Social Routing

Project and Seminar

ISEL - Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

Authors:
Baltasar Brito
email: baltasar.brito@gmail.com
phone: 915953552
Bernardo Costa
email: bjmcosta97@gmail.com
phone: 913897555

Supervisor:
Pedro Félix
email: pedrofelix@cc.isel.ipl.pt

May 27, 2019

Contents

1 Introduction

contexto do pronlema propor resolucao estrutura do doc

Background

talvez dentro da introducao

trabalho relacionado, sistemas ou aplicações similares características dos dados

2 System Architecture

The system architecture organizes the system's necessities into manageable blocks as shown in figure 2.1. It is essentially divided into two major components, the Social Routing Client Application [2] and the Social Routing Service with a third one being the external services.

The role of the Client Application is to provide an interface which the user can interact with. This component communicates with either the Social Routing Service or external services when required through the HTTP protocol [5].

The Social Routing Service processes and stores data that the Client Application can use at any time, as long as the user making the requests is authenticated correctly. It receives HTTP requests and exposes it's functionality through the Social Routing API [15].

The external API's are used to make some otherwise very difficult operations achievable in a short period of time. The API's used will be mentioned inside each of the components's description.

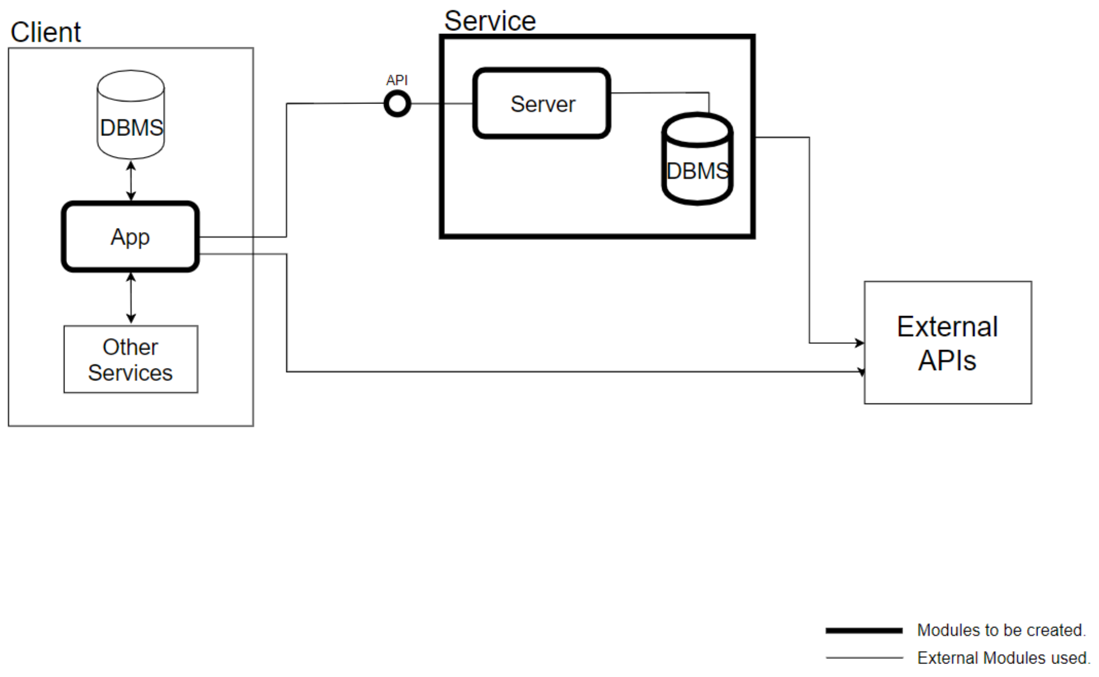

Figure 2.1: System structure.

3 Social Routing Client Application

The Social Routing Client is composed by four major components, each with it's own compromise and objective. The Activities/Fragments[TODO REF] to represent the User Interface (UI) where the user can interact with, the ViewModels[TODO REF] to store and manage UI-related data, a Repository[TODO REF] to handles data operations and knows where to get the data from and a Remote Data Source[TODO REF] to communicate with external components, for instance Social Routing API and Google API'S[TODO REF]. This logic is represented in Figure [TODO FIG]

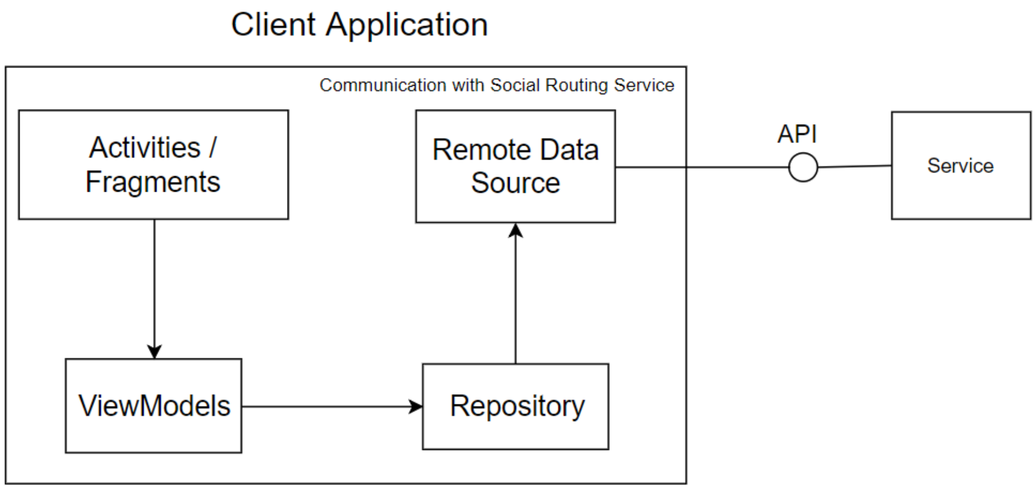

Figure 3.1: Social Routing Client architecture.

Activities/Fragments

The concern of this component is to represent the user the Application flow, where the user can navigate and interact with. All activities extend of a BaseActivity that as a global behaviour, like when the data was changed is necessary to reformulate the view to the user see the information updated.

Each Activity has its own :

- Design: defined by one or more layouts that contains buttons, images, input text, fragments (for instance the map fragment provided by Google), where the user can interact.
- Behaviour: when de data is changed the view need to be reformulated, and this behaviour is defined with a success behaviour and a Error behaviour, using the ViewModel.

A example activity is the NavigationActivity where the user can navigate to all the functionalities of the application, after google authentication, like the creation of a route, search by routes and the user profile. This activity as a left panel where you can navigate to other activities with different purposes like the route creation and the user profile, however the activity it self has the search functionality, where you can input text like the location name that is pretended and a button to navigate to the activity that shows the results of that search.

ViewModels

Component used when the UI experiences a change, the ViewModel calls other components to load the data, and it can forward user requests to modify the data, however the ViewModel doesn't know about UI components, it is completely separated from them.
This component has a simple implementation, the application contains two ViewModels one for the Routes information (get, creation, update, search) and the other to the User (get, delete). It uses the repository to obtain the data and then return it in the shape of Livedata[TODO REF].

Repository

The Repository Handles data operations, knows where to get the data from and what API calls to make when data is updated. A repository can be considered to be a mediator between different data sources, such as web services.
The application has a repository specified to the Social Routing API and another to Google Maps API. Each one as correspondent Web Service that uses the framework Retrofit[TODO REF], used to make a synchronous or asynchronous HTTP request to the remote webserver. The Repository obtains the data from the web server and only have to possible request status Failure and Success and returns the data contained in a LiveData, because when the data suffers and update it could be observable.
The Repository specified to our API (Social Routing API) knows all the endpoints that should make the request, depending on the objective and the functionality, like the endpoints to sign in, to get user info, routes, create a route, get all categories, update a route, etc...
On the other hand the, the other repository is used to make request to the Google Webserver (Google Maps API) about the geocode of a location and the directions to a coordinate in the map.

Remote Data Source

Module that has the objective of communicating with external APIs and it communicates by doing requests to the Social Routing API and the Google Maps API. The Component knows the structure of the HTTP request to the endpoints, like the parameters and meta-data necessary to obtain the required Response. After the request done, the webservers provide the response with Json[TODO REF] format, however the response is deserialized using the library Jackson[TODO REF], to convert it to Object. So was defined all the input model objects, to automatically deserialize the response to object.

The Client Application has the minimum API level 19 and the target API level is 28, so the Platform version is the Android 9. It uses the Kotlin as unique the programming language in the project, is a object oriented programming language and is getting more and more used nowadays. The goal was to improve the coding experience in a way that was practical and effectual. Kotlin is entirely compatible with Java and was specifically designed to improve existing Java models by offering solutions to API design deficiencies. The core functionalities of the application require a map to create the routes and to show them, the way that was done was using the Google Maps, setting up the Google Play Services to the project.
All the functionalities of the application are provided from de Social Routing Server, except all that is related to the Google Maps. All the information is required from the server doing requests to the correspondent endpoint and is always necessary send the token created by the server, except when is the authentication request that is needed to send to the server the idToken provided from the Google Account. The request that are related to obtain locations or to the Map is necessary to make a specific request to the Google Maps API.
As an example, the user first experience flow of the application is the following:

- The first screen is the user authentication with the backend server using the google account.
- The user will be redirected to a navigation screen that contains a routes search bar and a left panel with buttons to redirect to the screens of user profile and route creation.
- After the user search routes using a location, will be redirect to a new screen, that obtains a list of routes.
- After clicked in a route, the user will be redirect to a screen that shows the map, the route in it and a button to start Live Tracking.
- The user starts Live Tracking is showed the optimal way to the route.
- The user wants to see the his profile, so he may go back until the navigation screen and click in the left panel and then in the User Profile button.

- Is showed the user information (user rating, name, email and routes created).
- Then the user wants to create a route, go back to Navigation screen again and click in the button Route Creation.
- Is redirected to a new screen that shows the map, a button to finish and a form, asking the location of the route that will be created.
- User inserts the location and the map zoom in into the chosen location.
- The User clicks in the map to select the path of the route, if something wrong the user can delete the last point of the route clicking the button on the top of the screen.
- When finished the user click in the button to fill the final form that contains the name, description and category of the route.

4 Social Routing Service

The Social Routing Service is comprised of three major components, each with it's own purpose. A database to store user related information, a server to process data and an API to expose it's functionality. This logic is represented in the figure 4.1.

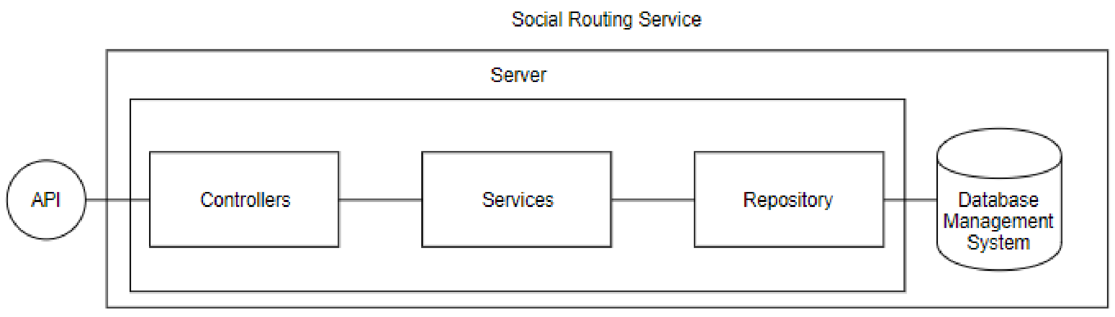

Figure 4.1: Social Routing Service architecture.

Social Routing API

Schema

The Social Routing API uses the HTTP protocol as a medium to communicate and all the data sent or received must be in the JSON [9] format. The base endpoint of the API is : http://api.sr.

Data obtained from the API is either a single resource or a collection of resources. For example, a request made to retrieve a route will have as a response a single resource which will contain the representation of that route. If a request is made to obtain routes by location then the response will be a collection of resources containing the several route representations. This single and collection terms when associated with a resource were decisive in the choice of how much data would be returned in either of them. A request for a collection doesn't require detailed information about each element of the collection, it needs to provide general information and a form for the API user to retrieve detailed information about a specific collection element. With this idea in mind the resource representations were divided into two types, a detailed representation for when a user requests a single resource and summary representation for each element inside a requested collection, containing only the information about that element that is necessary.

Examples of both a detailed and summary representation can be found in the Schema Documentation [18].

Authentication

The API's authentication is made with the support of the Google Sign-In API. The authentication with the is done in two phases. The first phase is the registration of a new user. On this phase a POST [6] request must be made to the endpoint: http://api.sr/sign-in/google, with the Google Sign-In token in the body of the request:

```
{
  token : "Google Sign−In token"
}
```

The token is provided by the Google Sign-In API and when it's received in the Service's API, it's authenticity is verified using the Google Sign-in-API utilities. After the validity of the token is verified the subject field is extracted from it to identify the user performing the registration and stored by the service. Then an API token is generated from a time based unique identifier generator, provided by the Log4J library [11], and associated to the previously received subject. Before being stored the token is hashed to increase security. The response to the request will contain the generated token in it's original form so the user haves the required information to make authenticated requests from there on.

The second phase of the authentication process is made in any request that follows user registration. The API user must send a request containing the previously received token in the Authorization HTTP header. When any request is received the headers are retrieved, and checked against the service's database. In this check, the received token is hashed and compared to the hashed version on the database, if no token with that hash is present then the authentication fails and the user receives an error response.

Supported HTTP Methods

Due to the nature of the HTTP protocol, the API supports four different HTTP request methods [14]: GET, POST, PUT and DELETE.

GET

This method is used to retrieve resources from the API. The request:

```
GET http://api.sr/persons/1
```

retrieves a resource representing a person resource with the identifier 1. The response to this request would be:

```
{
  "identifier": 1,
  "rating": 4,
  "routesUrl": "http://api.sr/persons/1/routes"
}
```

POST

The POST HTTP method is used to create resources. It requires that the Content-Type[3] HTTP header is defined and with the value application/json[1]. An example of a post request can be found in the API POST[16] documentation. A response to a POST request has an empty body and returns the location of the created resource in it's Location header. If successful the status code of a POST request response is 201.

PUT

The PUT method is used to replace or update a resource or a collection of resources. Like the post request it requires that the request contains the HTTP header Content-Type defined with application/json. The following request replaces the currently existing resource route with identifier 1 with the one sent in the body of the request. A successful response has the 200 OK status and an empty body. An example of a put request can be found in the API PUT[17] documentation.

DELETE

DELETE, as the name implies is utilized to delete a resource. The request:

```
DELETE http://api.sr/routes/1
```

deletes the route with 1 as identifier. A successful response will have the 200 OK status and an empty body.

Pagination

Requests that return a collection of resources will be paginated to a default value of 5 resources within the collection. A specific page can be requested with the query parameter page. The request:

```
GET http://api.sr/persons/1/routes?page=1
```

returns the first five routes that a person with identifier 1 created. To obtain the next 5 one would simply change de value of page to 2.

Errors

The error responses follow the RFC standard of type problem+json[12]. An error response example:

```
{
  "type": "Social-Routing-API#unsupported-media-type\cite{unsupportedmedia
  "title": "The requested type is not supported.",
  "status": "415",
  "detail": "The xml format is not supported."
}
```

Hypermedia

Some resources have links to other resources. Either to a parent resource or to a detailed representation of a resource within a collection. For example, a user resource haves a link to their created routes, which holds a collection of routes. That same collection haves a link to the profile of the person who created the routes.

Server

The server uses Kotlin as a programming language and the Spring framework[22]. It's role within the system is data receival, data processing, and to respond accordingly. It is divided in three major layers, each with it's role in the Social Routing Service's system. They are the Controllers, the Services and the Repository.

The Controllers are responsible for handling the reception of an HTTP request to the service and are mapped to it's endpoints. Upon receiving a request they will use the available services to perform desired operations either over a set of received data or to generate the requested data.

The Services are responsible for processing data and communicating the the Repository layer.

The Repository layer is the only layer with direct access to the database and as such is responsible for communicating with it. The communication is made through the use of JDBI[8], a library built on top of the driver JDBC[7]. This allows less verbose code while maintaining control over SQL queries.

Besides these three major layers the server contains other important components, the Interceptor[21] and the Exception Handler[20]. There are three different implementations of the Interceptor component.

The Authentication Interceptor is responsible for user authentication before the request reaches a controller. The goal of this implementation is to avoid server overhead, resolving the authentication before the request is processed allows for a fast response if the user is incorrectly authenticated instead of continuing with the unnecessary processing of data.

The Logging Interceptor is used both before and after the request is processed to provide information regarding each request for debugging purposes.

The Media Type Interceptor is used, like the Authentication Interceptor, to avoid overhead, since if a post request is made with wrong Content-Type headers or no headers at all then the service does not support that request and can respond with an error immediately.

