Academic year 2017/2018

Service engineering

Final Report

TeamUp

In http://blog.edmentum.com/sites/blog.edmentum.com/files/images/Team_Up_RGB.png

Project made by:

Arménio Baptista	68275
Fábio Silva	73786
Miguel Oliveira	72638
Rui Oliveira	73179

Professor:
João Paulo Barraca

Index

Introduction

TeamUp came about when, at the beginning of the semester, we felt a problem in scheduling meetings between friends to play football. We move the problem to the service engineering curricular unit with the main objective being to design an application to facilitate the marking of events of various types including trying to solve all the problems that derive from the scheduling of events between people.
Our proposal was based on the construction of a mobile application where the functionality is specific to each service, following a service oriented architecture, pointing to a total modularity between them excluding any dependency. The proposal has undergone some changes according to the new concepts addressed in the theoretical part.
This report follow a structure based on the presentation done on the last class, where we presented our architecture, the main features of each service, the external APIs used. We also added some considerations about the system and the project, and the respective conclusions.

System features

- Easy and efficient communication between the participants;
- Easy searching of activities by distance, activity and zone.
- Weather information according to the event schedule.
- Maximum and minimum number of participants to ensure the success of an activity/event;
- Public and private events;
- Easy registration and login with Facebook;
- Schedule and location voting process;

Architecture

Service-oriented architecture (SOA) is a design paradigm intended for the creation of solution logic units that are individually shaped so that they can be collectively and repeatedly utilized in support of the realization of the specific strategic goals.
So initially, our biggest problem has been to adopt this new concept and try to avoid as much as possible the dependency between the defined services.
But, we got our architecture to follow the service-orientation design principles, where the access to every services is provided using a prescribed interface with define policies, in order to achieve loose coupling. For each service we define one database that does not have any dependencies with as remaining.
The Application Server in our implementation works like an Orchestrator to provides an abstraction layer on the implementation of a communication system between mutually interacting software applications in a service oriented architecture. Also works like an Enterprise Service Bus (ESB), between the mobile applications and the enterprise services.
All services communicate with the Application Server over HTTP. In case of databases, each one is accessed over TCP by the respective service.
So, with this implementation, we achieved the following points: loose coupling, abstraction and reusability.

Project Structure and Considerations

The project structure is the following:

- AppServer/
- Services/
- docs/
- docker-compose.yaml
- trigger.py

The folder AppServer contains all the files directly related to the Application Server. It also contains a file with example requests that were done during the development process.

The folder Services contains a couple of folders with every Service that is present on our architecture.

All the services have a configuration file, with the extension ".ini", containing some configurations that can be changed in order to the service context.

The services also have a list of requirements for each one - requirements.txt.

When the services execution start, they produce a log file, with their name + ".log", containing all the requests and logging messages.

We also provide a set of calls for each service inside their folder with the name: "api_calls.py". They won't work without adding the access_token field on each request.

The requests examples related to the Application Server is inside its folder with the name "trigger.py".

The trigger file, on the root folder, is a script that should run just after all the services are running so that everyone authenticates and exchanges messages for authorization.

Technologies used

- XMPP;
- SMS;
- OAuth 2.0;
- Docker;
- Message broker;
- JWT;
- Flask;
- Ionic;
- Sphinx.

External Services

- Facebook;
- Google Maps;
- Plivo;
- Open Weather;
- Underground Weather;
- Firebase.

Internal Services Description

All the services have context insofar as the information of one client is not accessible to others. The approach used is based on the request identity. On our context the Application Server is the entity that communicates with each service, and all the information is related to and internal identifier that refers to the Application Server. If a request is done by other entity will not receive the information related to the Application Server ,so there is a separation of content.
In this section is presented a brief description of all services and its main features. A full description can be found in the official documentation localized in the path "docs/[service]/build_/html/index.html" in TeamUp repository.

Presence Manager

Presence Manager is a service that manage the presence of users in a XMPP chatroom. This service use use SleekXMPP library as xmpp client and supports Openfire XMPP server
Communication between client and service is done through a message broker, Mosquitto, that implements the MQTT protocol versions 3.1 and 3.1.1. MQTT provides a lightweight method of carrying out messaging using a publish/subscribe model.
This service allow to get the users' presence in a specific chatroom.

