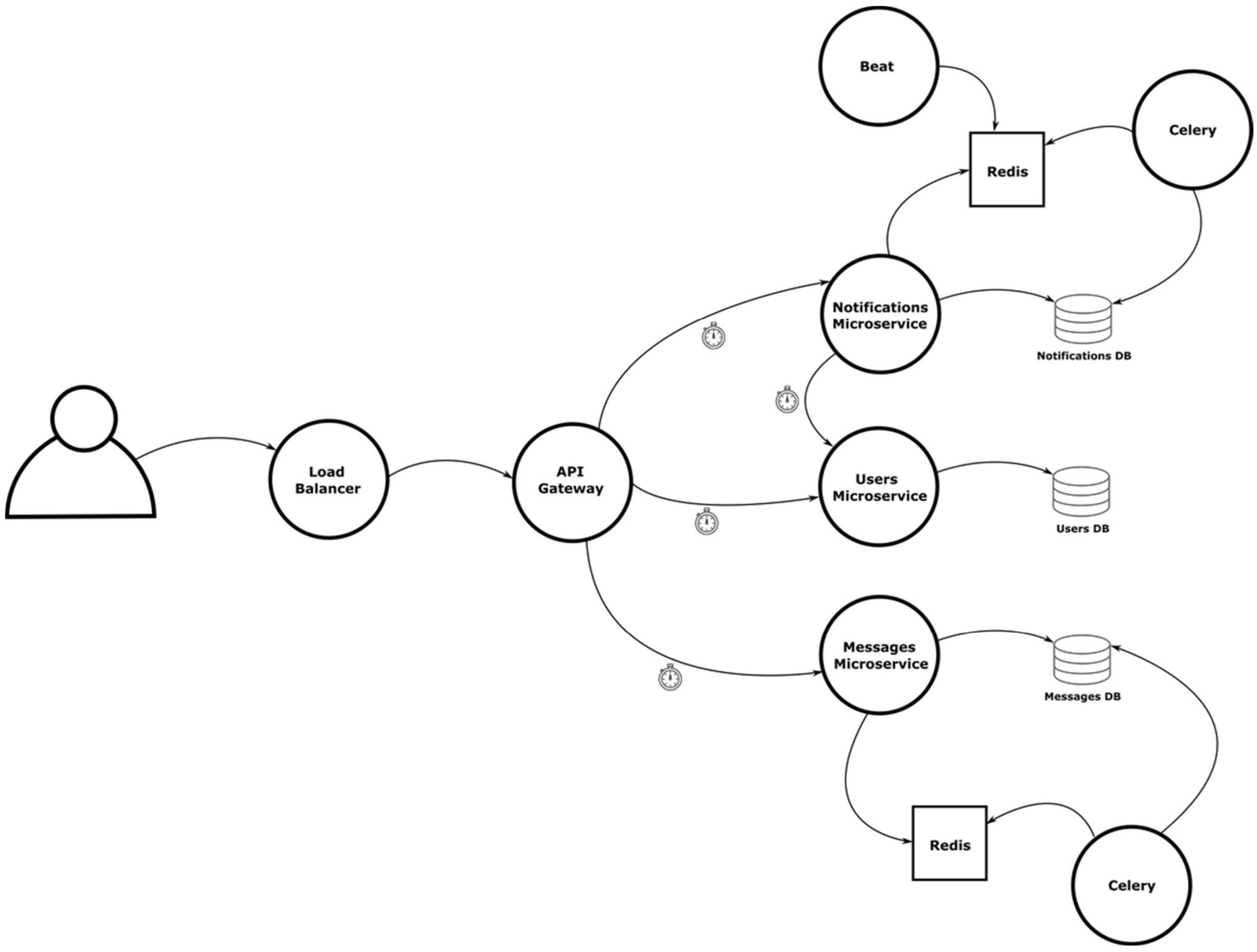
Group name: Squad 10

Leader: Giuseppe Crea

Recap Table

Student no.	Full name	GitHub ID	Commits no.	Added Lines no.	Deleted Lines no.
501922	Crea Giuseppe	giuseppe-crea	54	4073	934
627052	Gargiulo Francesco	gargiulofrancesco	34	2155	93
505876	Sarno Ivan	ivansarno	40	23187	11976
548415	Venturini Francesco	Portgas97	28	2794	343
		Total	156	32.209	13.346

System Description



In the microservice version of My Message in a Bottle users query the load balancer for an instance of the API-Gateway/frontend. This frontend gateway renders the views for them and queries other services to populate said views.
These queries happen via API calls with a given timeout built into them, to safeguard against microservice failure.
No actual data is stored on the frontend gateway.
All user data aside from the blacklist is stored in the Users microservice, while notifications and messages are stored in the respectively named Services.

The Microservices were built using Swagger Editor's Server Generator, starting from hand-written OpenAPI specifications. This generated a fair bit of boilerplate code, which we have excluded from coverage.

Celery is still used for asynchronous tasks, working in the same way as it did in the monolith, but now split over multiple services each with their own celery and their own redis.

Why we chose this split

The new architecture uses four microservices to replace the monolith.
Users connect to the API Gateway through the load balancer. The gateway acts as frontend and calls the other microservices as needed.
Some of these microservices use redis and celery to carry out asynchronous tasks, and they all use a database containing different tables, specific to their function.

The four-way split was chosen on which 'seams' seemed more apparent to our team, after a joint design meeting. They are divided as follows:

1 *API-Gateway:* This service acts as frontend for the user, hosting the code which takes care to render the needed views for our application. The choice to unify frontend and gateway was made due to the centrality of the frontend itself in the application. It requires all other services to function, and makes complex calls based on the user's flow through the application. Centralizing all calls in this service seemed natural.

2 *Users:* Storage for all user information, including Reports. This was an obvious split as the frontend could validate this information and then call the other services for specific tasks, without need to replicate user information across all remaining services.

3 *Messages:* Service in charge of all things which have to do with messages, blacklist included. This is another obvious seam in the architecture, as creating and fetching messages doesn't require all the information stored in other databases, and this is a prime target for scalability.

4 *Notifications:* Another self-sufficient part of our architecture. Once instanced, a notification doesn't need any information to be displayed or sent to the correct user. This service also calls the lottery task, querying the Users microservice once a month. This was done in Notifications rather than Users because of the already present instance of celery and redis.

How to run

After recursively cloning the repository, call docker-compose build from the root folder, followed by docker-compose up. The application will be accessible by navigating to localhost, on the default web port 80.

Other possible splits

Splitting Reports into their own microservice, dedicated to administrative functions.
Inserting blacklist in the User microservice rather than the Messages, having messages query it.
Adding a different container for Lottery altogether, querying both Notifications and Users.

Further Considerations

No Continuous Integration pipeline was used as the suggested one, Travis, results unavailable to all members of the team. We still proceeded with a test-driven development approach using pytest and tox.

Work was split by the microservice. Team members took one microservice each among the Users, Notifications and Messages and then worked together on the Gateway.

Test coverage

Microservice name	*Test coverage (out of 100%)*
MMIB-ApiGateway-Submodule	**64%**
Users-Service	**91%**
MMIB-Messages-Submodule	**97%**
notifications-service	**90%**

Repo link:

https://github.com/Portgas97/MMIB-main