Daytrip: a service-oriented web application for inspiring daytrips in Italy

Service Design and Engineering course - Final Project

Leonardo Vicentini, Luca Vian

github.com/vicentinileonardo/daytrip

20 June 2023

1 Introduction

The project topic was chosen based on the fact that the team deemed interesting to solve a simple yet **potentially very expandable** web application with a service-oriented approach. The concept is pretty straightforward, the web app suggest a **ranked list** of daytrip destinations, reachable with the time at disposal of the user, based on some indicators (like weather, air pollution, etc.). The project incorporates both **internal** (e.g. users and destinations data) and **external** sources of data (e.g. ranges, weather, email validator, etc.).

The service-oriented paradigm enabled the team to add ranking indicators in an iterative way during the entire development of the project. The **potential expansion** of the ranking system (with new indicators) is very easy thanks to the adoption of the service-oriented paradigm. The service-oriented approach allowed the team, with **different skills** and backgrounds, to work efficiently to achieve the final goal, implementing services both in **Python** (**Flask**) and in **JavaScript** (**Node.js**).

Another important concept is the one of **separation of concerns**: services are decoupled components so it is possible to modify a service without the worry of breaking another one. If a change on the codebase went wrong, leading to errors, only an isolated service would be affected and it would be easier to debug.

A goal set by the team before the development of the project was that service implementations should have been **general**, **reusable** in different scenarios (not attached to the domain specificity) and scalable.

2 Architecture

The application is designed with a 4-tier architecture made of: **Data layer**, **Adapter layer**, **Business Logic layer** and **Process Centric layer**. The total number of services used or created is **25**: 8 in the Data Layer, 9 in the Adapter layer, 5 in the Business Logic Layer, 3 in the Process Centric Layer.

Docker Compose were leveraged to define and deploy the **multi-container** application introduced above with ease. With a single file it has been possible to define and manage the entire architecture of services. Although Docker Compose it is not the industry standard for production due to some limitations (e.g., downtime during deployment, only single host), it is perfect for the scope of this software project.

The diagram of the architecture can be seen in figure 1.

2.1 Endpoints

REST (REpresentational State Transfer) architectural style was chosen for the development of the services due to its generality, simplicity, and style that was suitable for representing the resources identified. Indeed, all the endpoints built are able to operate on a single and well defined entity (for example, '/users' or '/geocodes') and also the hierarchical URI specification was followed (for example, in '/destinations/reachable' and in '/destinations/best'), in order to define relations between entities or in order to filter and organize them. The methods considered during the development of the architecture are GET, POST, PUT and DELETE.

Documentation of the endpoints was written leveraging **Apiary** service and the **OpenAPI 3.0.0** specification. The documentation shows different kind of successfully, failing and erroneous requests, focusing on describing their objective and some examples of response given when called. The documentation did not cover all of the errors addressed by the services, but only one per category for each method supported by a certain endpoint. For example, for each endpoint, only one error for a "missing parameter" will be displayed, not covering all possible parameters requested by the method taken into account.

2.2 JSON specification, Error Handling and Testing

The JSend specification was adopted in order to have a standard for defining the JSON responses of the services developed. The responses have a well define **type** (success, fail or error) and **keys** (data, code and message) that can be mandatory or optional depending by the response's type involved.

The errors, having an "error" or "fail" type depending by their nature (as defined in the JSend specification), were managed in order to be self-explanatory on why there were triggered. They could regard both errors caused by **users mistakes** (for example invoked by missing parameters or by calling a not existing endpoint) or by **server failures** (for example, when an external service is down). The errors returned will be propagated all among the hierarchy of calls established by the Business Logic Layer and the Process Centric one.

In order to perform testing for all the endpoints during their development, **Postman** tool was leveraged. **MongoDB Compass** was used for testing the internal databases. Some **flow testing** has been performed with **simulated crashes** of

some services, leveraging the Docker Compose GUI. For instance, stopping the Air Pollution API adapter will then led to a ranked destinations list where that indicator is not considered when calculating the destinations rating.

2.3 Data Layer

MongoDB is used as internal databases for data related to Users and Destinations. CRUD operations on these NoSQL databases are leveraging the Mongoose library for Node.js. The external data sources used are 8 (two of which are used directly by the presentation layer): **ip-api** and **ipify API** for dealing with the IP address of the user (in presentation layer), **EVA API** for the email validation, **TomTom API**, **OpenStreetMap API** and **Google Maps** for dealing with coordinates, **Weather API** and **OpenWeatherMap API** for weather data.