Chat Manager

ChatManager service is a RestFul API that manage XMPP chatrooms using Openfire XMPP server. This service use SleekXMPP library as xmpp client. This service manage chatrooms and users, set roles and affiliations to users and allow to send message to a chatroom.

SMS Manager

SMS Manager service is a RestFul API that sends SMS using PLIVO's API. Communication is done over HTTP using its methods. This service allow to send a SMS to users.

User Profiling

UserProfiling service is a Restful API that manage user's information. This service allow to save, update and delete user's profile and get their profile information.

Location Manager

Locations Manager API is a service that manage locations associated to an event.
This service allows the voting process on the required location for the event and have methods to get the three most voted and the most voted locations.

Weather

Weather Proxy API is a service that provides a 10 day weather forecast per location and a precise temperature for a given day and geographic coordinates.
Our service works as proxy that communicate with two microservices and gives the mean of temperature of this two microservices, to reach more precise values. This microservices are: OpenWeatherMap API and Wunderground API.
In each hour the service checks the current temperature and update if it is greater or less than the defined limit and trigger an "alarm".

OpenWeather

OpenWeather API is a service that provides a 10 day weather forecast per location and a precise temperature for a given day and geographic coordinates. This service communicates with the external API, OpenWeather.
This service plus the UndergrounWeather service compose the Weather service.

UndergroundWeather

UndergroundWeather API is a service that provides a 10 day weather forecast per location and a precise temperature for a given day and geographic coordinates. This service communicates with the external API, Underground.
This service plus the OpenWeather service compose the Weather service.

Push Notification

PushNotification service is a RestFul API that manage push notifications. This service allow to send a push notification with a message.

Schedule Manager

Schedule Manager API is a service that manage schedules in a wide range of situations. This service allow to associate a request with schedule and add and delete schedules and requests. Schedule manager also provides the election of a schedule with a voting option.

Alarm Manager

Alarm Manager API is a service that has a main objective: to trigger alarms according to our context. It can trigger alarms related to:

- The decision datetime
    - Each event has a final datetime where the voting process for the schedules and locations should end. At this time the Alarm Manager gets the most voted schedule and location, by contacting the two respective services, and send an alarm message to and endpoint that should be specified on the configuration file.
- The temporal approximation of the events
    - For each event is triggered two alarms:
        - One alarm, one hour before the most voted datetime with the purpose of remembering the occurrence.
        - One alarm, when the event starts.
- The climatic conditions:
    - Rapid changes in temperature are also an alarm.

Facebook

Facebook service is a RestFul API that manage user's information related with the usage of facebook as identity provider. It also validates the OAuth tokens provided or not by the Facebook. In relation to the users, the service provides its friends list for the ones that have an account on the facebook developers and maintain a register of the users.

Geolocation

Geolocation service is a Restful API that gets the latitude and longitude on a predefined parameters. It also provides a couple of methods related to the coordinates processing, geocode and lookup
This service also provides and maintain a history of the user's position .

Authentication Manager Service

Authentication Manager service is a Restful API that has a main objective: provide authentication and methods to confirm it. It also provides an interface for the registration of new services available only for administrators of the service.

Events Manager

Events Manager service is a RestFul API that manage events in a wide range of situations. The main features are related to the events and the participants of each event.

Authentication and Authorization

All the services of our context are authenticated with the exception of Presence Manager Service. We implemented a Authentication Service where all the services must be registered. The authentication project is quite simple and should be done in 2 steps/requests, by the following order:

- POST
    - This requests aims to initialize the authentication process by displaying its service name, which is registered in the authentication service, and should receive a nonce.
- GET
    - This requests should contain the digest of the nonce concatenated with the digest of the respective service password. If they match with the credentials that are present on the authentication server, the service will

receive a Json Web Token(JWT), which must be used to provide its identity over surrounding services.

