KU LEUVEN

FACULTEIT INDUSTRIËLE
INGENIEURSWETENSCHAPPEN

CAMPUS GROEPT

Distributed Systems Project Report

Team 18

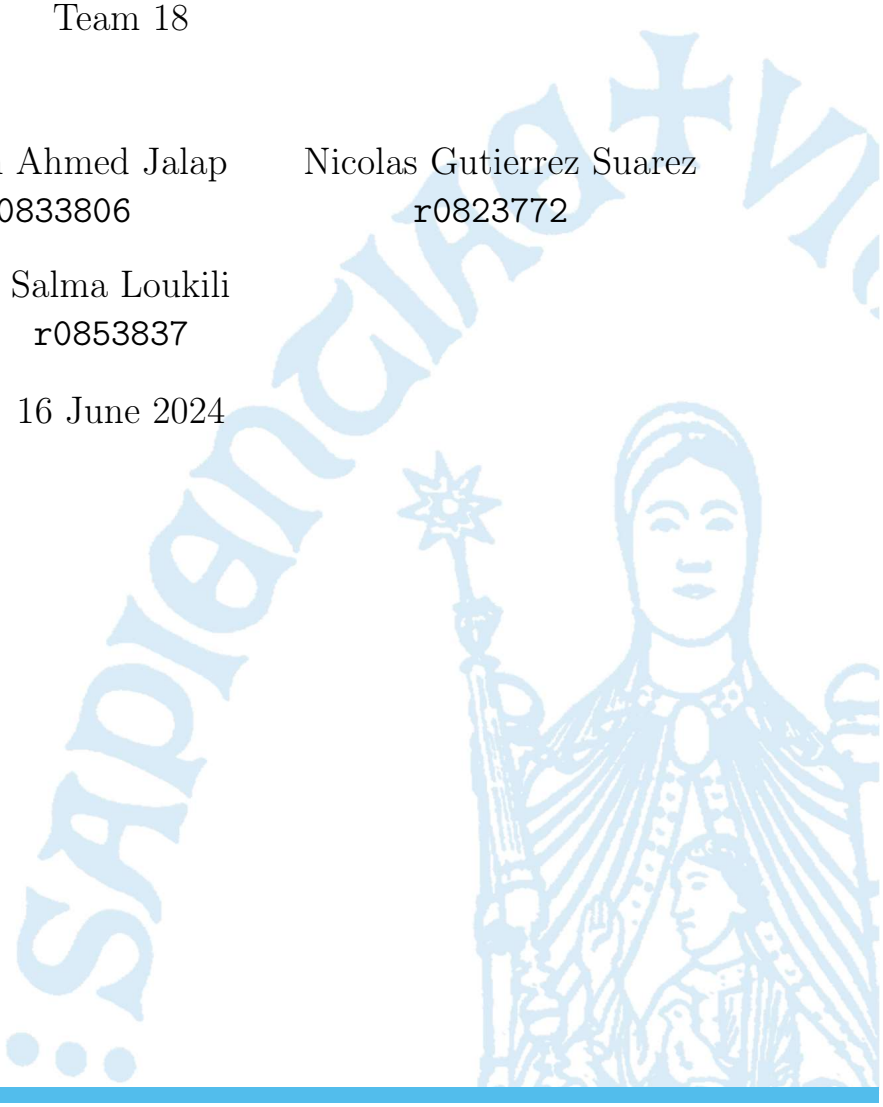
Nur Alda-Onggar
r0823742

Hassin Ahmed Jalap
r0833806

Nicolas Gutierrez Suarez
r0823772

Salma Loukili
r0853837

16 June 2024



Contents

1. Level 1: Basic Requirements

1.1 Architecture & Deployment

We designed an event broker that sells packaged products from transportation, catering, and event vendors. Users can purchase packages to go to an event with a bus, and select some food to eat there. Figure 1.1 provides an overview of our system, where the front-end (React) communicates with cloud functions (Typescript), which are also set up to communicate with the vendors (Flask). The functions serve as API endpoints for the front-end and also allow the system to keep an up to date state of any number of vendors. These micro-services provide fault tolerance from the failure of separate system nodes and allow almost seamless scaling for periods with high demand. All the vendors use JSON:API, a HATEOAS standard that is much more feature rich and complete than HAL.