2.3.1 TomTom APIs

Two external APIs provided by TomTom were used.

The first one was used in order to **calculate the Reachable Range**, returned as a boundary polygon, specifying an origin and some parameters regarding the resources that a user has (e.g. time, distance, vehicle). The related adapter '/ranges' will call the original API using only three arguments (lat, lon and timeBudgetInSec). It will be used in order to filter only the destinations that can be reached by the users, avoiding to propose the other ones.

The second one was used in order to **compute indirectly the crowdedness of a destination**. This was possible by comparing the currentSpeed and the freeFlowSpeed, given a couple of coordinates identifying a street. We have assumed that the most those speed's indicators are similar, the least crowded a place will be. The endpoint of its implemented adapter is '/crowds'. This indication, when retrievable (so, when the origin given is not isolated and has a street in its proximity), is used in a weighted way in order to compute an approval rating for the destinations filtered by '/ranges'.

2.3.2 Weather APIs

Data used for the weather-related indicators, which have the **largest share** in forming the ranking, are pulled from external web services: **Weather API** for dealing with temperature, precipitations and wind; **OpenWeatherMap API** for air quality.

2.4 Adapter Layer

The Adapter layer is mainly used for **standardizing** the data coming from the external APIs. Essentially, each service in this layer corresponds to a service in the data layer. It is used to map the data coming from the external APIs to the data model used in the application. For example, to make all the data sources **compliant** with the JSend specification of responses, some encapsulation or tuning is needed.

This layer is also helpful for adding sensitive information such as **API keys** to the external API requests.

2.5 Business Logic Layer

The Business Logic layer is where calculation and elaboration of data happens. This layer consists of: Valid Email Service, Coordinates Service, Reachable Destinations Service, Coordinates Rating Service, Boundary Service.

As an example, the **Coordinates Rating Service** performs calculations based on underlying data to provide a rating for the coordinates provided. If some underlying services are not available (e.g. crashed) the rating is calculated using only the available services and the weights are normalized accordingly.

2.5.1 Reachable destinations service and Boundary service

Initially, some reproducibility problems were encountered in the deployment of the Reachable Destination Service which leveraged directly the **Matplotlib** Python library to check if a geospatial point lied inside the boundary of a reachable range. In order to install it on a Docker container, its Dockerfile increase a lot in complexity (compared to the project's standard) while its deployment became more problematic with some specific system architectures.

This situation led to the design and implementation of a **Boundary service**, deployed **outside** the Docker environment. In particular **Amazon Web Services** free tier were leveraged using a **AWS Lambda Function** (Serverless Function as-a-Service) and **AWS API Gateway** with **Lambda Proxy Integration**.

As a result of this design decision, the deployment of the entire architecture has become smoother because the sole criticality discovered has been fixed by creating a service, leveraging a powerful abstraction paradigm (FaaS), that is entirely managed by a cloud provider.

2.6 Process Centric Layer

The Process Centric Layer provides high level functionalities that are the only ones that serves directly the users' requests. The services belonging to this layer are three:

- **Best Destination Service**: has the role, given an origin, a timeBudgetInHour, and a date, of ranking the destinations reachable from the origin in the given time based on the information retrieved by the TomTom and Weather APIs. The service has the role of **Orchestrator**, involving and combining the results given by the Reachable Destination, the coordinates Service and the coordinates Rating Service.

- **User Login Service**: **Orchestrator** that directly involves the User DB Adapter in order to verify if the credentials provided by the users are coherent with the profile of an already registered one. Doing that it allows to create or update a signed JWT.
- **User Registration Service**: has the role of **Orchestrator** that checks and manipulates the values that it receives during the registration of an user. Indeed, it involves the Valid Mail Service, the Coordinates Service and, if those calls are successfully performed, it creates the users with a POST on the User DB Adapter.

2.7 Presentation Layer

To make the utilization of the underlying services practical, a simple yet effective **frontend interface** was designed. In particular, **Bootstrap** framework was leveraged to build the three pages: main page, login, registration. As mentioned before, two external services are directly called from this layer: an IP address retrieval service and Google Maps.

NGINX is mainly used to serve the static files (html, css, js, images). Moreover, NGINX **reverse proxy** capability is leveraged to route the requests to the appropriate service only for request external to docker environment. It should be noted that services talk to each other using the **default docker network** which is created when using a docker-compose file.