With a challenge-response approach we can authenticate all the services. The digest function is SH256. More information about this process can be found on every service documentation, and the full description of the authentication service is also provided

With regard to authorization, only the services that communicate with others need to get an access token with the necessary permissions. So, for each service, the services with which it communicates are:

Service	Communicates with
Alarm Manager Service	Schedule Manager Service Location Manager Service Weather Service
Weather Service	Underground Weather Open Weather
Application Server	Facebook Service Geolocation Service User Profiling Service Events Manager Service Schedule Manager Service Location Manager Service Chat Manager Service Push Notifications Service Alarm Manager Service SMS Manager Service Weather Service

The left Services need to get an authorization token for each Service on the right in order to establish a successful communication with them.

Each service provide authorization using OAuth 2.0.

The authorization process, like the authentication, is quite simple and can be done in 3 steps:

- Create an app on the service where we want to get authorization access by registering an url where the grant token should be redirected and the desired scopes. This request produce two field: a client id and a client secret that will be user on the following request.
- Send a GET request containing the client id, secret to obtain a temporary grant token(1 min of lifetime).
- Exchange the grant token by an OAuth 2.0 token with the respective permissions.

The Json Web Token must be sent in the first two requests in order to confirm the identity of the service requester.

More information about this process can be found on every service documentation.

Application Server

Application Server provides an abstraction layer on the implementation of a communication system between mutually interacting software applications in a service oriented architecture (SOA). It should not be treated as a service. Although providing an interface with some methods should be considered as a bus that distribute information between services.
The main capabilities of this component are:

- Routing;
- Message Transformation/Message Processing;
- Process Choreography;
- Protocol Transformation;
- Security.

Basically acts like a mediator, an Enterprise Service Bus(ESB), between the mobile applications and the enterprise services. The communication on the left side, that is on the input between the mobile applications and the Application Server is always is done over HTTP using POST methods. Request is formatted as JSON, and the content type is application/json. On the enterprise side, the communication is mainly done over HTTP, however in some cases it uses mqtt protocol and a message broker - Mosquitto.
Currently is in the version 1.0.

Routing

The ability to channel a request to a particular service provider based on deterministic or variable criteria. For example, the authentication process between the mobile applications and the authentication server is routed by the application server based on requests content. The process is composed by two sequential requests: the first one must be a POST and the last one a GET request. Based on that the Application Server route the requests to the Authentication Service.

Message Transformation/Message Processing

Convert the structure and format of the incoming business service request to the structure and format expected by the service provider. For example, given a request to vote on a schedule for a specific event the Application Server must receive two parameters: the event identifier(event_id) and a schedule available for the respective event(schedule). The message that the schedule manager service expects is a little bit different and the necessary modifications are done by the Application Server in order to respond successfully to the resolution of the various requests.

Process Choreography

Manage complex business process that requires the coordination of multiple business service to fulfil a single business service request. For example given a request to create

an event there are a couple of requests that need to be done over multiple services in order to complete all the sub processes triggered by the application server, where the input was the user action.

The processes/services are:

- **create event** - Events Manager Service;
- **create chat room** - Chat Manager Service;
- **look up coordinates** - Geolocation Service;
- **create datetime schedule** - Schedule Manager Service;
- **create event** - Location Manager Service;
- **add each event schedule** - Schedule Manager Service;
- **add each event location** - Location Manager Service;

Protocol Transformation

Accept one type of protocol from the consumer as input (i.e.SOAP/JMS) and communicate to the service provider through a different protocol (i.e. IIOP). For example, given a notification requests related to temporal approximation of an event, the input request is done over HTTP and the appserver process the communication to the Presence Manager Service over MQTT.

Security

Protect enterprise services from unauthorized access. In SOA there are no more silos, services become visible to the entire enterprise through ESB. ESB should access a security manager or authentication and authorization rather than have the direct responsibility. In relation to the Authentication the Application Server is authenticated over a Authentication Service by a process that contains two requests: the first one is a POST and the last one is a GET. As above the process is the same as in mobile applications. Every request to the Application Server must contain a valid Json Web Token(JWT) to provide authentication of the entity and allow to the esb confirm it close to the Authentication Service providing also its JWT. Concerning to the authorization, the application server acts like a client to the enterprise services. Before you can receive commands, the application server must obtain an authorization token for all services it communicates with, except for the authentication service and the presence manager. The authorization is completed in 3 steps: the first concerns the registration of a redirect url, to receive the grant, and the desired scopes. The second and the third are related to obtaining the grant and the OAuth 2.0 token respectively. The first and second step must contain an JWT-Bearer to provide authentication and check it over the Authentication Service and only provide access to the service if the client is authenticated.

