "sport": Name of the sport.
        }
    ]
}
```

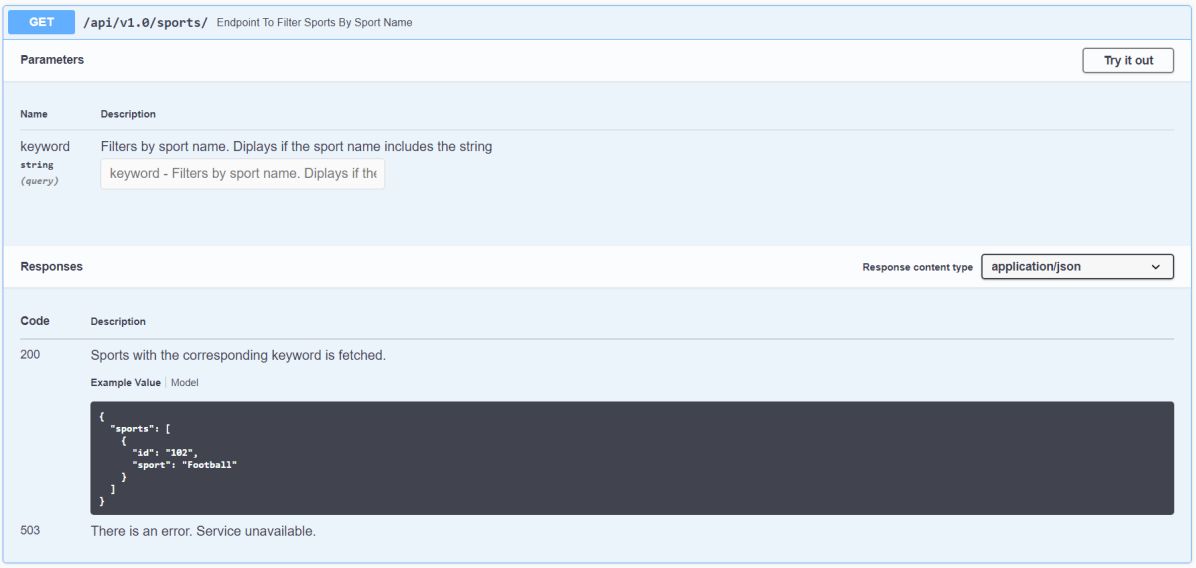
Responses:
200: `"Result returned with no error"`
Example Response for <keyword> = sp:

```
{
    "sports": [
        {
            "id": 103,
            "sport": "Motorsport"
        }
    ]
}
```

503: `"Service unavailable"`

```
{
    "error": “Service unavailable”
}
```

Swagger Documentation:



Related functions:

get_sport_by_keyword(sports, keyword) function is used to filter sports given the keyword. Parameter sports is a list of Sport objects, keyword is the string that is searched. If any other type than string is given, its string representation is checked. Event sport on the sports list is traversed, for each of them if their sport field which contains the sport name is checked to include the keyword, if so it is added to the result list. Finally the result list is returned.

Please note that exact match between the keyword is needed. “Basketball” will not match the keyword “basket”.

```
"""
    Filters the given sports list by name using the keyword.
    parameters:
        sports: "list of Sports"
        keyword: "keyword to be included in the sport name"
    returns:
        a list of Sport objects which contain the keyword in its name.
"""
def get_sport_by_keyword(sports, keyword):
    result = []
    for sport in sports:
        if str(keyword) in sport.sport:
            result.append(sport)

    return result
```

Endpoint Code:

This is the handler for the GET request sent to the /api/v1.0/sports.
All sports are taken from the database. If no keyword was given, keyword variable made equal to empty string, if it was given on the query parameters, keyword variable is assigned to it. Using the get_sport_keyword(sports, keyword) function, sports are filtered with the keyword. Result is returned in the json format explained on [Basic Filtering of Sports](). If an error occurs in any one of these steps, a response with 503 code and “Service Unavailable” message is returned.

```
@sports.route('/', methods = ['GET'])
@swag_from('doc/sport_GET.yml', methods=['GET'])
def sport():
    try:
        # Get all sports.
        sports = Sport.query.all()

        # Get keyword parameter.
        keyword = request.args.get('keyword') if 'keyword' in request.args
else ""

        # Filter by keyword.
        result = get_sport_by_keyword(sports, keyword)

        # Return filtered result.
        return jsonify({"sports":[s.serialize() for s in result]}), 200
    except:
        # Some error occured.
        return jsonify({"error":"Service unavailable"}), 503
```

Unit tests sport_test.py:

Sports that include the keyword are selected on the get_sport_by_keyword function. Following tests are written for that function:

Following data is used for testing.

```
def setUp(self):
        # mock data
        self.data = [Sport(id = 102, sport = "Football-3"), Sport(id = 103,
sport = "Basketball"), Sport(id = 105, sport = "Tennis")]
```

Filter by “ball”: Using the keyword “ball”, get_sport_by_keyword function should return the entries with names “Football” and “Basketball”.

```
def test_sport_keyword(self):
        """
            Given keyword "ball", sports that contain that keyword
should be returned.
        """

        ball_keyword = get_sport_by_keyword(self.data, "ball")
        expected = [self.data[0], self.data[1]] # Only "Football" and
"Basketball"

        self.assertEqual(ball_keyword, expected)
```

Filter by “”: Using no keyword, get_sport_by_keyword function should return all entries.