2.8 Authentication and Authorization

A basic authentication is provided by generating a signed JWT by passing a valid (email,password) couple at the endpoint '/sessions' using the GET method. The token generated contains in the payload the status (standard or admin), the id, the email and the origin's coordinates of the user authenticated. The token, if stored in the header as "Authorization", can be used in order to authorize the access to different routes (PUT in '/sessions' and POST, PUT and DELETE in '/destinations'). The access of those endpoints in '/destinations' also checks the status of the user authenticated, that must be "ADMIN". The token is also used to pre-fill the form on the frontend that invokes the Best Destination Service with its field "origin".

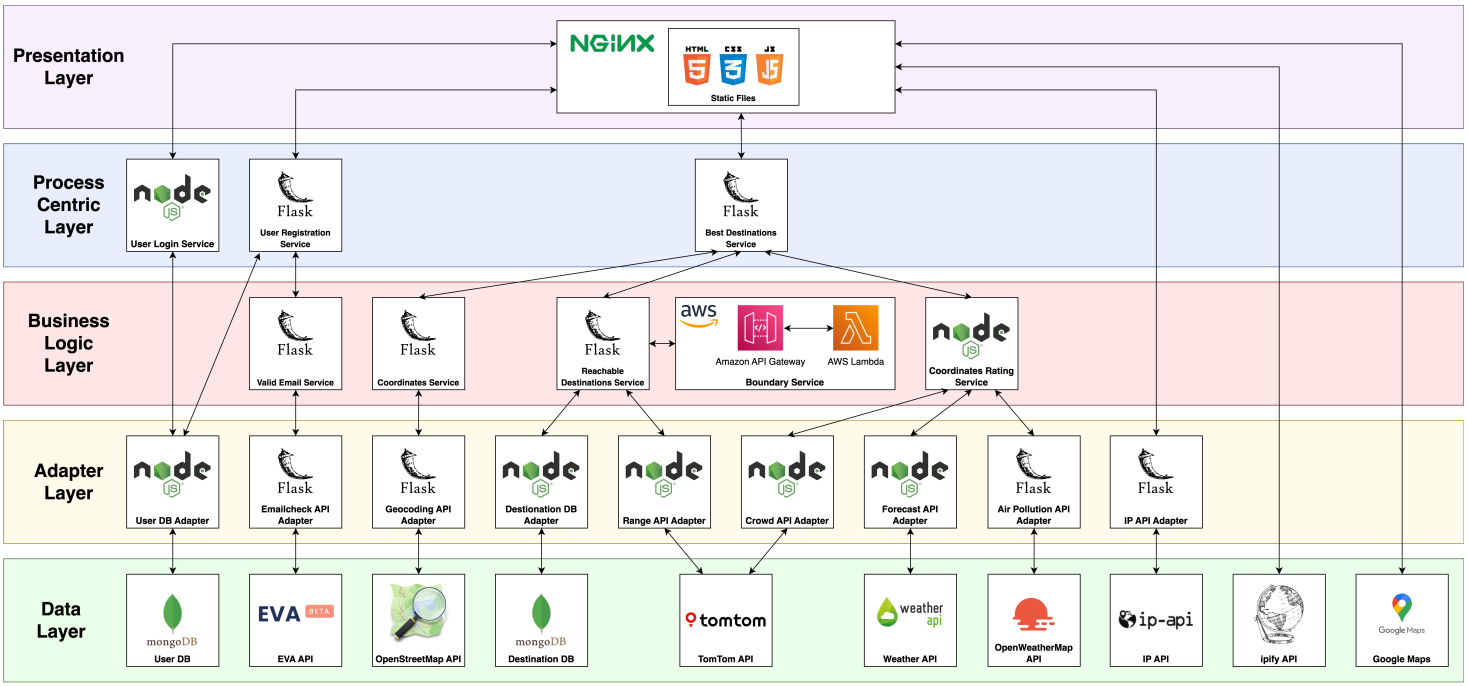

Figure 1: Architecture diagram

3 Conclusion

The scenario considered allowed to successfully use a **service-oriented architecture** in order to accomplish the set goal. Indeed, it was possible to implement the application in a hierarchical structure of services, that are heterogeneous in the purposes and in the programming languages used. The use of Docker Compose was a winning choice since it allowed a fast and smooth configuration and deployment of all the services of the application.

Some further improvements that have been identified rely on the need of modifying the authentication service, by storing the JWT generated in the session and not in the cookies in order to have a better server-side session control.

Several attempts have been made to deploy on the **Microsoft Azure** cloud. In particular, the services used were Azure Container Instances and Azure App Service, as the different version of the Docker Compose configuration file testifies. Unfortunately, despite most of the problems were solved and many hours were spent in troubleshooting, none of the above services led to a working deploy.