RESTaurant Reservation

Leonardo Emili, Alessio Luciani, Emanuele Mercanti, Andrea Trianni

June 4, 2021

Project report *for Cloud Computing course*
Department of Computer Science, Sapienza
Prof. Emiliano Casalicchio

Contents

1 Problem introduction

While there are lots of services to order food deliveries, there is seemingly little choice in the market when it comes to restaurant booking services. The most widespread option to book a restaurant is to use the restaurant's website. This makes it hard to integrate the different "processes" of going to the restaurant, namely searching for where to go, reading the place information (e.g. the menù) and finally booking a table. As things are now one would use Google / Tripadvisor for the search part and restaurants own websites/phone numbers for the booking part. Getting information about the menu is seldom a feature offered by the aforementioned service providers and an actual pain-point for users. Our web app conveys all the different steps into a single, easy-to-use platform where users can search, read the menù and book a restaurant. Additionally, restaurant owners are able to create their own business profile and manage them: updating the menù when needed and accepting the reservations made by users. This last point is particularly important, since restaurants may still receive bookings from different means (phone calls, customers unexpectedly showing up, etc.)

2 Solution design

We started with a system design step. Regarding this, we realized sequence diagrams that describe the sequences of interaction between the simple frontend and the microservice-based backend. These also take into account the communications that happen among different microservices that are involved in the same use case. One example is the restaurant reservation use case that passes through an authentication microservice and another microservice that handles the restaurants' data. Here we describe the microservices that we designed and attach the corresponding sequence diagrams showing the interactions among them.

2.1 Microservices

2.1.1 User auth

The user auth microservice exposes an interface for authentication needs of the user. These are mainly related to login and registration via the frontend web app and identity validation via the backend for operations that need authentication. The main idea is to have an API that responds to requests, by operating on a NoSQL database. The database contains information about the users: Name, Surname, Email address, Password, and Session Tokens. The user registers into the system and a new entry in the database is created. When a user logs in, a new session is created by generating a unique token. This token is stored in the database and sent to the frontend to be cached. The system is purely RESTful and thus stateless. Every request is authenticated via the auth token.
The available operations are:

- Login (Figure 1): The client provides email and password (or token) and receives a session token if the user is registered.

- Registration (Figure 2): The client provides name, surname, email and password, and receives a session token after being registered in the database. The operation fails if a user with the same email already exists.

- Logout: The current login session is canceled.

- Token validation: The calling microservice (the one that needs a request to be authenticated) provides the email and session token of the user that initiated the operation and receives a validation or an error, depending on the validity of the token. This operation is shown in Figure 5.

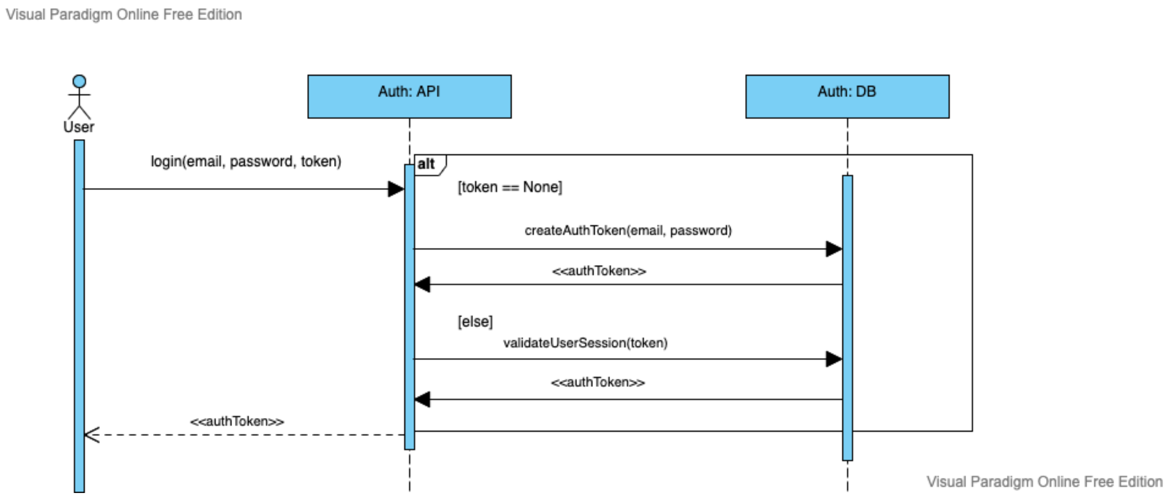

Visual Paradigm Online Free Edition
User
Auth: API
Auth: DB
login(email, password, token)
alt
[token == None]
createAuthToken(email, password)
<<authToken>>
[else]
validateUserSession(token)
<<authToken>>
<<authToken>>
Visual Paradigm Online Free Edition

Figure 1: User login sequence diagram.