Figure 1.2 shows the communication between the system nodes. The vendors' data is cached in Firestore and Cloud Storage for quick access for the front-end and API endpoints. This happens on a schedule, which means the data stored locally is eventually consistent. Whenever an operation that changes data is done (such as reserving, buying, etc), the vendors are contacted directly to ensure no data inconsistency has occurred. If the operation succeeds on the vendors, then it is also written locally. Critically, even if the cached data is out of date (for example the price of an event is higher), the moment a user decides to reserve the package they will be notified of the change.

As for the deployment, Figure 1.3, our main system is located on Europe Central. The front-end is deployed on serverless hosting, this makes it much cheaper than the Spring-

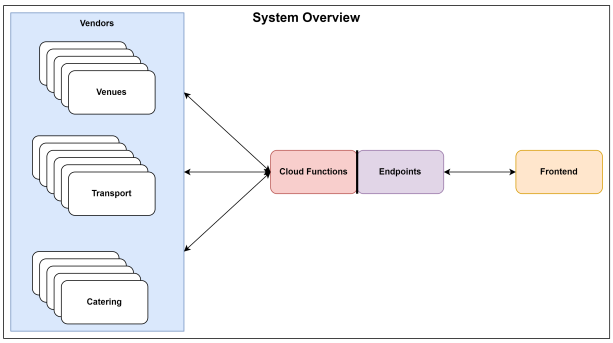


Figure 1.1: System Overview.

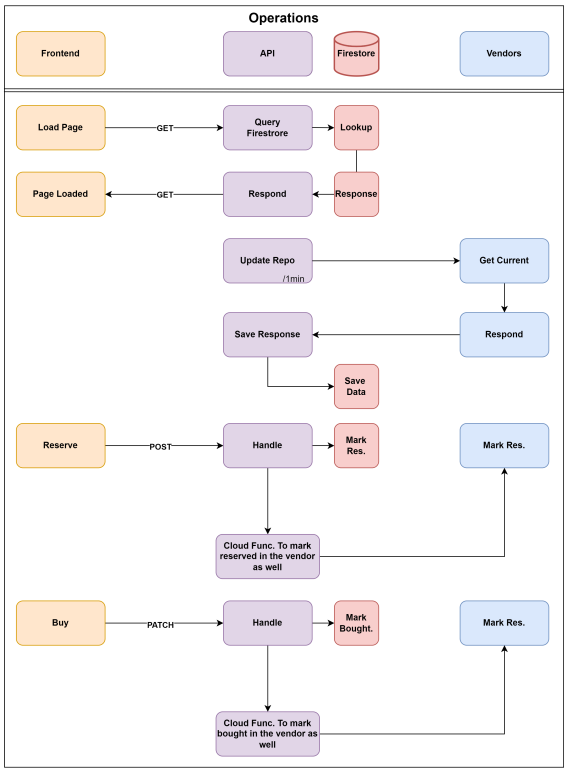


Figure 1.2: Operations Overview.

Boot alternative and allows us to use CDNs to serve the site. Access times throughout the globe can be reduced and there is no need to manually spin-up servers. The Cloud Functions are also managed and give us complete flexibility on deployment (since they are stateless). By default, they can handle 1000 re-