```
def test_no_keyword(self):
        """
            Given no keyword, all sports should be returned.
        """
        no_keyword = get_sport_by_keyword(self.data, "")
        expected = self.data # No filtering
        self.assertEqual(no_keyword, expected)
```

Filter for no result: Using a keyword not included in any sport, get_sport_by_keyword function should return no entries.

```
def test_no_result(self):
        """
            Given a keyword not existing on sports, no sport should be
returned.
        """

        no_data = get_sport_by_keyword(self.data, "aa")
        expected = [] # No match

        self.assertEqual(no_data, expected)
```

Filter wrong type: Using a keyword which is not a string, get_sport_by_keyword function should return entries with its string representation.

```
def test_wrong_type(self):
        """
            Given a keyword which is not a string, sports should be filtered
with its string representation and produce no error.
        """

        no_data = get_sport_by_keyword(self.data, 3)
```

```
        expected = [self.data[0]]

        self.assertEqual(no_data, expected)
```

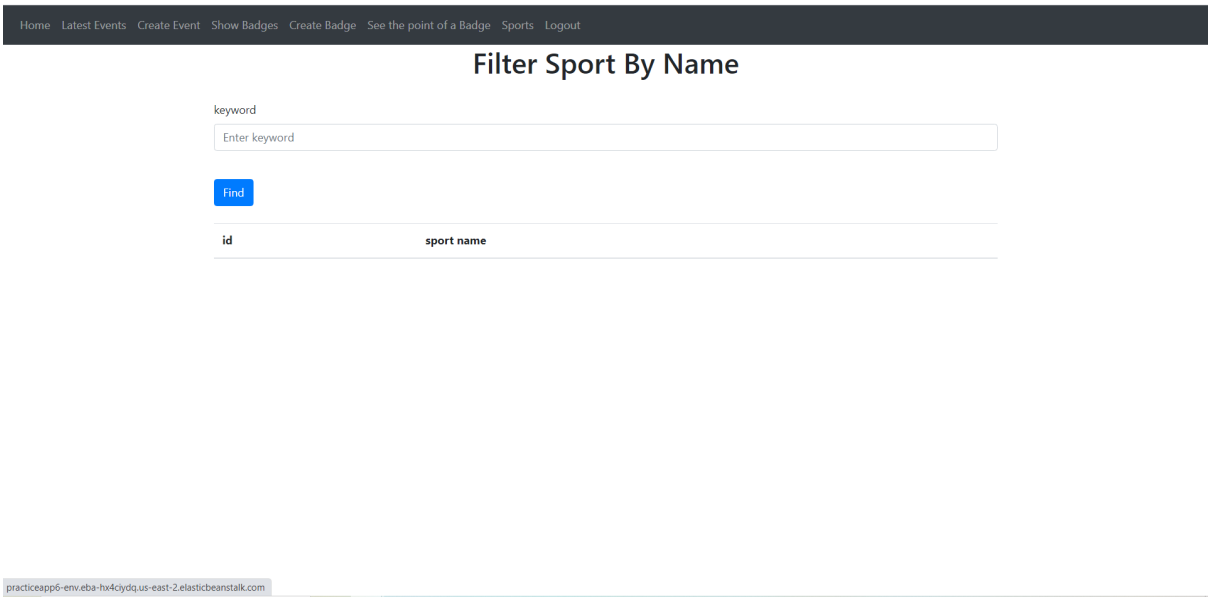
Frontend Part with Screenshots:

This is the main handler for the url sports. User is shown a text field to filter the sport names. Then a request to the api uri /api/v1.0/sports?keyword=<keyword> is made. If an error occurs a flash with error message is shown, otherwise, the list of sports that contain the keyword in their names is listed.
Here exact match is needed for filtering. “Basketball” will not match “basket” keyword.

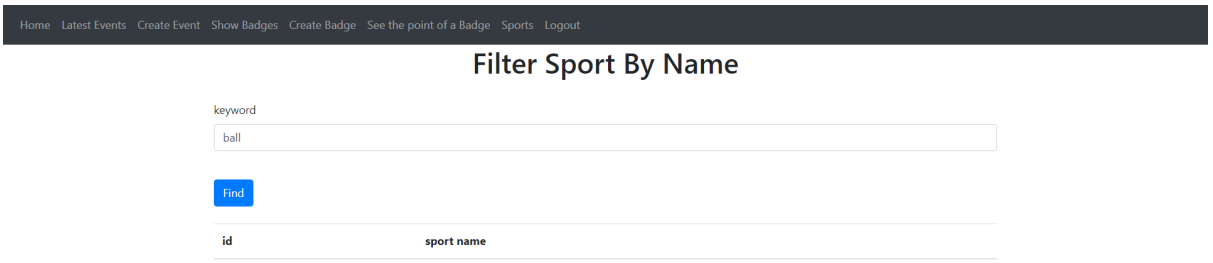
```
"""
    List all sport id-name pairs with a name filter
"""
@views.route('sports/', methods=['POST','GET'])
def sports_page():
    if request.method == 'POST':
        keyword = request.form.get('keyword')
        # Call sports API to get filtered sports.
        req = BASE_URL+"/api/v1.0/sports?keyword=" + keyword
        headers = {'Content-type': 'application/json'}
        response = requests.get(req, headers=headers)
        items = response.json()['sports']

        # No error occurred, list sports
        if response.status_code == 200:
            return render_template("sports.html", user= current_user,
items=items)
        # Error occurred, error message
        else:
            flash('Some Error Occured', category='error')

    return render_template("sports.html", user= current_user, items=[])
```

Screenshots:

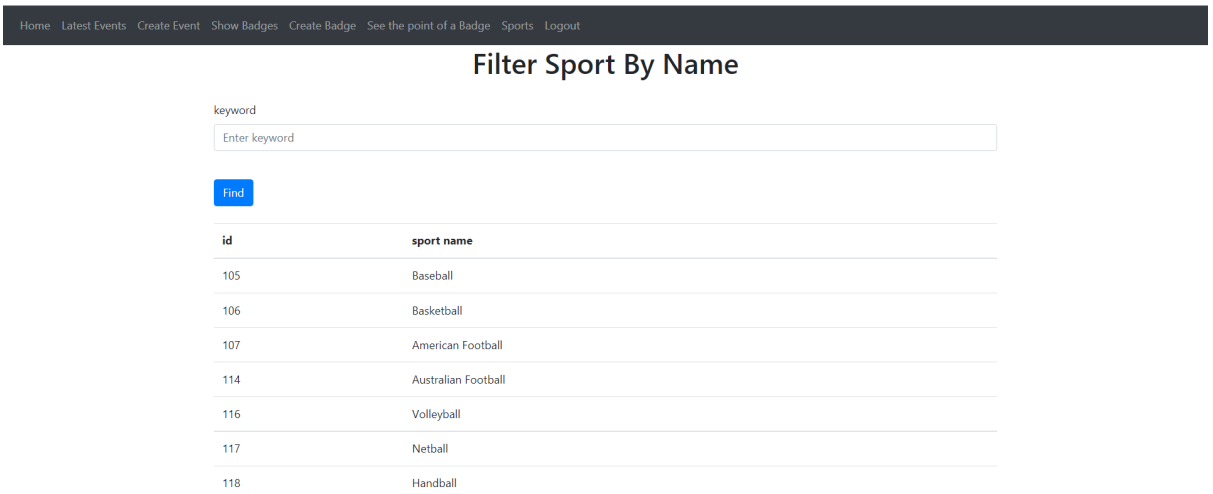
Initial page



User enters the keyword



Clicks find and results are listed:



4. Discussion Related Endpoints

4.1. Viewing the discussions for an event

Endpoint: /events/<event_id>/discussions

Method: Get
Authorization: Not Needed
Author: Musa Nuri İhtiyar
Parameters: {
"event_id": ID of the event
}

Expected Response When No Error:
{
"id": ID of the event
"description": Description of the sport type of the event
"text": Discussions of the event
}

Responses:
201: "Discussion has been found and returned in the proper format"
Example Response:
{
"id": 1,
"description": "Competitive and recreative sport opposing two teams, basketball is played on a court where two hoops are suspended in height at each extremity. With your team, your goal is to score points by throwing the ball into the basket. You will have to do some feints and dribble the ball to progress on the court and get as close as

possible to the basket before throwing the ball. Can you jump high enough to make a dunker?",

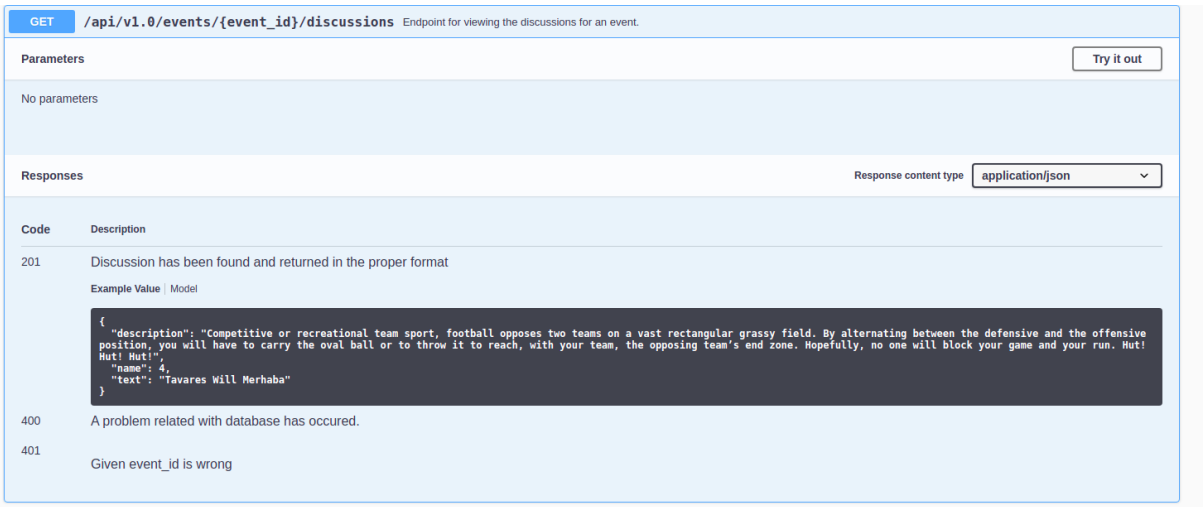
"text": "Hello how are you?",

}

400: "A problem related with database has occured."

401: "Given event_id is wrong"

Swagger Documentation



Related functions:

API:

This is API for which I have defined GET part. Normally it gets the discussions for the given event id by also adding a description for the sport type of the event. If an abnormal situation such as a wrong parameter or a database error occurs, appropriate message and signal are returned.

```
# When the id of the event given, corresponding discussion is returned
by adding the definition of the sport type in the json format
# Corresponding event must exist and have a sport type in db.
@events.route('<event_id>/discussions', methods=['GET', 'POST'])
@swag_from('doc/discussionForEvent_GET.yml', methods=['GET'])
def discussionForEvent(event_id):

   if request.method == 'GET':

       if int(event_id) <=-1:
           return "Wrong path parameters", 401
```

```
        try:
            eventList = Event.query.all()
        except exc.NoReferenceError as e:
            return "Database error", 400
        eventList = Event.query.all()


        sportName = ''

        # Finds the event with the given id and its sport type
        for i in range(len(eventList)):
            if eventList[i].serialize()["id"] == int(event_id):
                sportName = eventList[i].serialize()["sport"]


        # ############# New

        sportList = Sports.query.all()

        for i in range(len(sportList)):
            if sportList[i].serialize()["id"] == int(sportName):
                sportName = sportList[i].serialize()["sport"]
                break

         ############# New

        description = 'No definition found for ' + sportName

        # Find the corresponding definition for the sport type
        response = requests.get(
            'https://sports.api.decathlon.com/sports/' +
sportName.lower())  # API to use
        if response.status_code >= 200 and response.status_code < 300:
            json_data = json.loads(response.text)
            description = json_data['data']['attributes']['description']
            if description == None:
                description = 'No definition found for ' + sportName

        try:
            discussionPostList = DiscussionPost.query.all()
        except exc.NoReferenceError as e:
            return "Database error", 400

        discussionPostList = DiscussionPost.query.all()
        # Get the discussion from the database for the given event
        text = 'No discussion found'
```

```
        for i in range(len(discussionPostList)):
            if discussionPostList[i].serialize()["id"] == int(event_id):
                text = discussionPostList[i].serialize()["text"]

        result = {"id": event_id, "description": description, "text":
text}
        return jsonify(result), 201
```

Unit tests:

Also I have unit tests, yet they're a bit different. In the meeting with TA, we've asked what is meant by unit test. What I have understood was testing the parts of the code which doesn't send a request or interact with database. Almost all of the code I have written was doing either of the mentioned points, so I decided to write an additional function for which I will write test cases in order to learn the topic better. I have tried to design it as close as possible to the project. It works as the following. It gets sent messages for an event and names of the people related with an event. Then it checks whether the number of sent messages are at least 3 or messages contain the name of the people given as a list.

I have divided these functionalities into 2 different new functions to be able to write more tests. All three of the functions and their unit tests are like the following.

```python
# Checks the number of messages sent in the discussion
def hasMoreThan2Messages(messages):
  return len(messages) > 2

class TestHasMoreThan2Messages(unittest.TestCase):

  def test_1(self):
      # For no discussion case
      messages = []
      result = hasMoreThan2Messages(messages)
      self.assertEqual(result, False)

  def test_2(self):
      # For 2 messages
      messages = ['Hello how are you?', 'Fine thanks, and you?']
      result = hasMoreThan2Messages(messages)
      self.assertEqual(result, False)

  def test_3(self):
      # For 3 messages
      messages = ['Hello how are you?', 'Fine thanks, and you?', 'Good']
      result = hasMoreThan2Messages(messages)
      self.assertEqual(result, True)


# Checks if at least one of the people is mentioned in the
# discussion or not
def hasReferenceToName(messages, namesOfPeople):
  for i in range(len(messages)):
      for j in range(len(namesOfPeople)):
          if namesOfPeople[j].lower() in messages[i].lower():
              return True
  return False


class TestHasReferenceToName(unittest.TestCase):

  def test_1(self):
      # For not mentioned case
      messages = ['How are you today?', 'I am very happy']
```

```
        namesOfPeople = ['Ayse', 'Ahmet', 'Hasan', 'Osman']

        result = hasReferenceToName(messages, namesOfPeople)
        self.assertEqual(result, False)

    def test_2(self):
        # For mentioned with the first letter as capital case
        messages = ['Hello how are you Ayse?', 'Fine thanks, and you?',
'Thanks I am fine', 'Good']
        namesOfPeople = ['Ayse', 'Ahmet', 'Hasan', 'Osman']

        result = hasReferenceToName(messages, namesOfPeople)
        self.assertEqual(result, True)

    def test_3(self):
        # For mentioned with the all letters as capital case
        messages = ['Hello how are you OSMAN?', 'Fine thanks, and you?']
        namesOfPeople = ['Ayse', 'Ahmet', 'Hasan', 'Osman']

        result = hasReferenceToName(messages, namesOfPeople)
        self.assertEqual(result, True)

    def test_4(self):
        # For mentioned without any capital letter case
        messages = ['Hello how are you?', 'Fine thanks, and you frank?',
'Thanks I am fine', 'Good']
        namesOfPeople = ['Ayse', 'Ahmet', 'Hasan', 'Osman', 'Frank']

        result = hasReferenceToName(messages, namesOfPeople)
        self.assertEqual(result, True)