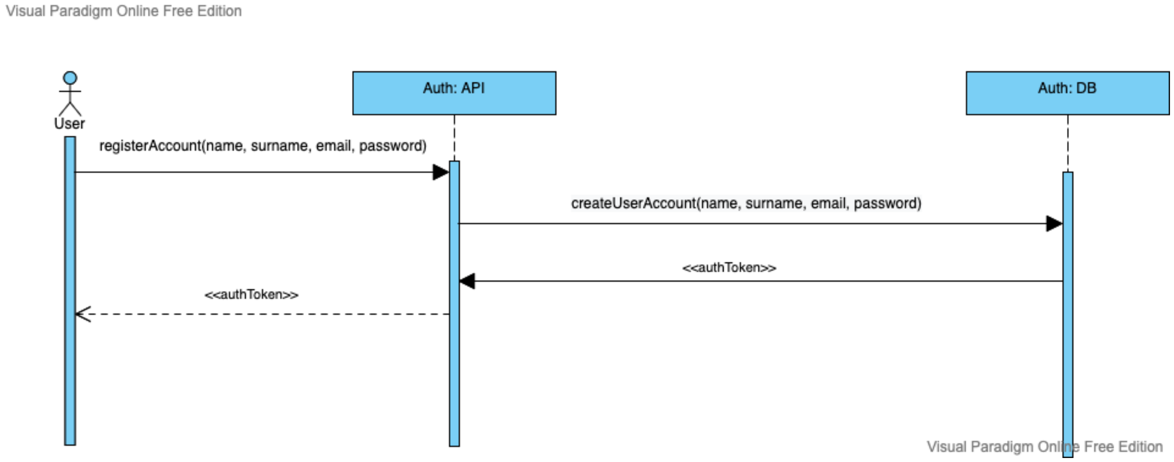

Visual Paradigm Online Free Edition
User
Auth: API
Auth: DB
registerAccount(name, surname, email, password)
createUserAccount(name, surname, email, password)
<<authToken>>
<<authToken>>
Visual Paradigm Online Free Edition

Figure 2: User registration sequence diagram.

2.1.2 Restaurant auth

The restaurant auth microservice exposes an interface for authentication needs of the restaurant. It is very similar to the one of the user, but it is separated to better divide contextes. In fact, the scalability needs of the two authentication services are quite different. After the release of such a system, the increase in the number of users is expected to be much more significant than the one of the number of registered restaurants. This division guarantees individual scalability. Also in this case there is an API that responds to requests, by operating on a NoSQL database. The database contains authentication information about the restaurants: Name, Email address, Password, and Session Tokens. The tokens work the same way they do in the user auth module.
The available operations are:

- Login (Figure 3): The client provides email and password (or token) and receives a session token if the user is registered.
- Logout: The current login session is canceled.
- Token validation: The calling microservice (the one that needs a request to be authenticated) provides the email and session token of the restaurant that initiated the operation and receives a validation or an error, depending on the validity of the token. This operation is shown in Figure 6.

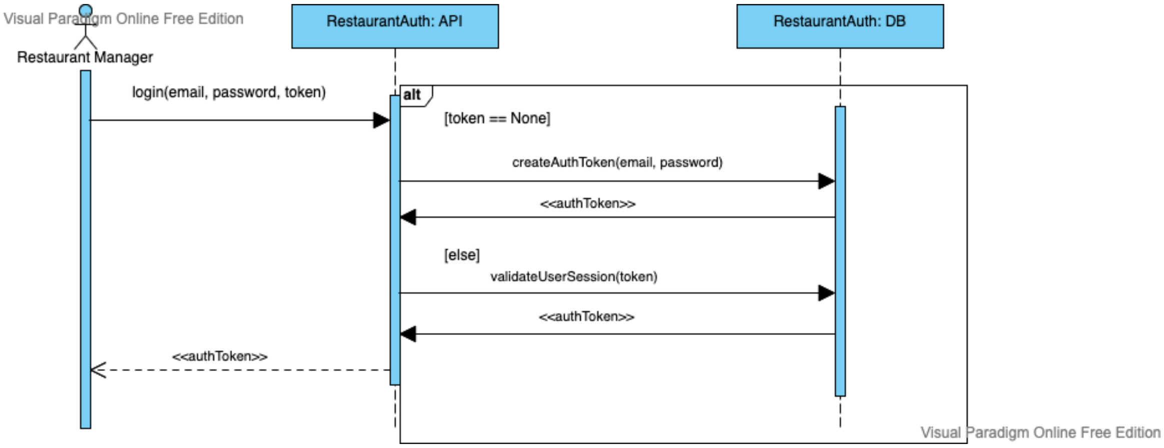

Visual Paradigm Online Free Edition
Restaurant Manager
RestaurantAuth: API
RestaurantAuth: DB
login(email, password, token)
alt
[token == None]
createAuthToken(email, password)
<<authToken>>
[else]
validateUserSession(token)
<<authToken>>
<<authToken>>
Visual Paradigm Online Free Edition

Figure 3: Restaurant login sequence diagram.