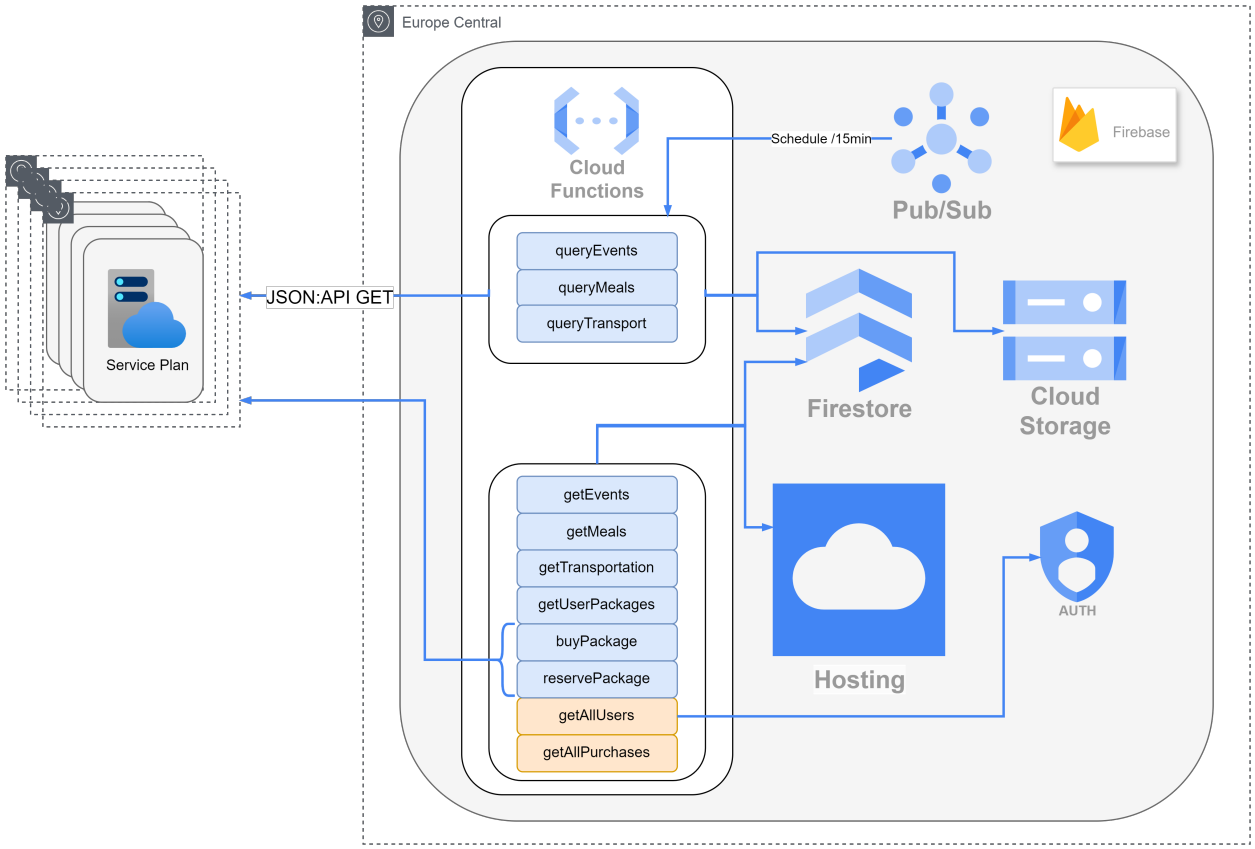


Figure 1.3: Application deployment diagram.

quests per instance and automatically spin-up more when needed. They can also be deployed all over the world by just adding a line of code. As a database, we are using Firestore, this service can also be replicated around the globe with very little effort. Images from the vendors are all cached with Cloud Storage, this is a bucket service for long-term, high-volume storage. Overall monolithic designs would be too cumbersome and require constant manual supervision to deploy, although in the long run serverless tend to be more expensive.

For the vendors, the Flask application was added to a Docker image. The application was split into 3 main modules, each one for a vendor. These modules activate based on a random choice that gets seeded through an environment variable. This ensures that the data stays the same for a specific vendor even after being deployed many times. After creating the Docker image, it was pushed to the Azure Container Registry, Figure 1.4. From there, 4 service plans on different regions were spawned and 5 different vendor images spun-up on each service plan. After the images get initialized, they automatically generate some fake data that the broker can ingest.