# If an event has more than 3 messages in discussions
# or a person is referenced by name in one of the messages,
# the event is special. The function isn't case sensitive.
def areDiscussionsForEventSpecial(messages, namesOfPeople):
    return hasMoreThan2Messages(messages) or hasReferenceToName(messages,
namesOfPeople)
```

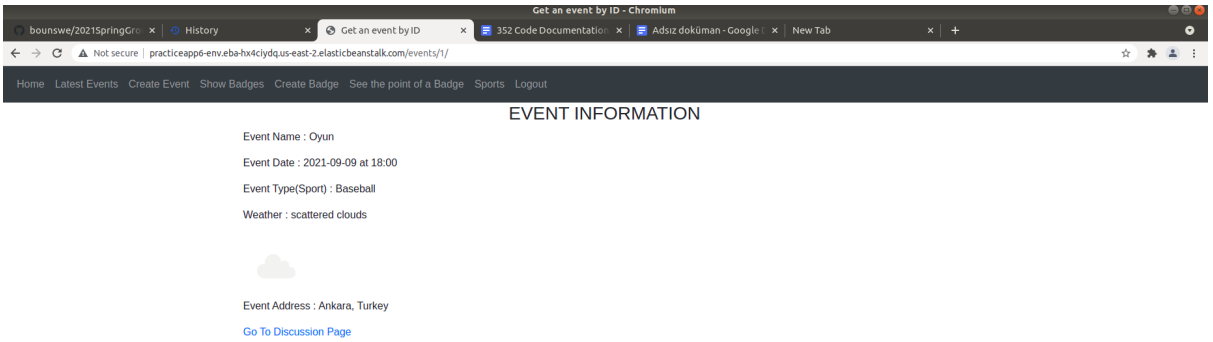
```python
class TestAreDiscussionsForEventSpecial(unittest.TestCase):

  def test_1(self):
      # For no discussion case
      messages = []
      namesOfPeople = ['Ayse', 'Ahmet', 'Hasan', 'Osman']

      result = areDiscussionsForEventSpecial(messages, namesOfPeople)
      self.assertEqual(result, False)

  def test_2(self):
      # For more than 2 discussions without referencing
      messages = ['Hello how are you?', 'Fine thanks, and you?', 'Thanks I
am fine', 'Good']
      namesOfPeople = ['Ayse', 'Ahmet', 'Hasan', 'Osman']

      result = areDiscussionsForEventSpecial(messages, namesOfPeople)
      self.assertEqual(result, True)

  def test_3(self):
      # For less than 3 discussions, but with referencing
      messages = ['Hello how are you OSMAN?', 'Fine thanks, and you?']
      namesOfPeople = ['Ayse', 'Ahmet', 'Hasan', 'Osman']

      result = areDiscussionsForEventSpecial(messages, namesOfPeople)
      self.assertEqual(result, True)

  def test_4(self):
      # For no discussion case
      messages = ['Hello how are you Ahmet?', 'Fine thanks, and you?',
'Thanks I am fine', 'Good']
      namesOfPeople = ['Ayse', 'Ahmet', 'Hasan', 'Osman']

      result = areDiscussionsForEventSpecial(messages, namesOfPeople)
      self.assertEqual(result, True)
```

Front-end:

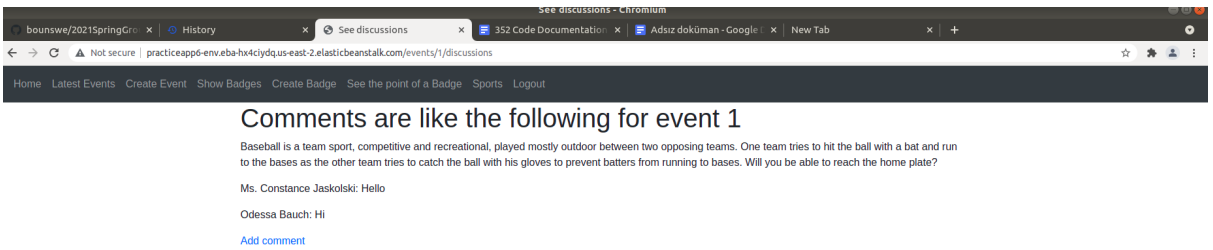
This section is the one where I define the described page in python. It mainly gets the discussions using the API I have defined. Then unless there's a problem, it shows the discussions as described in the previous part. Else, it says Error.

```
# Shows the discussion page for the event with id event_id.
# It also shows the description of the sport type of the event
# using an external API
@views.route('/events/<event_id>/discussions', methods=["GET"])
@login_required
def discussionPage(event_id):

  BASE = 'http://127.0.0.1:5000/'  #  should be changed
  response = requests.get(BASE + '/api/v1.0/events/' + event_id +
'/discussions')

  if response.status_code == 201:
      description = response.json()["description"]
      text = response.json()["text"]
      return render_template("discussionPage.html", user= current_user,
event_id=event_id, definition = description, text = text.split('#'))
  else:
      return f"<h1>Error<h1>"
```

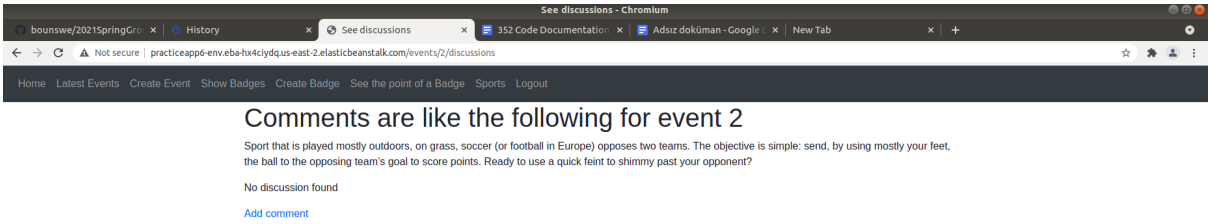
Screenshots from the front-end:
How to go to the discussion page using the page of an event.



After the click available discussions are shown by also adding the description of the sport type on top of messages.



If no message is sent, then only the description is shown and necessary explanation about the nonexistence of the discussions for event is described as the following.



4.2. Posting a comment to an event's discussion

Endpoint: api/v1.0/events/<event_id>/discussions

Method: Post
Authorization: Required
Author: Can Atakan Uğur
Parameters: {
"event_id": ID of the event
"text": The text of the comment to be posted.
}

Expected Response When There is no Error:
{
"id": ID of the event
"text": The text of the post.
}

Responses:
201: "Discussion has been found and returned in the proper format"
Example Response:
{
"id": 12,
"text": "When will the event start exactly?",
}
400: "Wrong parameters."
503: "Service not available."

Swagger Documentation

POST `/api/v1.0/events/{event_id}/discussions` Endpoint to create a discussion post

Parameters Try it out

Name	Description
body * required *(body)*	Example Value | Model `{ "event_id": 12, "text": "When will the event start, exactly?" }` Parameter content type application/json

Responses Response content type application/json

Code	Description
201	New comment to the discussion is created and added to the database. Example Value | Model `{ "event_id": 12, "text": "When will the event start, exactly?" }`
400	Wrong parameters.
503	Service not available.

Related Functions

There is one function defined for this endpoint and it handles both GET and POST requests. Since only the POST method is explained here, please note that the GET part is collapsed and is not shown in the screenshot.

I handled POST requests in my individual work, in other words, my code will work if the user wants to post a comment to an event's discussion page. First the checks about the parameters are done. If the event_id is invalid and or the event does not exist, appropriate error messages are returned. Also, the text of the comment is checked if it is empty or not.

If the request is able to pass these tests, then the comment is added and the necessary changes in the database are made. If an error occurs during the database transactions, an error message with the code 503 is returned.

```python
# checks whether an entered comment has text
def check_comment_has_text(message):
    if len(message) < 1:
        return False
    else:
        return True
```

```python
"""
This method handles all requests for adding a discussion post and getting all
discussion posts in order to show those in the event's discussion page.
"""
@events.route('<event_id>/discussions', methods=['GET', 'POST'])
@swag_from('doc/discussionForEvent_GET.yml', methods=['GET'])
@swag_from('doc/discussionForEvent_POST.yml', methods=['POST'])
def discussionForEvent(event_id):

    if request.method == 'GET':…


    elif request.method == 'POST': # if the user wants to post a comment to a discussion

        # event_id is wrong
        if int(event_id) < 0:
            return jsonify({"error":"Wrong Parameters"}), 400


        # users shall not be able to enter a comment to a nonexistent event
        event = Event.query.get(int(event_id))
        if not event:
            return jsonify({"error":"Event Does Not Exist"}), 400

        # get a random name
        response = requests.get(
                'https://api.namefake.com/english-united-states/random/')

        # if the GET request sent is successful, use the name returned
        if response.status_code >= 200 and response.status_code < 300:
            data = json.loads(response.content)
            name_shown = data['name']
```

```python
else: # if cannot get the name for some reason, show the user's name as "No Name"
    name_shown = "No Name"


message = request.json["comment"]
test = check_comment_has_text(str(message))
# if the text field is empty, this request is erroneous
if not test:
    return jsonify({"error":"Text Field Cannot Be Empty"}), 400

message = name_shown + ': ' + message # combine the comment with the name

discussionPostList = DiscussionPost.query.all()

doesExist = False
for i in range(len(discussionPostList)):
    if discussionPostList[i].serialize()["id"] == int(event_id):
        # check if there exists a discussion page for that event_id
        doesExist = True
        break

# if there is not a discussion page for this event, create one
if not doesExist:
    newPost = DiscussionPost(id=event_id, text=message)
    try:
        db.session.add(newPost)
        db.session.commit()
        return jsonify(newPost.serialize()), 201
    except exc.SQLAlchemyError as e:
        db.session.rollback()
        return jsonify({"error":"Service Unavailable"}), 503
```