Application Server Interface

The application server exports multiple methods according to the operations around the context. The structure is simple and follows the next rule: given a base url - http://127.0.0.1:5012/proxy/ - and a desired operation - i.e: login - the final url to materialize is the concatenation of both: http://127.0.0.1:5012/proxy/login. Hereupon, the methods implamented and the respective url are:

- **login** - http://127.0.0.1:5012/proxy/login
- **create event** - http://127.0.0.1:5012/proxy/create_event;
- **update event** - http://127.0.0.1:5012/proxy/update_event;
- **update user** - http://127.0.0.1:5012/proxy/update_user;
- **search event** - http://127.0.0.1:5012/proxy/search;
- **add user** - http://127.0.0.1:5012/proxy/add_user;
- **add schedules** - http://127.0.0.1:5012/proxy/add_schedule;
- **add location** - http://127.0.0.1:5012/proxy/add_location;
- **vote datetime** - http://127.0.0.1:5012/proxy/vote_datetime;
- **vote local** - http://127.0.0.1:5012/proxy/vote_local;
- **get groups** - http://127.0.0.1:5012/proxy/get_group;
- **get groups** - http://127.0.0.1:5012/proxy/get_groups;
- **notifications manager** - http://127.0.0.1:5012/proxy/notifications_manager;
- **invite friends** - http://127.0.0.1:5012/proxy/invite_friends;
- **leave group** - http://127.0.0.1:5012/proxy/leave_group;
- **delete group** - http://127.0.0.1:5012/proxy/delete_group;
- **authentication app** - http://127.0.0.1:5012/proxy/authentication_app.

Application Server Services

All services are registered in the application server and it establishes communication with them for a certain purpose described by each method of the receiving service. The services are:

- Alarm Manager Service;
- Authentication Service ;
- Chat Manager Service;
- Events Manager Service;
- Facebook Service;
- Geolocation Service;
- Location Manager Service;
- Presence Manager Service;
- Push Notifications Service;
- Schedule Manager Service;
- Sms Service;
- User Profiling Service;
- Weather Service.

Docker Process

In order to deploy our server side application, it was used the Docker technology that allows to create several containers, each one running his own environment.
In this project, each service has its own container and its database, if needed. In each service repository, there is a 'Dockerfile' file with the building instructions for each service.

- **Docker databases**

The containers that deploy the databases use the 'mysql' registry from the Docker Hub repository with the 'latest' tag. For each database, the 'init.sql' file present in the service directory is copied to the container into the '/docker-entrypoint-initdb.d' file, which allows the container, when started, to fetch the SQL queries present in the file and create the database based on this queries.

- **Docker services**

The containers that deploy the services use the 'ubuntu' registry from the Docker Hub repository with the '17.10' tag. At the build phase of the container, the dependencies needed to run the service are installed, p.e. the python3 package. Then, the directory and the files of the service are copied into the root of the container and the python dependencies of the service present in the 'requirements.txt' file are installed. Lastly, the entrypoint of the container is configured to the python file that starts the service, which makes the container run the service when it starts..
After this steps, the container is ready to run.

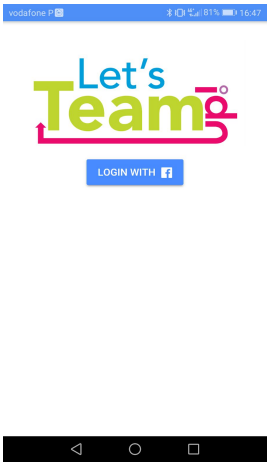
To deploy all the services and databases, it was created a 'docker-compose.yaml' file that contains all the services to start. Each service has the instructions to run the service, p.e. the path to the 'Dockerfile' with the build instructions. These instructions contain the Docker image to use, the port-mapping from inside the container to the outside, the internal IP of the Docker network, some initial variables (in the database's services) and also health checks (in order to keep the container running and to restart it in case of failure).
It is also created an internal network in the Docker environment to allow the communication between containers.