1.2 Access Control

The first type of access control is implemented in the front-end. Using Firebase Authentication users can create new accounts or login to the service. Everything is left to the Google library to handle *(Dont Roll your Own)*, once login is complete a callback is generated and we can access user credentials. In the site itself, the admin pages are not shown to the user if they don't have the correct credentials. Being realistic, anything client-side can be manipulated, so we assume a user could get access to the admin-only pages. Nevertheless, the second layer of security comes into play. When calling a Cloud Function, the user credentials are once again checked and injected into the request. This means that if I somehow manage to gain access to the admin page, I could not get any data from the endpoint

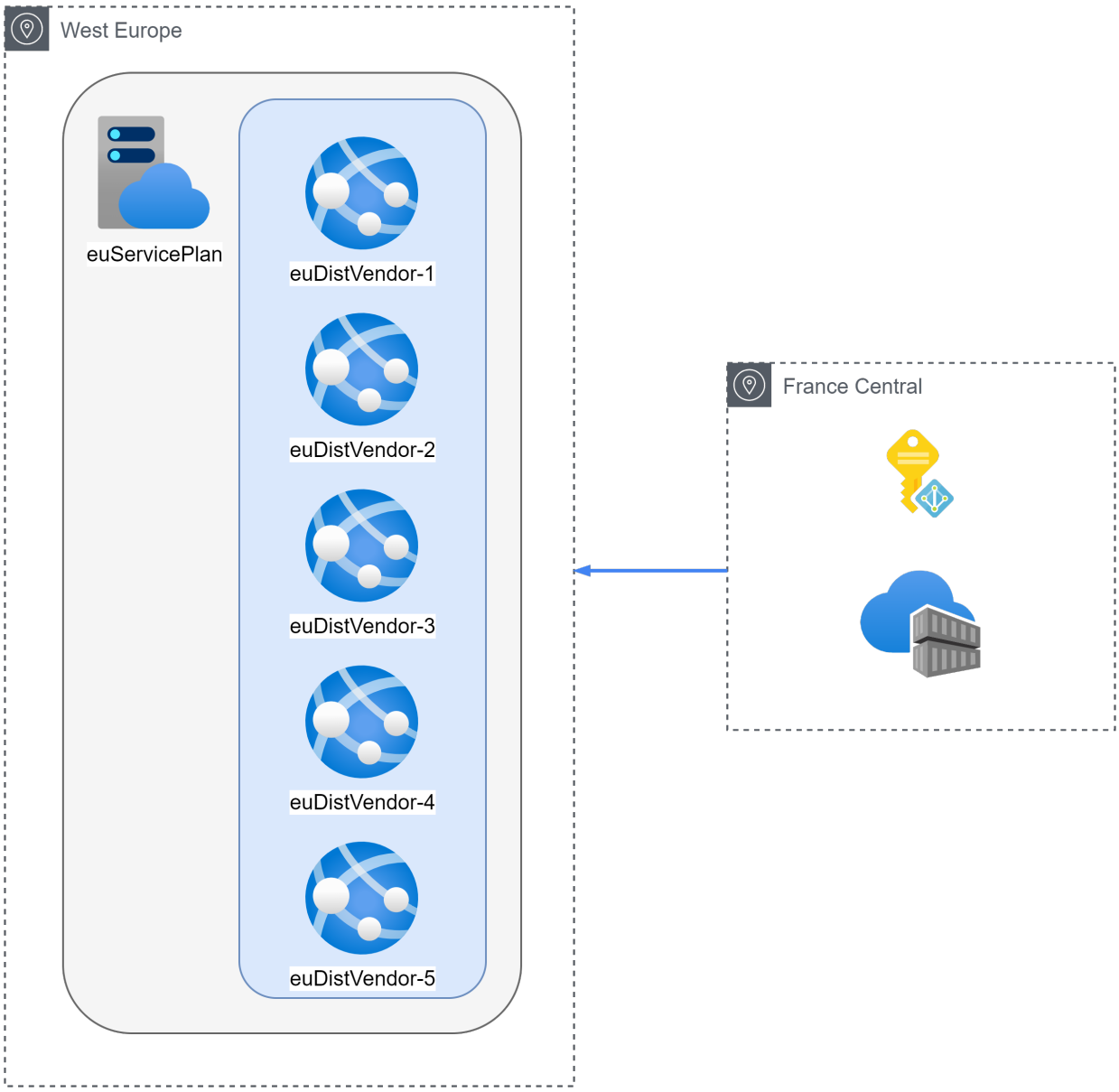


Figure 1.4: Vendor deployment diagram.

since Google ensures they are correct.

For the vendors, a simpler scheme was developed due to the large amount of vendors. A unique API key is generated randomly and gets checked whenever anyone makes a request through a bearer token. This key-gen is based on a seeded (pseudo)random generator, so since we pass the vendor a seed when deploying we can know what the value of the key is. This is by and large a bad approach, but for the purpose of this more than enough to ensure only we can access the API. A much better implementation would be OAuth 2.0, then we can control which access tokens we still want to allow and also make sure they have a time limit.

1.3 ACID

The cloud functions are performed in an atomic way, if at any point a vendor or Firestore fails, the function fails. An exception to this is the buy all button, which simply skips over the packages it could not buy. The functions are scaled up to perform the operation in a parallel manner when multiple users request it. The data is "eventually" consistent which is ensured through the sequential manner of operations by firstly updating the data in the

vendors' database, if needed, through cloud functions, and if the operation is successful the same happens in Firestore. Firestore ensures durability and has constant backups and replicas.

1.4 Fault Tolerance

Our application is fault-tolerant towards the external supplier's service, as the endpoints send the updated data to the vendor and update the data of that vendor in Firestore. Hence, this operation is atomic, and if one of the operations fails the other one is not performed. If any of the vendors fail then the user cannot buy from that vendor temporarily but the other vendors will not be affected. If a Cloud Function fails, another instance will take its place. Finally, the front-end will always be accessible unless the CDN fails (unlikely) or there is a serious software bug that made it to production.

2. Level 2: Advanced Requirements

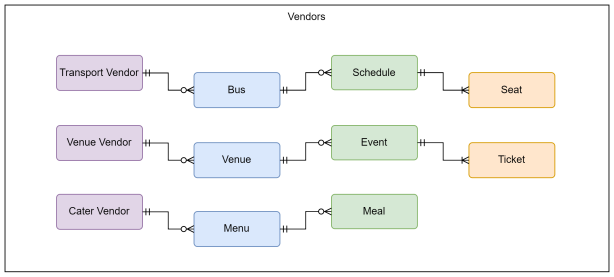


Figure 2.1: Vendors Data Model.

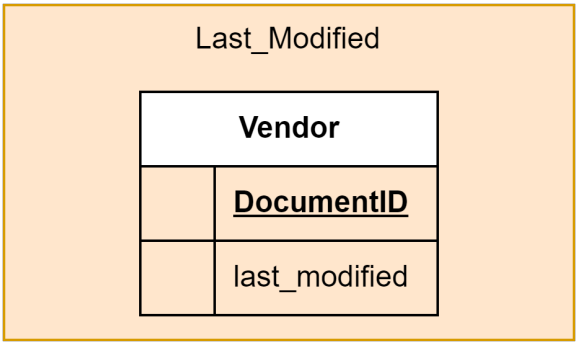


Figure 2.2: Last Modified Data Model.

2.1 Data Model

Firestore holds 3 collections: *purchases*, *vendors* and *last_modified* where the second is used to store vendor data and the last to store the last time our application checked the updates in the vendors' own databases. Purchases store duplicate data with reservations or bought packages.
As can be seen from figures 2.1 and 2.2, which provide a high-level view of our collections; our data model follows a tree-like structure (as it is easy to query) for nested objects, while flattening things that do not require to be nested. For example, *vendors* collection, holds the document of each vendor with the respective fields, and each document has a corresponding sub-collection and so on. This tree-like structure provides us with a flexible way of mapping the vendors' data to Firestore. Firestore makes it very simple to query sub-collections, if multiple vendors have multiple buses and these multiple schedules (vendor/**/schedule/**/) finding all would require a for loop. Collection Groups allows us to index certain collections and then easily querying all of them without having to worry about their parent refs. 0 The *last_modified* collection holds the document of each vendor with the last_modified field. The Purchases collection is a flat collection of packages that were sold. This includes data like the event, ticket, venue, meal, etc. This is not good practice for relational databases but it is acceptable in NoSQL to make queries easier. The data schema for NoSQL should be focused on the queries and not so much on the data.

2.2 Query Limitations

The database was designed with the queries in mind and so we did not observe any severe limitations or roadblocks. It is sometimes hard to keep track of objects but the solution was found to pass the reference of the object to the front-end and therefore never lose track of it. The hardest part is probably doing the batch updates (to ensure atomicity) since they can sometimes behave strangely.

2.3 Transactional Behavior

At the beginning, all the endpoints were all or nothing. Eventually, the buy all button was converted to transactional (in a way that makes sense). When a user buys a package it makes no sense for it work when only two

vendors were successful, nevertheless, when a user buys 4 packages, it makes sense that if one failed the other ones should not be affected. This transactional behavior was not much different to make. All of our system was designed to be as modular as possible and as fault tolerant as we can.

2.4 Google Cloud Tie-In

The system is heavily reliant on Cloud Functions, Firestore, and Authentication. The main issue with leveraging serverless/managed solutions is that we end up tightly coupled to them. Other cloud vendors do not really offer solutions comparable to Firebase when it comes to the integration of services (like Authentication and Storage) directly to the front-end. It would be very hard or almost impossible to migrate this without a complete rewrite.

2.5 App Engine User Credentials

The site is not using App Engine, it uses Firebase Hosting.

`https://event-package.firebaseapp.com/login`

Admin User Credentials:

- admin@admin.com
- AdminPassword1234

Regular User Credentials:

- user@regular.com
- UserPassword1234

3. Team Member Contributions

Nicolas Gutierrez Suarez:

- Role: Full-stack/Cloud Developer
- Hours: 120
- Tasks:
  - System Architecture
  - Created the Vendors
  - Made the logic to get data from vendors.
  - Made some endpoints.
  - Fixed integration errors.
  - Created local environment.
  - Dockerized/Deployed all vendors to Azure.
  - Deployed to Firestore.

Salma Loukili:

- Role: Frontend developer
- Hours: 40
- Tasks: Developed the UI in React and implemented all the functionalities, mainly by querying the API endpoints:
  - Rendering all package items: Events, transportation, and food. Including all their details, corresponding images, and vendors.
  - Cart: Reserve, buy or cancel packages.
  - Login: Firebase authentication and user roles distinction.
  - Admin dashboard: render all transactions.
  - Admin Users: render all users and their information.
  - Tickets page: render all packages of a user.
  - Error handling.

Nur Alda-Onggar:

- Role: Backend Developer
- Hours: 30
- Tasks:
  - Vendor database retrieval
  - Scheduled functions to update firestore to be consistent with vendor's external database
  - Schema definition for all source objects

Hassin Ahmed Jalap:

- Role: Backend developer
- Hours: 35
- Tasks:
  - Developed endpoints
    - Fetching events, transportations, meals
    - Fetching all purchased packages for a user
    - Reserving, purchasing and removing a package
    - Ensuring consistency between local database and vendors
  - Contributed to local schema definition

4. Instructions to build the project

Setup

To set up the project for the first time run the following:

```
chmod +x ./setup.sh
```

```
./setup.sh
```

Activate the Virtual Environment

Activate the virtual environment by running the following command:

```
source ./.venv/bin/activate
```

Running

The setup script automatically builds the sites so you only have to run:

```
npm run dev
```

This deploys the website locally and initiates a Firestore database, which will store data locally for faster access. The local website can be accessed through the following URL: `http://localhost:3030/`