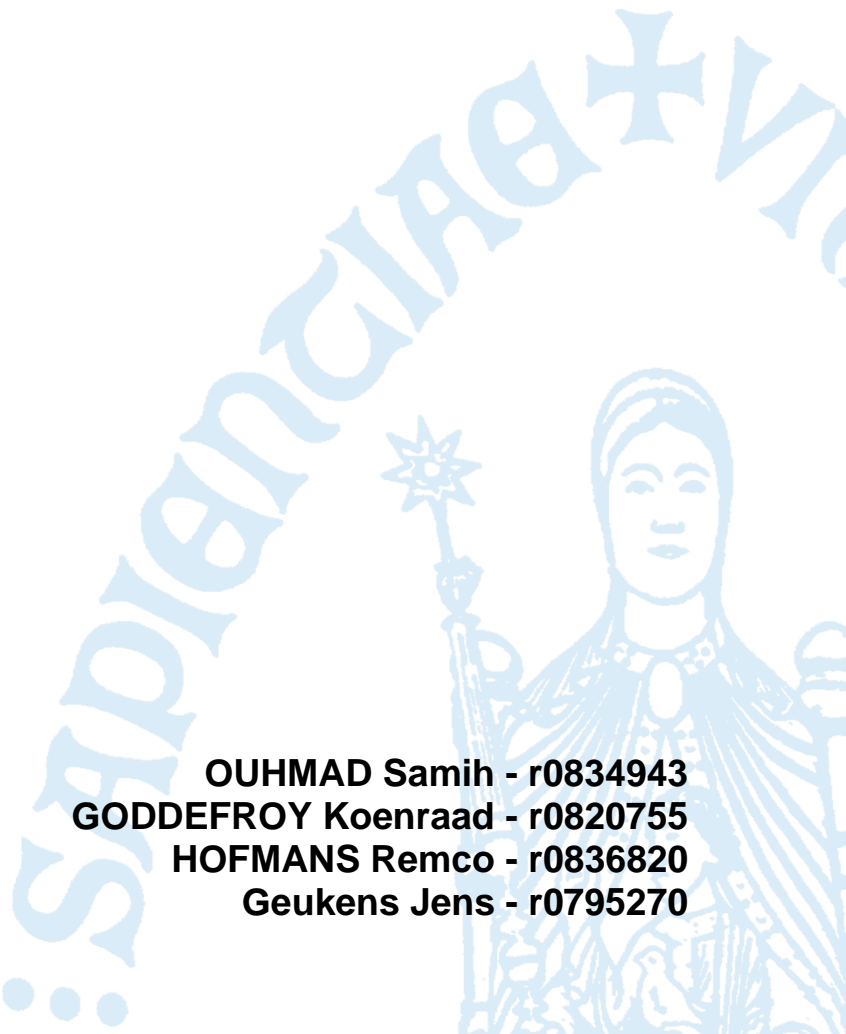


Final project: service-oriented distributed applications on a cloud platform

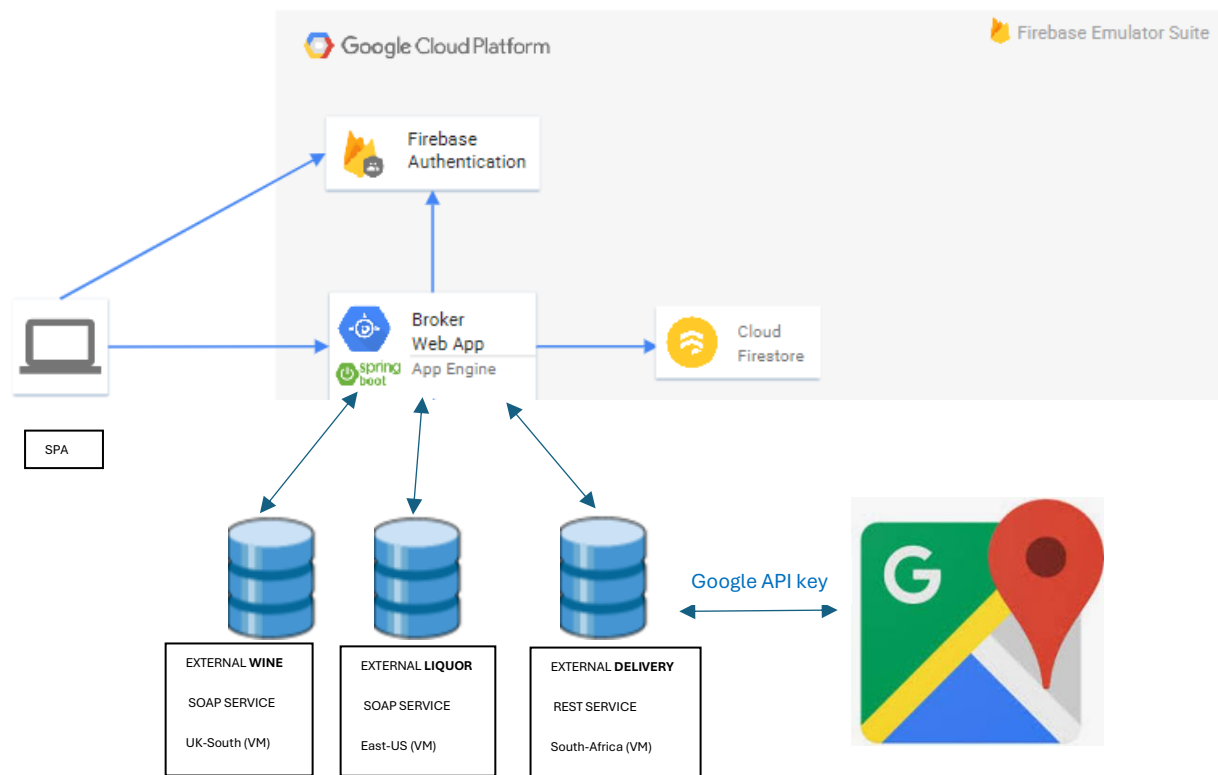
Distributed systems



OUHMAD Samih - r0834943
GODDEFROY Koenraad - r0820755
HOFMANS Remco - r0836820
Geukens Jens - r0795270

maart 2024

Distributed architecture scheme



In order to run the complete application:

Run “run-script.sh” .

Add a user in the Firebase.

Check <http://localhost:9090/> to see the website.

Broker

Wine distributor - SOAP web service

The customer can order exclusive wines/liquors through the broker. On the ordering page of the broker, a drop-down menu allows the customer to make a selection of all available wines and liquors, with each item accompanied by its corresponding cost price. These drinks can be added to a shopping basket that is stored in the Firestore database, together with the total price for the drinks and delivery costs.

Project Considerations and Configurations

The team opted to work with SOAP to establish communication between the wine/liquor endpoint and the wine/liquor web services, as it would most likely present itself as a challenge on an educational level.

We began by ensuring that the project's *pom.xml* file included the Spring-WS dependency. Spring Boot will then automatically export the defined XML schema (XSD) as a WSDL, to offer an interface to the functionalities of the (online) web service.

In more detail, the *pom.xml* configuration utilizes Spring Boot's *spring-boot-starter-parent* for managing dependencies and plugins. This parent project ensures that all necessary dependencies, including those transitive from *spring-boot-starter-web-services*, are readily available. Additionally, the *jaxws-maven-plugin* is configured to generate Java classes from a specified WSDL file (in our wine application defined at <http://dapp.uksouth.cloudapp.azure.com:12000/ws/wines.wsdl>). This plugin handles the entire process of parsing the WSDL and generating the required Java classes, which simplifies the client-side integration of the web service and improves consistency and accuracy!

To fetch the data from the liquor Service a call is made to

<http://dappvm.eastus.cloudapp.azure.com:12000/api/liquor-info> .

Client Request and Communication Flow SOAP

In a typical SOAP-based communication scenario between the wine/liquor endpoint and the supplier for wines and liquors, the process involves serialization and deserialization of XML requests and responses. Initially, before sending the request over the network, the *SoapClient* marshals a JAXB-generated *GetWineCardRequest* into XML and is handled by the *Jaxb2Marshaller* (from Spring's OXM module).