2.1.3 Restaurant data

The Restaurant data microservice exposes an interface to search among different restaurants. After the log-in step, it is possible to search and select a restaurant in order to book it.
The search service operates on a NoSQL database containing data about the restaurant (name, address, email, rating, menu). Each restaurant is stored as a JSON object. The user can query the database through the service RESTful API. The sequence diagram is shown in Figure 4.
After the user has selected a restaurant, its unique ID is passed to the booking microservice. Disentangling the search service form the booking service allow us to have separated databases and thus to distribute the workload better. Indeed, this way individual scalability is assured: a user can search among all restaurants, but can book only one at a time. This consideration leads to the necessity to have decoupled services.
There is on possible operation to perform:

- Search: the client provides a search string and receives a list of restaurants satisfying the search criteria;

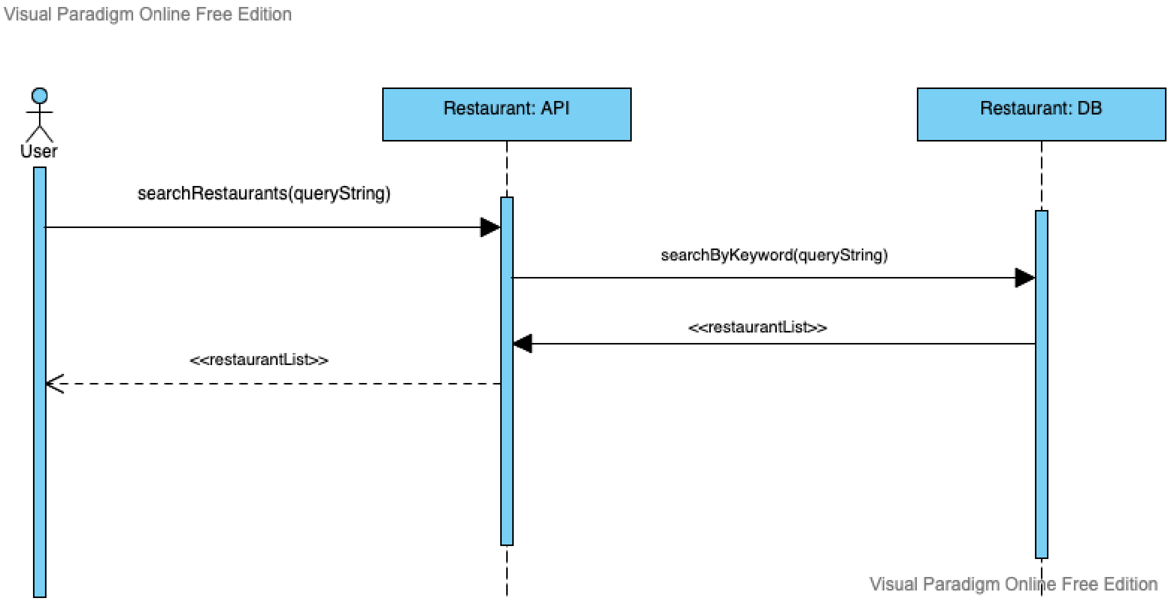

Visual Paradigm Online Free Edition
User
Restaurant: API
Restaurant: DB
searchRestaurants(queryString)
searchByKeyword(queryString)
<<restaurantList>>
<<restaurantList>>
Visual Paradigm Online Free Edition

Figure 4: Restaurant search sequence diagram.

2.1.4 Booking

This microservice offers all the functionalities that are needed to deal with the reservation of the tables in the restaurants. These functionalities can be used both from standard users and restaurant accounts, depending on the specific needs trough a REST API. More in detail users can send a request of table reservation to a specific restaurant, and this last can accept or deny this request. The service also have to offers other methods to retrieve the list of the reservations of a given user/restaurant.
This is one of the most important part of our whole application and also one of the most complex. To successfully complete all the task, this service has to communicate with *restaurant and user auth microservices* to validate the session tokens before commit all the operations.
As like the others, this service relies on a NoSQL database to store reservations records. For each reservation a unique code identifier is generated and also the email of both restaurant and user are saved as contact info and also to link the requests with the accounts. The reservation also require these other fields: *"date", "service", "time", "seats", "notes", "status"*. When a request is sent, its first status is set to *"pending"*. The tree possible status that a request can assume and that the microservice allow are: *"pending","accepted","refused"*.
So, in detail, the available operations are:

- Reserve (Figure 5): A standard user send a request of a reservation to a specific restaurant providing all the required fields plus its session token that has to be validated in order to successfully complete the request.