Technologies used

- XMPP;
- SMS;
- OAuth 2.0;
- Docker;
- Message broker;
- JWT;
- Flask;
- Ionic;
- Sphinx.

Application Instruction Manual

The application will start at the login page where the user will have to authenticate with the facebook account. **After the first login, its necessary to remove loginIdp from facebook Apps**. Once logged in, the user is referred to the home page.

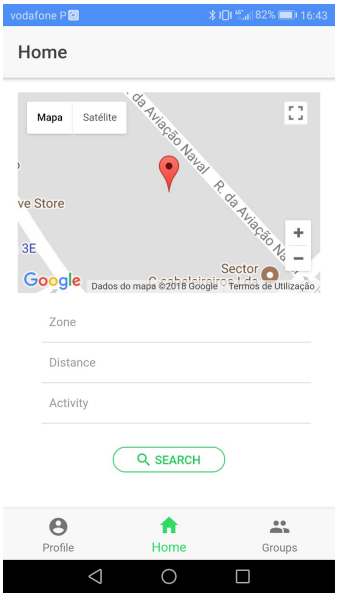


Home page

On this page, the application will access the current location of the user as well as all events that are around him within a radius of less than or equal to 5 km. Both this locations activities will be display on the map. The location of the user is represented with a red mark, while the events that surround it are defined with a flag-shaped mark.
The user can also easily search events by zone, distance and activity. After searching, all the events related to the search in question will appear, and the user can see the name, type of activity and description of each event by clicking on the marks on the map.

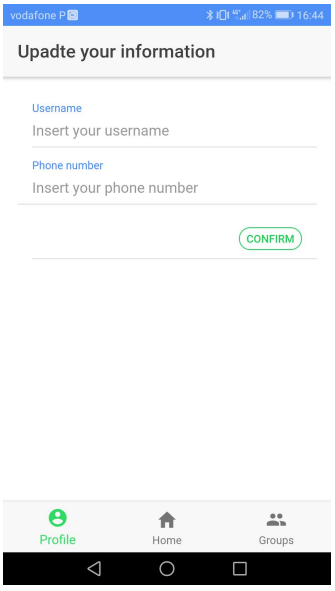
Also, in this click the user have the possibility to begin the process to join a group, by clicking on “join group button” (user than go to the GroupInfo Page).

Note: On this page the refresh mechanism has been implemented to update the page with the latest results. Just pull the page down (usual refresh movement).



Update Profile page

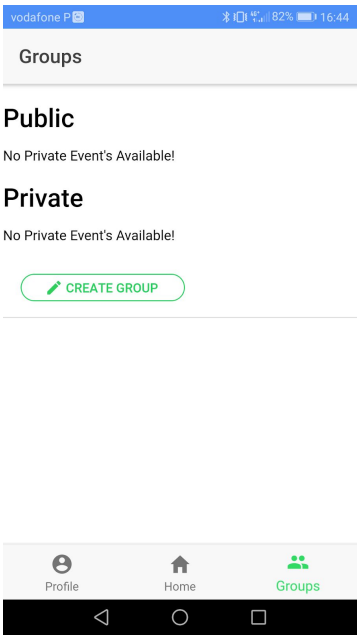
This page is accessed when the button on the left side of the bottom tab is pressed. This allows the user to define his / her username and enter his / her phone number.
in case of success will pop up a confirmation popup and the user will be later forwarded to the home page.



Groups Page

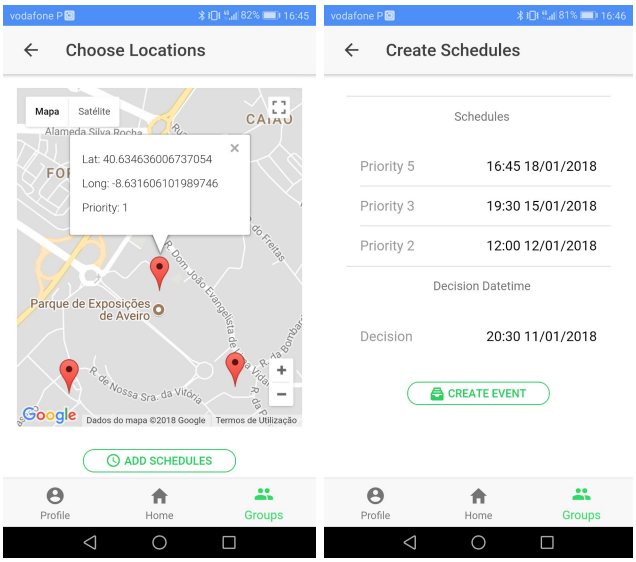
This page will show the list of the private and public groups that user has access to. In each item group there is a button to check the information of this group in question. On click, the user then go to the Group Information Page.
Also, on this page the user can begin the create group process after clicking on “Create Event button”, that will show the CreateEvent Page.

Note: On this page the refresh mechanism has been implemented to update the page with the latest results. Just pull the page down (usual refresh movement).



CreateEvent Page

The group creation process is composed of a sequence of 3 pages. In the first, the user defines the name, activity and description of the event, choose whether the event is public or private and the minimum and maximum number of participants. Than in next page, the user will pick the locations for the event he / she can choose up to 3 locations, in which the first will be the highest priority and the last selected the least priority. The pick location process is done by clicking on the desired location on the displayed map. Lastly, on the last page, the user will define the schedule for each location, and choose the decision datetime.
If the event creation was successful, than the user will redirect to the HomePage.



Group Page

On this page the user can see the existing locations and schedules for an event and, in the other hand, can vote on the location and / or schedule that want for the event in question.
Also, in this page the user can leave the group (pressing on Exit group button) or remove the event (pressing on Remove Event button). After click in booths this buttons the user goes to the Home Page after confirm the operation in pop up that will appear in case of success.

GroupInfo Page

This page is responsible for showing the weather forecast (temperature as well as the atmospheric condition) for each schedule related to the event. At the bottom, exist a button that on click, adds the user to the group, that is, the user will join group, and the app will redirect the user to the Group Page associated tho the event.

Project Limitations

The project as a whole has some limitations related to several types. There are some more problematic than others, but we show here the ones that are known:

- Limitation on the use of external APIs to consult the weather forecast since it has a maximum forecast of 10 days and consultations with a difference of more than 10 days will trigger an error;
- The services were developed to be authenticated only with the help of jwt;
- In the request to exchange the grant by the OAuth token the request is not authenticated by limitations of the library used;

- The project has a central point of failure, in case the authentication service fails any requests will be executed successfully;
- Authorization policies are based on a String.
- There is no refresh of OAuth tokens;
- The presence manager service is neither authenticated not authorized;
- The docker compose does not have volumes, so, if the service container stops, the logging file will be lost and all the information created on runtime.
- In the app, the users cannot remove other users in a group;
- In the app, leave group button exists in the app of all elements of the group but the group only can be removed by the owner of the group;
- In the app, there are some limitations related to the usability;

Future improvements

- Change requests to save cookies instead of always sending;
- Implement XACML;
- Refresh the token;
- Modularize the authentication method;
- Change the authorization (Library);
- Improve the docker deployment;
- Develop a authorization, authentication process for the presence manager service;
- Improve the app and the Application Server or even use an ESB.

Conclusion

After finishing this project we extend our knowledge in Service oriented architecture.
We think that we design a total system supported by a service oriented architecture in which the functionalities of the mobile application are made available by each service and these are connected by a Bus (Application Server). The request-reply paradigm is present on almost every service and is the main pattern of communication. Following this principles we got a flexible and adaptable system, where our services can be incorporated on different contexts and we can swap ours by others. Thus, there is a total interoperability between services, where each one represents its business value by what it returns and it is a black box for the services that communicate with it.
The mobile application as final "product", does not show the capability and complexity of our system in terms elaboration of services and the connection between them. There are some features implemented on each service that the app doesn't know. So, our mobile app should be the first component improved in order to show more and better functionalities providing a better user experience.