The serialized XML request is then ready to be transmitted via HTTP(S) to the distributor's endpoint (*SoapServiceEndpoint*). The wine or liquor web service consequently parses the received XML request based on the operation specified (e.g., requesting the availabilities or fulfilling a wine or liquor order and updating inventory).

After processing, the supplier generates a SOAP response, which is serialized back into XML using JAXB. This serialized XML response is then sent back to the wine/liquor endpoint.

Upon receiving the XML response, the wine/liquor endpoint deserializes it using JAXB. The XML response is more specifically converted into POJOs that the broker's application can understand and process. The deserialized response contains the results of the operation

performed by the distributor, such as a confirmation of the order status or details about the requested wines or liquors.

The communication between the web client and the liquor/wine endpoint is done by using a rest service. Firstly, a soap client was made for the liquors and the wines, and a working dummy client-side web app was made. Afterwards, communication between the different clients was done by connecting the endpoints by using REST calls. These calls made it possible to populate the HTML files dynamically inside the JavaScript file. The dynamic population of the pages is the first form of failure protection as the client will not be able to order a non-existent drink.

Delivery Service

The delivery service is deployed on the virtual machine in South-Africa: dapp.southafricanorth.cloudapp.azure.com:13000/rest/f1e2d3c4-b5a6-7890-1234-56789abcdef0/overviewOfAlldelivery

The delivery service utilizes a Google Distance API key to estimate the delivery distance and time. It checks the longitude and latitude of the delivery request's origin. Based on these coordinates, the appropriate delivery person is selected. Once the delivery address coordinates are obtained, a request is sent to Google's Distance Matrix API to calculate the estimated distance and delivery time. This is done with the help of the Distance Matrix API of Google: <https://developers.google.com/maps/documentation/distance-matrix/overview>. Next to determining the delivery date, the distance is also needed to calculate the total price.



Distance Matrix API

[Google Enterprise API](#)

Travel time and distance for multiple destinations.

MANAGE

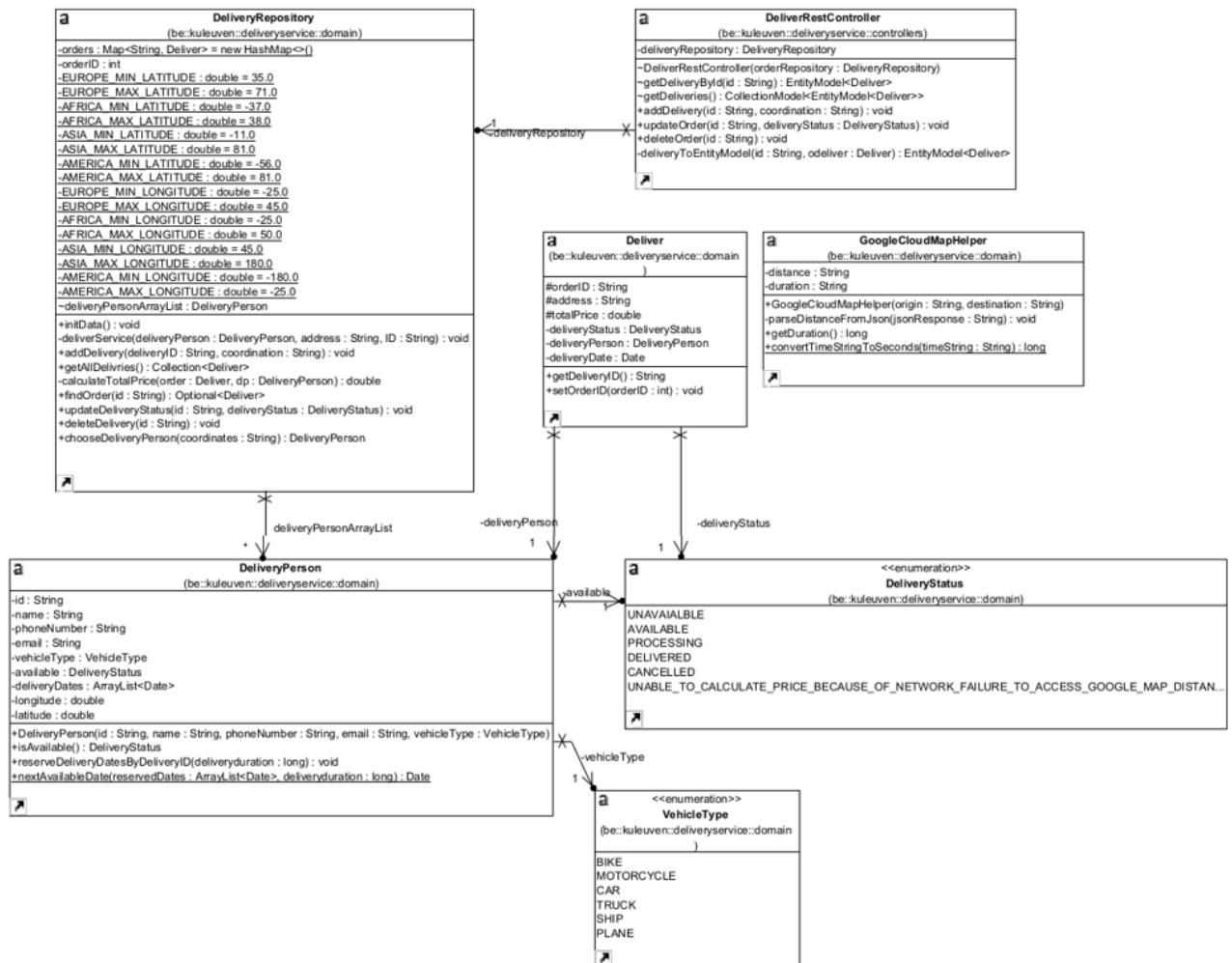
✓ API Enabled

Name	↓ Requests	Errors (%)	Latency, median (ms)	Latency, 95% (ms)
Distance Matrix API	15	0	88	917

If accessing the GoogleMapHelper results in an error, the delivery status will be updated to

`UNABLE_TO_CALCULATE_PRICE_BECAUSE_OF_NETWORK_FAILURE_TO_ACCESS_GOOGLE_MAP_DISTANCE` to notify other services when they read the delivery status.

This is the UML diagram of our delivery service:



ACID Properties

- Atomicity:

When the user logs in, an inventory is made in Firestore based on the provided inventories from the suppliers. In our case, these two suppliers are the liquor supplier and the wine supplier. When the user confirms their order, the quantity of each order product is checked against the loaded inventory. When a user orders more of a product than is available in the inventory for both liquor and wine, one or the other or both, the order is cancelled. The user will see an error that there is not enough in stock and that the order cannot proceed. The users will be redirected again to the order page to recommence their order.

Authentication

The authentication is only checked when logging in. When a non-existing user tries to log in this will not be possible, the new user first needs to create a new account that will be stored in the Firestore. Afterwards, the new user will be stored inside the Firestore and the user will need to log in. When an existing user logs in there will be a check if the user is a manager or a normal user. When a normal user logs in, he/she will be redirected to the order page, when a manager logs in, he/she will be redirected to the manager page where all the processed / pending deliveries are shown afterwards the manager can also go to the order page. As this is a one-page application it suffices to only check the authentication when a person first loads the page as they won't be able to display any other page if they did not pass the first authentication of the login.

Name	Tasks	Hours contributed
Jens	Liquor-SOAP-client, end-point communication web server – liquor client, atomicity	>55 hours
Koenraad	Web client (JavaScript and HTML), communication web server – wine soap client, endpoint communication web server – delivery client, order storage to Firestore	60+
Remco	Wine-SOAP-client, authentication, SOAP endpoint communication, Second version + runscript	±70
Samih	Delivery-REST-client + endpoint communication web server – delivery client, authentication, Second version + runscript	±70