- Change status (Figure 6): The restaurant wants to change the status of a request, accepting or denying the reservation. To complete this operation the restaurant has to sent the reservation_id, its email and the auth token to do other checks, and the new status. The operation can fail if the new status is not into the set of the allowed status, if the token is not valid, or if the reservation is not associated to that restaurant.

- Retrieve: Each restaurant or standard user has the possibility to retrieve all its reservation record.

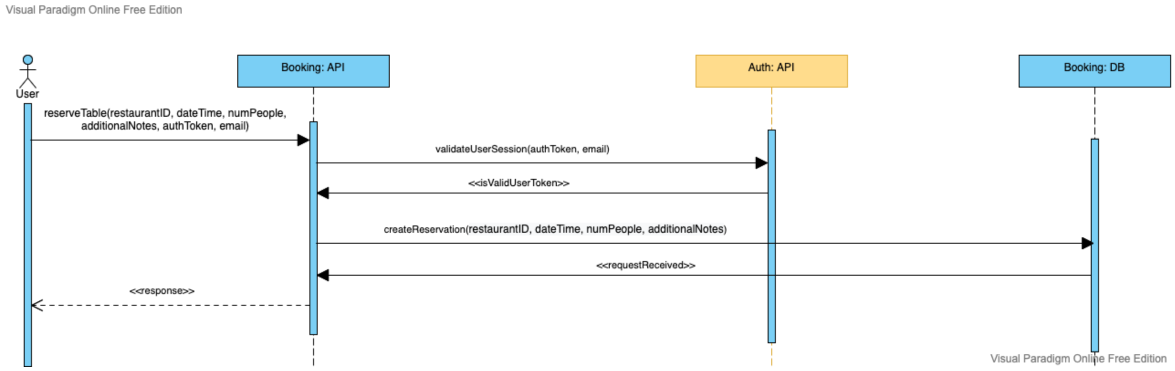

Visual Paradigm Online Free Edition
User
Booking: API
Auth: API
Booking: DB
reserveTable(restaurantID, dateTime, numPeople, additionalNotes, authToken, email)
validateUserSession(authToken, email)
<<isValidUserToken>>
createReservation(restaurantID, dateTime, numPeople, additionalNotes)
<<requestReceived>>
<<response>>
Visual Paradigm Online Free Edition

Figure 5: Reservation sequence diagram.

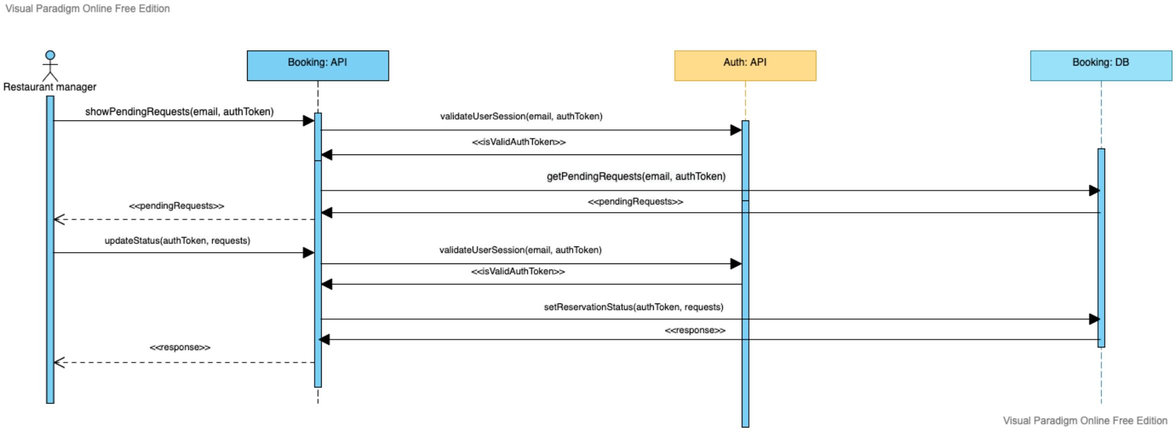

Visual Paradigm Online Free Edition
Restaurant manager
Booking: API
Auth: API
Booking: DB
showPendingRequests(email, authToken)
validateUserSession(email, authToken)
<<isValidAuthToken>>
getPendingRequests(email, authToken)
<<pendingRequests>>
<<pendingRequests>>
updateStatus(authToken, requests)
validateUserSession(email, authToken)
<<isValidAuthToken>>
setReservationStatus(authToken, requests)
<<response>>
<<response>>
Visual Paradigm Online Free Edition

Figure 6: Booking management sequence diagram.

