The purpose of this document is to present the architecture of a microservices-based project.

Idea:
The project is proof-of-concept that features:

- Frontend application where the user can log in / sign up in order to use the service
- The frontend application connects to a backend which exposes several endpoints.
- The frontend communicates to 2 microservices which serve different purposes.

UX flow:
The user can fetch a list of Start Wars vehicles, and they can add them to their owned vehicles (this operation in turn removes the vehicle from the buyable vehicles) or they can sell a vehicle (an operation which in turn makes the vehicle buyable again).

Technology stack:

1. **FRONTEND**

The frontend is a React application that uses Bootstrap for styling the user interface.

In order to secure this web application, I used firebase. Both the frontend and the backend are connected to Firebase. When logging in, the user obtains a bearer token from Firebase. Then, when making any request to the said backend, If this bearer token is missing, then the backend will refuse any attempt to call its endpoints.

2. **BACKEND**

The backend consists of a NEST JS application which listens of port 3002 to any incoming connections from the frontend. Then, based on what endpoint has been called, it will emit an event to the microservice client that can handle this event. The following endpoints are available:

- /vehicles - sends a get-vehicles event to the Swapi microservice
- /addVehicles - sends an add-vehicles event to the Swapi-crud microservice
- /deleteVehicles - sends a delete-vehicles event to the Swapi-crud microservice

SECURITY: the security is ensured by the preauth middleware from the backend. This middleware verifies whether or not the bearer token that is sent in a request is indeed one from firebase. If not, then access is denied.

3. **MICROSERVICES PROVIDERS**

As mentioned above, in order to showcase the microservices architecture, the backend makes use of 2 microservice providers which can do different operations.

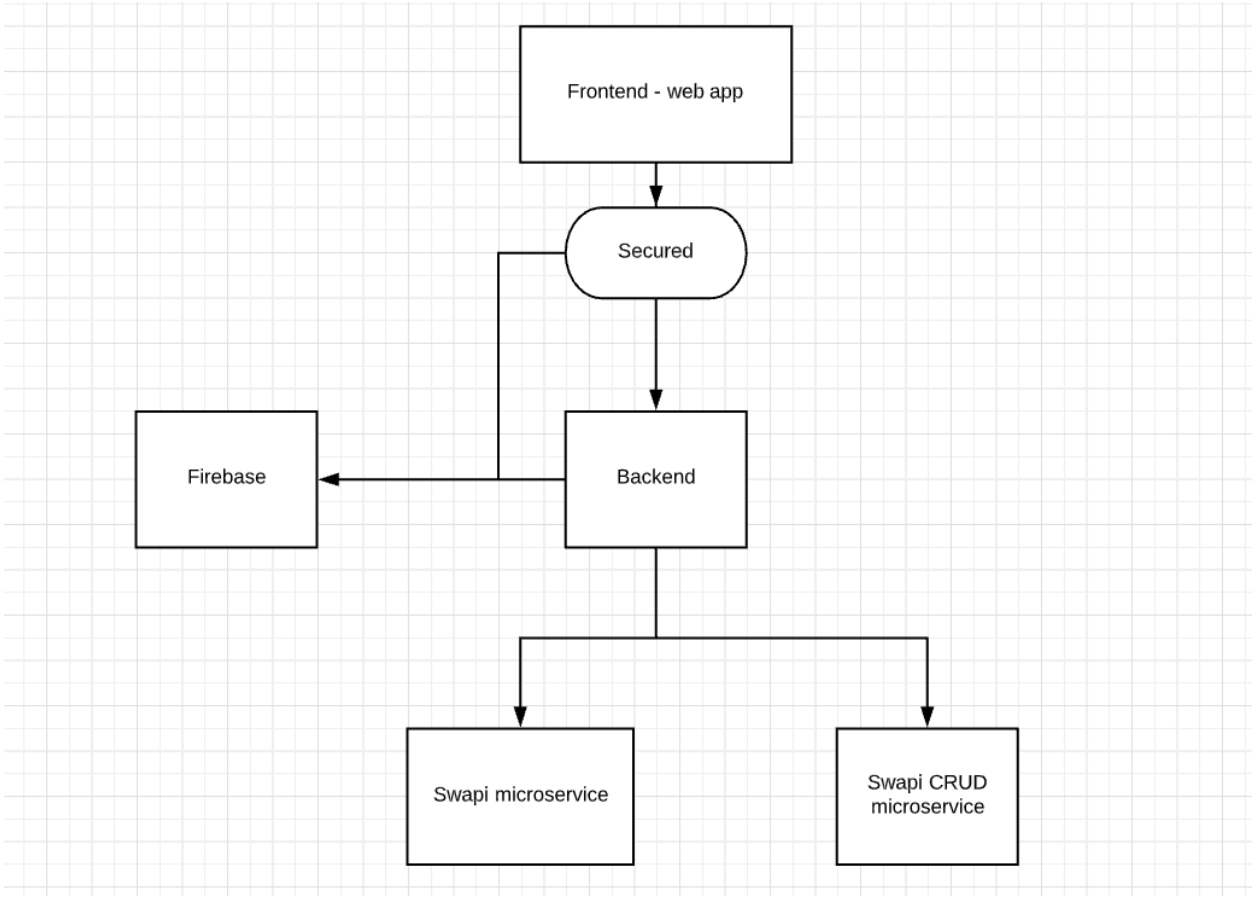
a. Swapi - this is the first microservice, which is able to handle the /vehicles (get-vehicles event) operation. When this event is sent to the microservice provider, it calls the free public API https://swapi.dev/api/vehicles/ which in turn responds with a list of vehicles that appear in the Star Wars universe.
b. Swapi-crud - this is the second microservice. Its purpose is to handle the add-vehicle and delete-vehicle events.
    i. Add-vehicle: for this event, the backend also sends the data that can identify the vehicle that should be deleted. In our case, the UID of each vehicle consists of its URL, since no other unique ID is provided from swapi.co. in turn, the microservice makes a call to swapi.co in order to fetch the vehicle data. Then it adds it to the list of vehicles which lives on the microservice (volatile) memory. Then it sends it in the response to the backend, which in turn sends it to the frontend to show it on the UI.

    ii. Delete-vehicle: this event needs to receive the name of the vehicle in order to delete it from the list of vehicles. After completing the operation, the updated list of vehicles is sent in the response to the backend, then back to the frontend.

SOA concepts and patterns:

1. **Decompose by business capability** - the main functionalities of the application are spread among several microservices (in our case, 2) which do not depend on each other. Each microservice implements a small set of business operations. Each microservice is small enough and easy to maintain. This leads to a set of loosely coupled services.
2. **API Gateway:** this pattern is showcased by having a backend as the single source of truth for the frontend. This backend exposes a set of API endpoints which are callable. By calling them, some specific events are afterwards sent to the microservice providers which can then handle them, without the frontend knowing about them or them knowing about each other.
3. **Access token:** this pattern is showcased by the aforementioned Firebase-enabled security which ensures that all requests must have a bearer token provided directly from firebase. Without this, the backend will deny the access.

Architecture diagram:



Screenshots: