**Technical Report**

This report presents the implementation details of the MUG application. MUG is a web-based application that guides a museum visitor among the exhibits, providing a map of the museum. The user experience is enriched by accessing the details about exponents with the help of QR codes.

**Project Progress**

For the final presentation of our project, we managed to implement the following:

* Application access via QR code
* Deployed application in Google Cloud environment
* Authentication and login
* Museums floor plan and presentation of each's museum exhibitions
* QR code exhibit information search
* User friendly and interactive interface

**Implemented API**

For our application we have implemented one API that aggregates four services:

1. Spring Boot authentication service
2. Spring Boot with SQLite integration for storing the exhibits
3. Spring Boot with Apache Jena adaptor for SPARQL queries to Wikidata endpoint
4. Main service that aggregates the above services

All of these are aggregate into one service which serves as a bridge between the front-end and the back-end.

The four services and the Node.js user application are hosted on Google Compute Engine, each on a virtual machine running Ubuntu.

**Use of knowledge sources**

Our application serves as a museum guide and the museums we chose for the purpose of this project are the four museums from the Palace of Culture Iasi.

The Palace of Culture hosts the following museums:

* Art Museum
* Ethnographic Museum
* Science Museum
* History Museum

All of these museums host well-known exhibits from Romanian artists and are a source of Romanian culture from centruies ago until present days.

We went to the Palace of Culture, made an idea about the exhibits and took some photos so that the information presented to the user could be as much closer as possible to reality. After this research, we searched Wikidata for the exhibits we chose to present in our application and, for our surprise, didn't find any relevant information. For this reason, we added a few new items to Wikidata, but the approval of the content was not immediate. In order to achieve the project's goal, having SPARQL queries to an endpoint, we decided that we will keep the exhibits' photos in a persistent SQL database and for the sake of this project, we will query Wikidata for exhibits similar to the one from the Palace of Culture's museums.

Although the approval of our items was not immediate, the items were finally added to Wikidata, thus contributing to the community. Below are the two items we managed to add to Wikidata.

Some of the images used in our application ar etaken from the Palace of Culture's site and some were taken by us when visiting the museums for research.

**Preliminary considerations about the project - Midterm evaluation**

**Preliminary considerations about the internal data structures/models to be used**

**Scenario: MockGps registration and subsequent packet receival.**

The api will send a packet to the mock gps service with the following structure:   
**{   
   “user\_token”: “23fgd343fdsfds3ssasda”,   
   “target\_museum”: “Antipa Museum”   
   “strategy”: “path”   
}**

* **user\_token** is the authentication token for the current user. We use this data to differentiate different users authenticated in the system
* **target\_museum** is the name of the museum. The mock gps will have saved coordinates for a list of already existing museums. The user has to specify the targeted museum before the service will generate coordinates
* **strategy** the coordinates can be generate randomly or in a direction

The mock gps service will send sequential packets with the following form:  
**{   
   “longitude”: 45.2,   
   “latitude”: 34.2,   
   “direction”: “LEFT”   
}**

* **longitude**- geographical longitude
* **latitude**- geographical latitude
* **direction**- the coordinates generated by the service if using any strategy are in a direction. The value of this field is compared to the previous pack. So if this pack is going left related to the previous one, the direction is left

**Scenario: Museum Exponates Service, query, review and rating**

**1.Querying an exponate**   
When at a particular position in the tour, in order to render the right UI graphics, the part of the app running on the user’s device will query the app for the which exponates are available at those coordinates. The input for the query action looks like this:  
**{   
   “action”: “query\_exponate”   
   “selected\_museum”: “antipamuseum”   
   “longitude”: 45.5,   
   “latitude”: 46.3,   
   “user\_token”: “3234jhgg234j2h34gj23”   
}**

* **action**- this field is present in the packets that come from the end client in order to make distinction of the action the end user wants to do and to know how to properly dispatch the request
* **selected\_museum**- The museum in which the user is currently in
* **longitude**- geographical coordinate
* **latitude**- geographical coordinate
* **user\_token**- the token generated when the user logged in

The produced output will have the following form:  
**{   
   “exponate\_id”: “34”   
   “exponate\_name”: “mamut siberian”   
   “graphics”: “path/to/exponate/graphics”,   
   “data”: “information about exponate”   
   “rating”: 5   
}**

* **exponate\_id**- the id we currently use to identify this exponate
* **exponate\_name**- the name of the exponate
* **graphics**- pictures or any media used to graphically expose the exponate in the user’s UI
* **data**- information about an exponate, can be retrieved from the database or using the knowledge service
* **rating**- the average of the ratings the users gave to this exponate

**2. Load reviews**  
The user sends this pack to load the ratings for an exponate:  
**{   
   “action”: “load\_comments”,   
   “user\_token”: “32423hjg42j3h4gj2h3g423”,   
   “exponate\_index”: 34   
   “current\_comment\_index”: 0   
}**

* **exponate\_index**- the id we currently use to identify this exponate
* **current\_comment\_index**- the server will respond with multiple packets because the there be a lot of comments and loading all at once is just not that efficient. So sending the last comment index is the right way to do this action

The packets sent from the server will have the following structure:  
**{   
   “comments”: [   
     {   
          “author”: “author\_name”,  
          “text”: “The siberian mamut is unique in Romania”,  
          “rating”: 3.5  
     }  
     ...  
   ]  
}**

* **comments** contains a list of json pack which have the following structure: author, the actual comment, number of stars for this comment

**3. Write review**  
To write a review to an exponate, the end user will send a packet with the following structure:  
**{   
   “user\_token”: “433hjg3jh45345ghj34g534hj53gj45”,   
   “review”: “....”,   
   “exponate\_id”: 34   
}**And the server will respond with one of the standard http statuses if the review was created or not or something else.  
  
**4. Rate an exponate**  
The user will send the following packet in order to rate an exponate:  
**{   
   “user\_token”: “fgdf342fd4dgdfgd32423dgdfgdf”,   
   “rating”: 5,   
   “exponate\_id”: 34   
}**The server will respond with one of standard http statuses if the rating was added or not or something else.

**Scenario: Knowledge service, retrieving information about an exponate**

The data about an exponate can be cached locally in the database we keep for all the exponates or, in the majority of cases, it is looked up on the internet. If it is not found in the database by the museum exponates service, it will provided by the knowledge service. The packet the api send to the knowledge service has the following structure:  
**{   
   “exponate\_name”: “mamut siberian”,   
   “aggregated\_info”: “Mamutul siberian….”   
}**The API will redirect the obtained data to the museum exponates service and then it will rendered back to the user.

**Scenario: Registration/Authentication service, registering and authenticating users**

**1. Authenticating an user**  
When the user authenticates it will send a packet with:  
**{   
   “username”: “foo”,   
   “password”: “56sdfgjds5sdf”,   
   “ip”: “89.98.23.23”   
}**

* **password** - the hashed version for the actual password the user entered

If the credentials the user provided are correct, the server will return the following packet:  
**{   
   “status”: “success”   
   “auth\_token”: “343gh32jg4h23g4234jh234”   
}**And the token will be locally saved in a cookie on the user’s browser.  
  
**2. Registering an user**  
When an user wants to register in the app, he will send a packet with the following structure to the API:  
**{   
   “username”: “foo”,   
   “password”: “sha256bar”,   
   “email\_address”: “foo@bar.com”   
}**If the user is available the API will respond with the following packet:  
**{   
   “status”: “success”   
}**The **status** field will contain the error message if any is needed.

**Considerations about the external data sources**

**Description of the diagram elements**

**1. Mock gps service**

Normally, the device running the app should send information regarding the longitude and latitude to the API. Because we want to emulate. Because it is easier to test using mock data we replaced this with a service that generates this data. On the service we will implement a functionality to manually add the login token of the current user. The API will ask for user position at every 0.5 seconds, instead of asking the phone gps it will ask the mock service. This means that also the graphics that the user is seeing locally will also be updated based on the mock data generated by this service.  
  
**2. Registration and authentication service**  
Initially, the user will be asked for email, real name and a password to be kept inside the database. Based on the email and password he will be authenticated later. When an user is successfully authenticated a token is generated, this token is used by the rest of the services.  
  
**3. Knowledge base service**  
Whenever an user is near an exponate, the knowledge base service will be queried for information about that thing. This service agregates results obtained from multiple sources like **wikidata**.  
  
**4.Exponates service**  
All the exponates ever visited on the museum will be saved in a database. This way we can store comments and other user related information about that item. When adding a comment, this service **will be queried**.  
  
**5. SPARQL endpoint**  
After adding info about an exponate this data will be transferred to the SPARQL endpoint. This service just exposes the data to an SPARQL endpoint.  
  
**6. API**  
This is the main component connecting all the services. First has the role of dispatching the UI to the users. The it dispatches the users request to the appropiate service from the ones described above and processes the about so it can be delivered back to the user.