2.2 Scalability, Availability and Continuous Deployment

By structuring the system into many microservices, there is the possibility to scale the single microservices indipendently. This is useful since the microservices may receive different loads of incoming traffic and this way there is more control over which component should be scaled. In a real world scenario, of course, standard users are expected to be much more respect restaurant account and we want to have an high level degree of elasticity.
Another important features is availability and fault isolation. If a part of the system goes down (e.g. restaurant data service), the rest of the application can still live and allow other operations (e.g. the reservation service).
The last important quality of this system is the simplicity, it is composed by small microservices that can be developed quickly, and that can also be upgraded in a fast way. We can explore the benefits of continuous development, in fact we can also decide to add other microservices in the future, as one for the reviews management, one for the mail notifications, and so on.

3 Solution implementation

Here we discuss the implementation of our solution. As already anticipated, the core of the system is a microservice architecture that communicates through the REST API. Since every microservice is independent of each other, the technologies that are used internally in the various modules do not have to be the same ones. So, apart from keeping a standardized REST interface among the components, we could develop the microservices with different programming languages and frameworks. Here are described the specific implementations of the single services.

3.1 Microservices

3.2 User auth

This module was built using Typescript that runs on a Node.js environment after being compiled into Javascript. The corresponding database was built using MongoDB that provides a NoSQL structure. The module uses the Express.js library to expose its REST API and the Mongoose library to communicate with the database. The POST http method was used to perform operations that edit entries in the database, such as during the registration. The GET http method, instead, was used for operations that do not modify content in the database, but only read information, such as the token validation.

Listing 1: User auth exposed API

```
POST register
JSON body data
    name: string
    surname: string
    email: string
    password: string
JSON response
    token: string OR error: string

POST login
JSON body data
    email: string
    password: string OR token: string
JSON response
    token: string OR error: string

POST logout
JSON body data
    email: string
    token: nullable string
JSON response
    token: string OR error: string

GET validate
JSON body data
    email: string
    token: string
JSON response
    token: string OR error: string
```

3.3 Restaurant auth

This module was implemented very similarly to the user auth one.

Listing 2: Restaurant auth exposed API

```
POST register
JSON body data
    name: string
    email: string
    password: string
JSON response
    token: string OR error: string

POST login
JSON body data
    email: string
    password: string OR token: string
JSON response
    token: string OR error: string

POST logout
JSON body data
    email: string
    token: nullable string
JSON response
    token: string OR error: string

GET validate
JSON body data
    email: string
    token: string
JSON response
    token: string OR error: string
```

3.4 Restaurant data

This module was build in python. Its corresponding database is build using MongoDB, which provides a NoSQL structure. Each restaurant is stored as a JSON object. The service uses the Flask library to expose the REST API and the pymongo library to communicate with the database.
The POST search method is used to search for a keyword among different restaurants. To overcome the MongoDB limitation of performing pattern matching for a nested object (the menu), each restaurant has a search_string attribute which is a string containing all the items in the menu.
The GET search method is used to get a restaurant from its unique id.
Additionally there are two GET methods, namely pin and pingdb, which are used to ping the server and the database to perform a health check. These methods are not exposed.

Listing 3: Restaurant data exposed API

```
POST search
JSON body data
    query: string
JSON response
    restaurants: list of json OR error: string
```

```
GET search
JSON body data
    id: string
JSON response
    restaurant: json OR error: string
```

3.5 Booking

This microservice module was completely written in python leveraging mainly the flask library to implement a REST API interface. Another technology that we used is MongoDB NoSQL database to store reservation records. As for the other services, database and applications logic are deployed in 2 different container. Apart from utilities methods such as ping the database or the microservices itself, the method that the service offers, following the design choices, are:

Listing 4: Booking exposed API

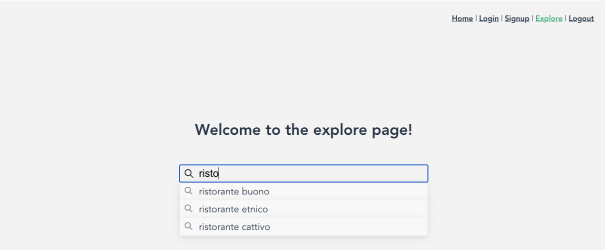
```
 POST reserve
 JSON body data
     email: string
     authToken: string
     rest_email: string
     date: string
     service: string in {'lunch' or 'dinner'}
     time: string
     seats: int
     notes: string
     status: string in {'pending'[DEFAULT],'accepted','refused'}
 JSON response
     body : "reservation␣pending"
     reservation_id: string
    OR error: string

 PATCH change_status
 JSON body data
     res_id: string
     status: string in {'pending','accepted','refused'}
     authToken: string
     email: string
 JSON response
     res: "Status␣Updated" OR error: string

POST my_reservations
JSON body data
     email: string
     authToken: string
     user_type: int [0 -> user, 1 -> restaurant]
 JSON response
     list of reservation records OR error: string
```

3.6 Web application

For the purpose of the project, we also implemented a web application that serves as a showcase to test the functionalities of our system. The backbone of our website is composed of elements of HTML and CSS, tied up using Vue.js to add the responsive component. To enforce types, we opted to use TypeScript in strict mode instead of the plain Javascript. We decided to use AJAX to send requests from the client to the servers. The interface is quite simple and implements the most important use cases that may be useful in the context of restaurant reservations, namely, we implemented the following use cases: user authentication, restaurant owner authentication, restaurant search using keywords, reserve a table, and handle an incoming user booking for a table. For a full description of how these functionalities are implemented please refer to their documentation in the microservices section. In particular, all requests that need to be authenticated are provided with an authentication token relative to the user session. Once the user is authenticated by our system, we store the authentication token that is provided by the user auth API in the local storage object. In this way, we are able to maintain the user session open even if the browser was closed. In such cases, we perform a technique known as silent sign-in using the above-cited token-based technique, enabling the user to proceed with their reservation transparently. For security reasons, we do not store user passwords on the client-side, neither encrypted ones, and the random generation of authentication tokens is performed on the server-side. Moreover, the website allows the non-authenticated user to view all available restaurants and see the details of a particular one, exception made for the booking request that can only be performed by a user that is currently logged into the system. A logged user can also see the list of their reservations, that have to be accepted by a registered restaurant owner. On the other side, we allow restaurateurs to see the collection of pending reservations that need to be accepted. Once the status of the reservation is changed on the restaurant side, the change is reflected in the user dashboard as well. It's worth noting that even though in this context the web application is only used to test the real system functionalities (implemented in the back-end), future work could be in the direction of improving the user experience on the website. Two example pages are shown in Figure 7 and Figure **??**.



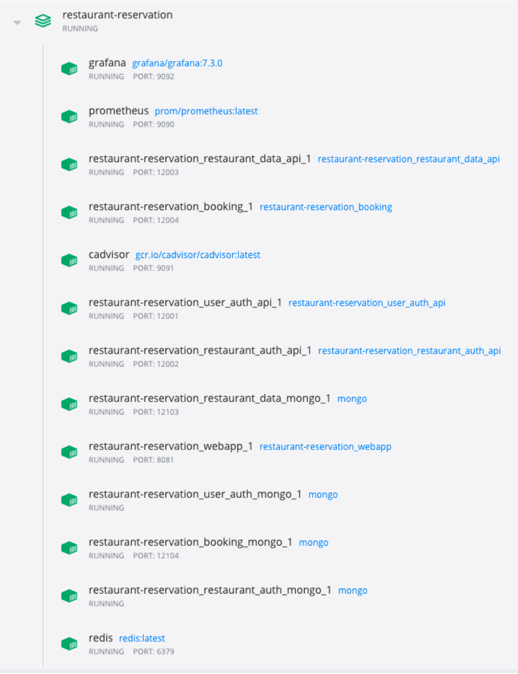
Welcome to the explore page!

Figure 7: Web app explore page.

Note: *To try our webapp and see all the microservices at work, please clone our git repository, compile with make, and run docker-compose up –build. WebServer is running locally on port 8081. A demo video sample is available* here*.*

4 Solution deployment

Our solution was deployed through Docker containers to make it more portable and easily manageable on different platforms. Thus, the microservices, the web app, and the monitoring tools have a dedicated Dockerfile that is used to instantiate the specific component and wrap it into a container. The containers are not started directly but are handled by Docker Compose (Figure 8). This tool makes it easier for the single containers to communicate with each other via internally resolved IP addresses and to perform replicas scaling. The wep app and monitoring tools are exposed on external ports in order to be accessible from outside the Docker Compose. All the microservices can instead communicate with each other via internal addresses.



```
restaurant-reservation
RUNNING

    grafana  grafana/grafana:7.3.0
    RUNNING   PORT: 9092

    prometheus  prom/prometheus:latest
    RUNNING   PORT: 9090

    restaurant-reservation_restaurant_data_api_1  restaurant-reservation_restaurant_data_api
    RUNNING   PORT: 12003

    restaurant-reservation_booking_1  restaurant-reservation_booking
    RUNNING   PORT: 12004

    cadvisor  gcr.io/cadvisor/cadvisor:latest
    RUNNING   PORT: 9091

    restaurant-reservation_user_auth_api_1  restaurant-reservation_user_auth_api
    RUNNING   PORT: 12001

    restaurant-reservation_restaurant_auth_api_1  restaurant-reservation_restaurant_auth_api
    RUNNING   PORT: 12002

    restaurant-reservation_restaurant_data_mongo_1  mongo
    RUNNING   PORT: 12103

    restaurant-reservation_webapp_1  restaurant-reservation_webapp
    RUNNING   PORT: 8081

    restaurant-reservation_user_auth_mongo_1  mongo
    RUNNING

    restaurant-reservation_booking_mongo_1  mongo
    RUNNING   PORT: 12104

    restaurant-reservation_restaurant_auth_mongo_1  mongo
    RUNNING

    redis  redis:latest
    RUNNING   PORT: 6379
```

Figure 8: Docker compose in execution.