The Exception Handler is the component responsible for the handling of exceptions of the system. In the Spring framework there are several ways to handle exceptions, but the choice to make is to either handle the exceptions locally or globally. The handler implementation groups all the exceptions thrown by the system in a single class and produces their respective error messages. It allows for an easier work flow when treating exceptions. The global handling was chosen because most of the exceptions happen in more than one endpoint and would produce a lot of repeated code if handled locally.

As an example, the flow of a correct HTTP POST request to the routes resource that arrives on the server is the following:

- The request is intercepted by the Logging Interceptor and logs the request information.
- It is then intercepted by the Media Type Interceptor, that checks if the request data format received is supported by the service.

- The Authentication Interceptor checks the user credentials to see if the user can indeed access the service.
- The endpoint is reached in it's mapped Controller, which receives the Route information and that it maps to the correct object. In this case the Route Controller which will then call a service responsible for processing the request data.
- The service, in this case Route Service, will process the data and map it to the correct data type and make a request to the repository to store the received data.
- The repository communicates with the database, to which it sends the data in a database accepted format.
- The database stores the data and returns the identifier of the newly created Route.
- The repository receives the identifier and passes it through to the Route Service.
- The Service passes the received information to the Route Controller.
- The Route Controller builds the newly created route resource URI with the received Route identifier and maps it to the header Location of the response and returns.

Database Management System

The database management system [4] (DBMS) chosen was PostgreSQL[13], using the hybrid functionality of storing valid JSON[10] directly in a table field. The database is used to store all entities required for the service to function and to deliver them to repositories in need. The decision of choosing JSON as a type to store data comes with the need of storing large sets of coordinates belonging to a single entity, this will allow us to make faster and easier calculations of times and distances between routes and points rather than if a point was it's own database entity.

Conceptual Model

The database entity diagram is shown in figure 4.2.

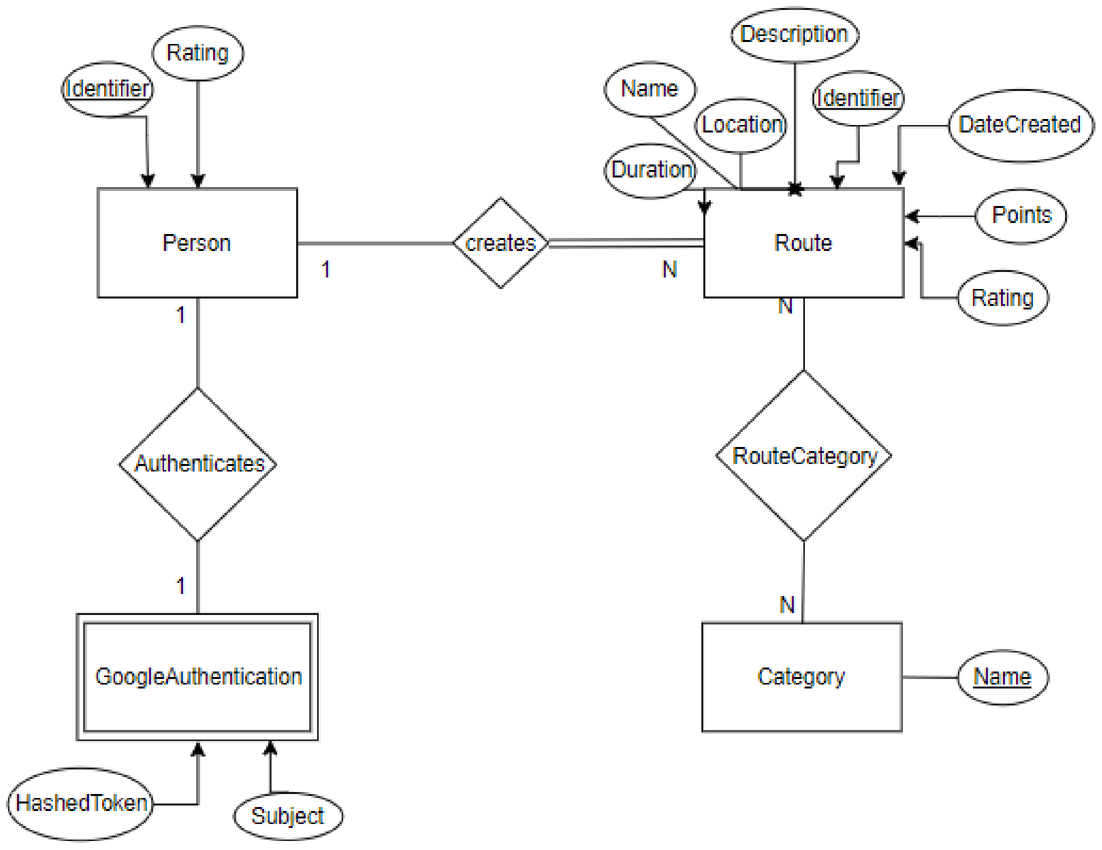

Figure 4.2: Entity Relationship Diagram.