```python
# if there is already a discussion page for this event, update it
else:
    try:
        currentRow = DiscussionPost.query.filter_by(id=event_id).first()
        # different comments under the same discussion page are separated by sharp as we agreed on
        currentRow.text += '#' + message
        db.session.commit()
        return jsonify(currentRow.serialize()), 201
    except exc.SQLAlchemyError as e:
        db.session.rollback()
        return jsonify({"error":"Service Unavailable"}), 503
```

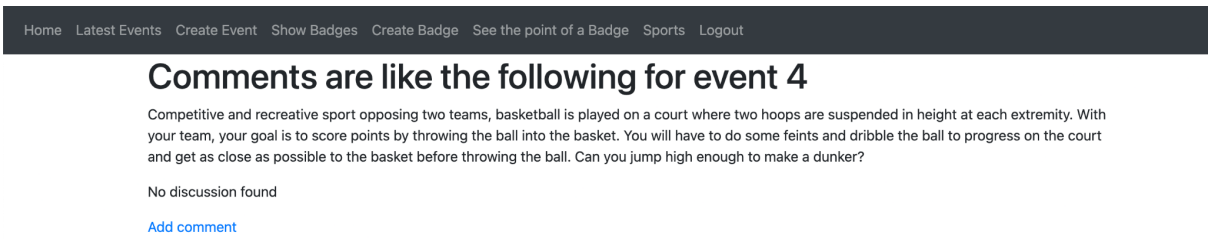
Unit Tests:

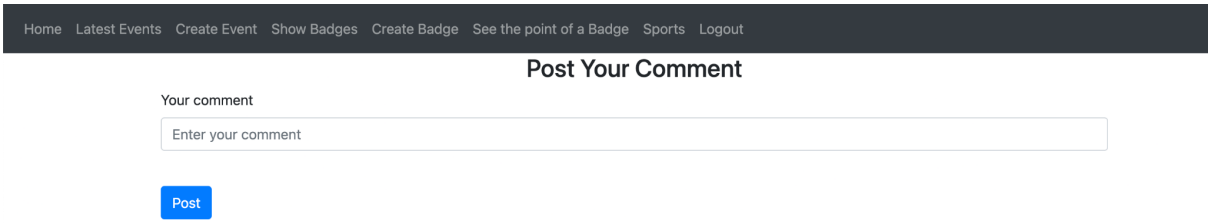
I also coded unit tests merely for the code segment that is not involved in an external transaction like sending requests or database transactions. Here, this code tests whether the method is able to distinguish between the empty and well-formed comments.

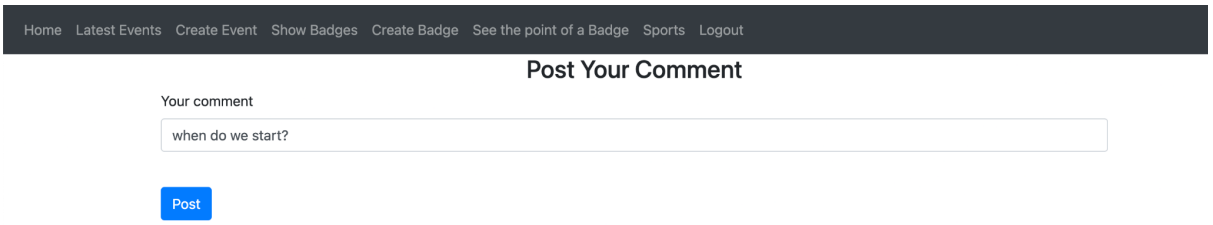
```python
import unittest
import requests, json
from ..api.event import check_comment_has_text

class TestEventAPI(unittest.TestCase):

    # given an empty comment, it returns false
    def test_comment_empty(self):
        comment = ""
        result = check_comment_has_text(comment)
        self.assertEqual(result, False)

    # given any comment that has text, it returns true
    def test_comment_has_text(self):
        comment = "When does the event start?"
        result = check_comment_has_text(comment)
        self.assertEqual(result, True)
```

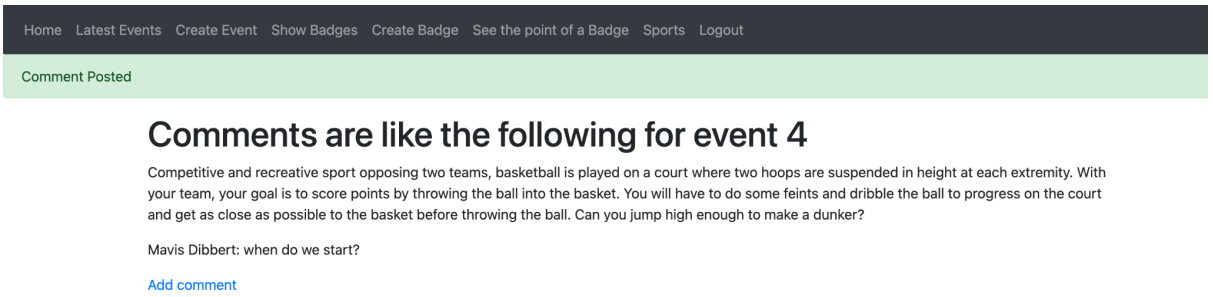
Screenshots from Front-End

1) Adding a comment to a present event successfully

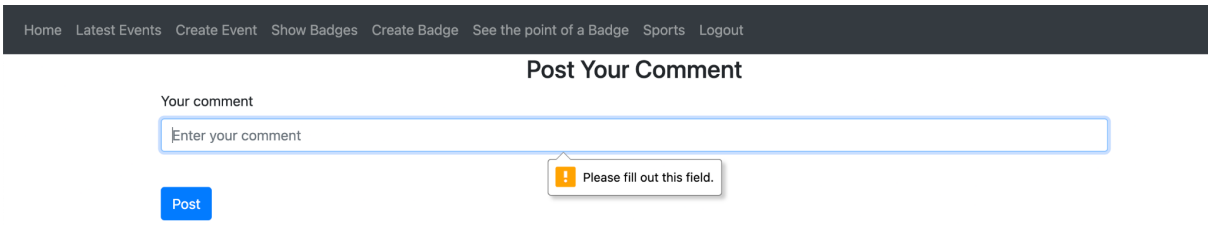








2) The front-end design does not let the user not fill the comment but even if the request is sent with an empty text field, the API would return an error.



5. Equipment Related Endpoint

5.1. Posting the equipments to the database

Endpoint: /equipments/

Method: Post
Authorization: Not Needed
Author: Ömer Faruk Süve
Parameters: {
"name": name of the equipment
}

Expected Response When No Error:
{
"id": ID of the equipment
"name": Entered name of the equipment
"price": Price of the equipment
}

Responses:
201: "Equipment Created!"
Example Response:
{
"id": 1,
"name": "Ball",
"price": "10.0",
}
400: "Check Information Entered"
409: "Equipment Already Exists!"

Related Functions:

```
# returns the arbitrary price for equipment using an API
# due to not having a price api, selection random number as index 0
def get_price():
   url = 'https://api.blockchain.com/v3/exchange/tickers'
   response = requests.get(url)
   return response.json()[0]['price_24h']
```

Endpoint Code:

```
# endpoint of the api of equipments
# applies a Post operation
# documentation for Post in .yml file
@equipments.route('/', methods=['POST'])
@swag_from('doc/equipments_POST.yml', methods=['POST'])
def post_equipment():
   # checking parameters
   if not request.json or not 'name' in request.json:
       return "Parameter not correct", 400


   result = request.json
   equipment_name = result['name']
   price = get_price()


   tempEquipment =
Equipment.query.filter_by(name=equipment_name).first()

   # if equipment exists, returns code 409
   # handling error if an error occurs on Sqlalchemy
   # returns 201 if everything is fine
   if tempEquipment:
       return 'Equipment already exists!', 409
   else:
       new_equipment = Equipment(name=equipment_name, price=price)
       try:
           db.session.add(new_equipment)
           db.session.commit()
       except exc.SQLAlchemyError as e:
           db.session.rollback()
           return "Try Later", 403

       return jsonify(new_equipment.serialize()), 201
```

6. Authentication

6.1. Adding User to the database

Endpoint: /sign-up/

Method: Post

Author: Ömer Faruk Süve
Parameters: {
"email": email of the user
"first_name": name of the user
"password1": password of the user
"password2": password confirmation
}

Expected Response When No Error:
{
"id": ID of the user
"email": email of the user
"first_name": entered name of the user
"password": encrypted by sha256 password of the user
}

Responses (There is no backend API for this):
If category: "success":
"Successful"
Example Response:
{
"id": 1,
"email": "omerfaruk@gmail.com",
"first_name": "Ömer",
"password": "omer123",
}
If category: "error":
"Email already exists"
"First name must be greater than 1"
"Passwords are not same!"
"Minimum Password length is 8 chars"

Related Functions:

```
# endpoint for User Sign-up
# gets the necessary inputs, checks if email exists
# checks several steps, if does not fit, gives error
# if no error, then creates new user by entered inputs, adds to the database
# generate hash for password using sha266
def sign_up():
   if request.method == 'POST':
      email = request.form.get('email')
      first_name = request.form.get('firstName')
      password1 = request.form.get('password1')
      password2 = request.form.get('password2')
```

```
        user = User.query.filter_by(email=email).first()
        if user:
            flash('Email already exists', category='error')
        elif len(email) < 4:
            flash('Email must be greater that 4 chars.', category='error')
        elif len(first_name) < 2:
            flash('First name must be grater than 1 chars.', category='error')
        elif password1 != password2:
            flash('Passwords are not same!', category='error')
        elif len(password1) < 8:
            flash('Minimum Password length is 8 chars', category='error')
        else:
            new_user = User(email=email, first_name=first_name,
password=generate_password_hash(password1, method='sha256'))
            db.session.add(new_user)
            db.session.commit()
            login_user(new_user, remember=True)
            flash('Successful', category='success')
            return redirect(url_for('views.home'))
    return render_template("sign_up.html", user=current_user)
```

6.2. User Login

Endpoint: /login/

Method: Post
Author: Ömer Faruk Süve
Parameters: {
"email": email of the user
"password": password of the user
}

Expected Response When No Error:
{
"id": ID of the user
"email": email of the user
"first_name": entered name of the user
"password": encrypted by sha256 password of the user
}

Responses (There is no backend API for this):
If category: "success":
"Logged in successfully!"
Example Response:
{
"id": 1,

"email": "omerfaruk@gmail.com",
"first_name": "Ömer",
"password": "omer123",
}

If category: “error”:

```
"Incorrect password, try again."
"Email does not exist."
```

Related Functions:

```
# endpoint for User Login
# filters by email; if email does not exists, gives error
# if email exists, checks password; if does not match, gives error
@auth.route('/login', methods=['GET', 'POST'])
def login():
   if request.method == 'POST':
      email = request.form.get('email')
      password = request.form.get('password')
      user = User.query.filter_by(email=email).first()
      if user:
         if check_password_hash(user.password, password):
            flash('Logged in successfully!', category='success')
            login_user(user, remember=True)
            return redirect(url_for('views.home'))
          else:
            flash('Incorrect password, try again.', category='error')
      else:
         flash('Email does not exist.', category='error')

   return render_template("login.html", user=current_user)
```

6.3. User Logout

Endpoint: /logout/

Method: Post
Author: Ömer Faruk Süve
Parameters: None
Expected Response When No Error: Returns Login page
Responses (There is no backend API for this): No Response

Related Functions:

```
# endpoint for User Logout
# logs out using imported logout_user function
@auth.route('/logout')
@login_required
def logout():
```

```
    logout_user()
    return redirect(url_for('auth.login'))
```