5 Test design

One of the benefits of having a microservice-based architecture is the ability to scale different microservices independently, based on the load on a particular service. Therefore, we designed our solution with that goal in mind. Docker Compose allows scaling the number of replicas of given containers (i.e. our microservices) via "docker-compose scale". To perform scaling in an automated fashion we implemented an autoscaler through a Python script. This script monitors the resource utilization per container with a fixed cadence and decides whether to scale up, down or leave the single microservices as they are. More specifically, the script gets resources information using "docker stats" and, for every microservice, it increases or decreases the number of replicas in the Docker compose depending on whether the usage is above or below fixed thresholds.

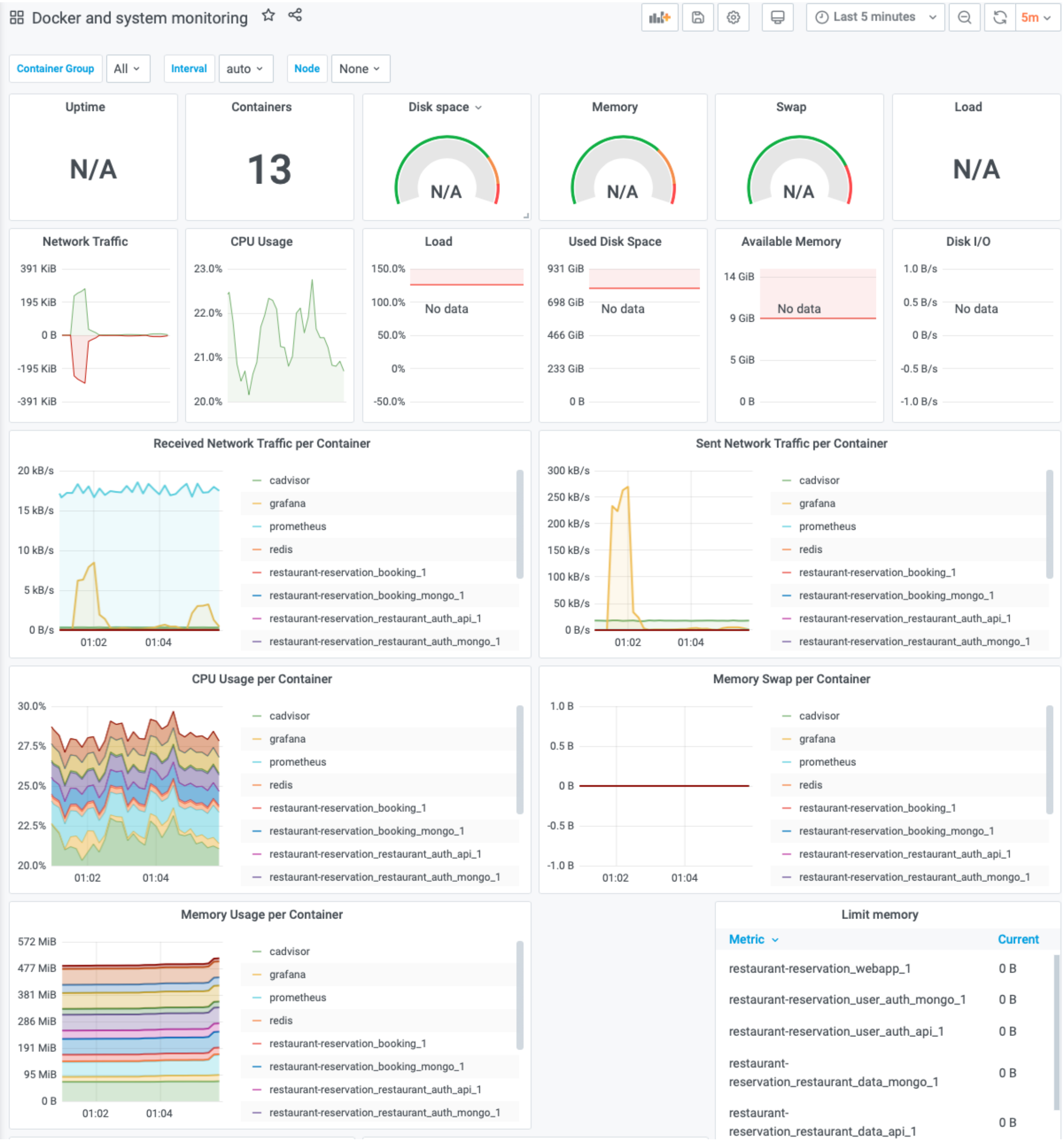
In order to visualize and test the performances of the microservices under regular and stress conditions, we utilized a series of tools. cAdvisor was used to extract metrics from the containers, Prometheus to query cAdvisor and collect the necessary metrics, and Grafana to show the metrics on charts and dashboards interactively (Figure 9). The metrics are temporarily stored by these tools using Redis. Using this monitoring system, it is easy to see the resources that are used by each container at any point in time. To stress targeted containers, we decided to use the Pumba tool the leverages the Linux stress-ng mechanism and sends workloads to Docker containers. It can be used to stress a particular container for a given amount of time and this allows us to see the metrics changing in the dashboard and the autoscaling mechanism operating on a per-container basis.