Person

The entity Person represents a single user in the database. A Person can create multiple Routes and have a single form of Google Authentication.

Route

Entity that stores every information required to hold a Route. Special consideration was taken into the making of this entity as a Route must have large sets of coordinates, representing the path that it undergoes. The use of json as a table field Points provided more customization and a more efficient way to make calculations of distances between routes and points.

Category

Each route must be assigned at least one category, and considering a single category can have multiple routes the relation between a Route and a Category must be N to N.

GoogleAuthentication

This entity is used to store authentication metadata regarding each database user. In the future there will be multiple tables with different forms of authentication, hence the name GoogleAuthentication of the entity. It provides scalability to further augment the authentication process, which can be done with a new table FacebookAuthentication for example.

Physical Model

The physical model can be seen in a detailed form in the DBMS documentation.[19]

5 Tests

6 Conclusion

timeline evaluation here

Bibliography

[1] *application/json documentation.* URL: https://tools.ietf.org/html/rfc4627.

[2] *Client Application Documentation.* URL: https://github.com/baltasarb/social-routing/wiki/Client-Application.

[3] *Content Type documentation.* URL: https://tools.ietf.org/html/rfc7231#section-3.2.

[4] *DBMS definition.* URL: https://en.wikipedia.org/wiki/Database#Database_management_system. (accessed: 29.04.2019).

[5] R. Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1.* URL: https://tools.ietf.org/html/rfc2616?spm=5176.doc32013.2.3.Aimyd7.

[6] *HTTP POST request documentation.* URL: https://tools.ietf.org/html/rfc2616#section-9.5.

[7] *JDBC driver definition.* URL: https://en.wikipedia.org/wiki/JDBC_driver. (accessed: 29.04.2019).

[8] *JDBI Documentation.* URL: http://jdbi.org/. (accessed: 29.04.2019).

[9] *Json Documentation.* URL: https://www.json.org. (accessed: 29.04.2019).

[10] *JSON PostgreSQL Documentation.* URL: https://www.postgresql.org/docs/9.3/functions-json.html. (accessed: 29.04.2019).

[11] *Log4J documentation.* URL: https://logging.apache.org/log4j/2.x/.

[12] M. Nottingham, Akamai, and E. Wilde. *Json+problem Documentation.* URL: https://tools.ietf.org/html/rfc7807. (accessed: 29.04.2019).

[13] *PostgreSQL Documentation.* URL: https://www.postgresql.org/files/documentation/pdf/11/postgresql-11-A4.pdf. (accessed: 29.04.2019).

[14] *Social Routing API - HTTP Verbs Documentation.* URL: https://github.com/baltasarb/social-routing/wiki/Social-Routing-API#http-verbs. (accessed: 29.04.2019).

[15] *Social Routing API Documentation.* URL: https://github.com/baltasarb/social-routing/wiki/Social-Routing-API. (accessed: 29.04.2019).

[16] *Social Routing API POST documentation.* URL: https://github.com/baltasarb/social-routing/wiki/Social-Routing-API#post-example.

[17] *Social Routing API PUT documentation.* URL: https://github.com/baltasarb/social-routing/wiki/Social-Routing-API#put-example.

[18] *Social Routing API Schema Documentation.* URL: https://github.com/baltasarb/social-routing/wiki/Social-Routing-API#schema.

[19] *Social Routing Service's DBMS documentation.* URL: https://github.com/baltasarb/social-routing/wiki/database-management-system.

[20] *Spring Exception handling documentation.* URL: https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/servlet/mvc/method/annotation/ResponseEntityExceptionHandler.html.

[21] *Spring interceptor documentation.* URL: https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/servlet/HandlerInterceptor.html.

[22] *Spring MVC Documentation.* URL: https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html.