6 Experimental results

Tests were conducted by individually stressing some containers using Pumba and checking the autoscaling response to those events.

In the reported example, the User Auth API and Restaurant Auth API have been stressed. The initial load of the experiment is very low and, as shown in Figure 10, there only is one replica per microservice and the CPU usage is very low. Then, we stress User Auth API for a short amount of time and, the autoscaler detects the load increase and creates a new replica (Figure 11). After stressing that microservice, the load goes back to a low level and, there is not enough time nor load for the autoscaler to create a third replica; Restaurant Auth API still only has one replica since it has not been stressed yet (Figure 12). Eventually, we start to stress Restaurant Auth API and, as shown in Figure 13, the autoscaler creates a new replica of that microservice. In the meantime, the autoscaler also removes the additional replica previously created for User Auth API, since its load decreased enough since then.

As a result, the system turned out to properly scale according to the given thresholds and, this way, the load can be balanced across several replicas that in a potential production system could be located on different nodes, distributing the computation on more than one physical machine.



Container Group: All
Interval: auto
Node: None

Limit memory

Metric	Current
restaurant-reservation_webapp_1	0 B
restaurant-reservation_user_auth_mongo_1	0 B
restaurant-reservation_user_auth_api_1	0 B
restaurant-reservation_restaurant_data_mongo_1	0 B
restaurant-reservation_restaurant_data_api_1	0 B

Figure 9: Grafana metrics dashboard.

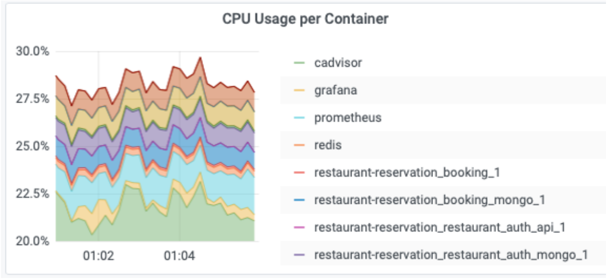


Figure 10: CPU usage per container before stressing the User Auth API microservice.

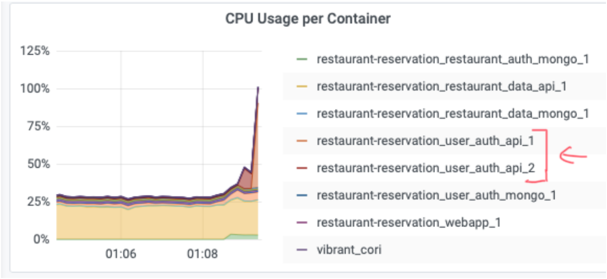


Figure 11: CPU usage per container after stressing the User Auth API microservice.

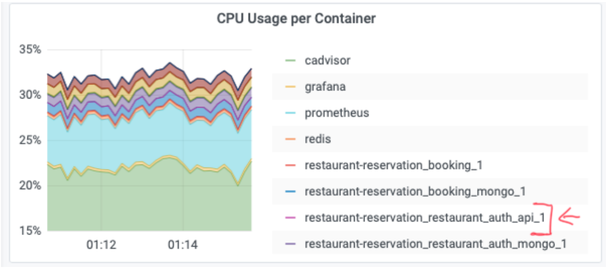


Figure 12: CPU usage per container before stressing the Restaurant Auth API microservice.

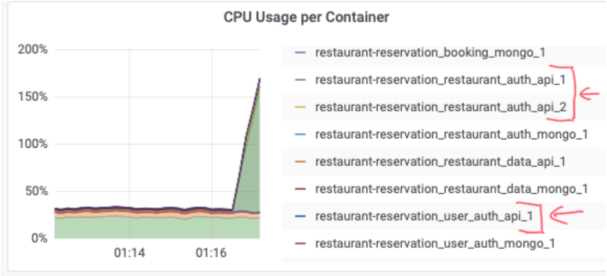


Figure 13: CPU usage per container after stressing the Restaurant Auth API microservice.