KU LEUVEN

FACULTY OF ENGINEERING TECHNOLOGY

GROUP T LEUVEN CAMPUS

Final Project

Service-Oriented Distributed Applications on a Cloud Platform

16/06/2024

Ivan BADENHORST (r0821191)
Maria KAREGLA (r0825039)
Sudarshan Raghavan IYENGAR (r0826532)
Benedicta Marietta AMANDA (r0823860)

Academic year 2023-2024

1 INTRODUCTION

This report presents our distributed web application, a broker platform that connects customers with curated product bundles from independent suppliers, implementing Level 1 and Level 2 of the assignment.

This report will explore our system architecture, access control mechanisms, transaction handling, data model, and cloud integration strategies. We'll also discuss the challenges of developing and deploying a distributed system on a cloud platform and address potential migration considerations.

2 SYSTEM ARCHITECTURE

The designed system architecture, has 4 major components:

- Suppliers
- Broker
- Firebase
- Firestore

In the following sections, we will look at the system architecture and deployment of each of these components individually. We will see how they are deployed, their roles within the system, and their interactions with one another. Finally, we will present a comprehensive deployment diagram that illustrates the interconnections and overall structure of the entire system.

2.1 Suppliers

The supplier components in the system acts as webstores that provide REST services for other businesses (like our broker platform) to interact with their inventory and purchase items programmatically. This enables integration between the supplier's inventory and the purchasing systems of other stores.

Each supplier service is implemented as a Spring Boot application, packaged as a JAR file, and deployed on a dedicated Microsoft Azure Virtual Machine (VM). These VMs are distributed across three geographic locations: Canada, Japan, and Switzerland. Each of these webstores forms a node in the deployment diagram. One example is shown in the following figure.

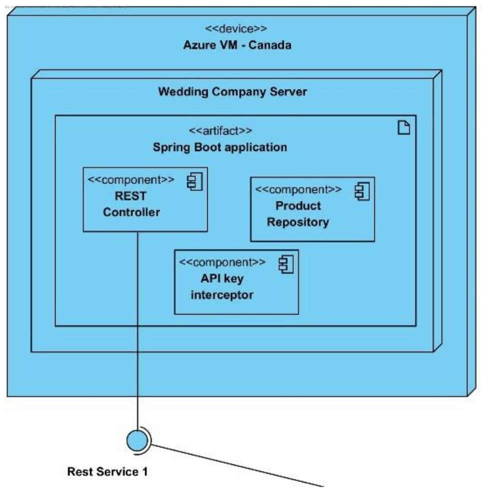

Figure 1: Supplier Node

To secure the supplier services and prevent unauthorized access, each API endpoint is protected with an API key. This is used to limit access to the webstore only to those with a valid API key. The products are stored in memory in the Product Repository and a REST controller is used to provide the REST service.

The REST API exposed by each supplier service follows the principles of HATEOAS (Hypermedia as the Engine of Application State). This means that clients can discover available endpoints and actions by following links provided in the API responses. The RestController class provides the following endpoints for managing products and reservations:

- **`/products` (GET):** Returns a collection of all available products, each represented as a JSON object. The response includes HATEOAS links to individual product resources (`/products/{id}`) and to the reserve endpoint for creating reservations.
- **`/products/{id}` (GET):** Retrieves a specific product identified by its unique ID. The response includes HATEOAS links similar to the /products endpoint.
- **`/products/reserve` (POST):** Allows clients to create a reservation for one or more products. The request body specifies the product IDs and quantities to be reserved. The response returns the created reservation object, including HATEOAS links to retrieve the reservation (`/reservations/{id}`), cancel the reservation (`/reservations/{id}/cancel`), and confirm the reservation (`/reservations/{id}/confirm`).
- **`/reservations/{id}` (GET):** Retrieves a specific reservation by its unique ID. The response includes HATEOAS links to relevant actions (cancel or confirm) based on the current status of the reservation.
- **`/reservations/{id}/cancel` (POST):** Cancels an existing reservation. The response includes the updated reservation, now with a CANCELLED status.
- **`/reservations/{id}/confirm` (POST):** Confirms an existing reservation. The response includes the updated reservation, now with a CONFIRMED status.

2.2 Broker

The broker component acts as an intermediary between the supplier services and the end-clients. It provides two main functionalities:

- **Webstore for End-Clients:** The broker offers a webstore user interface for end-clients to browse and purchase product bundles offered by the suppliers. This webstore is a single-page application (SPA) implemented with JavaScript. The SPA interacts with the broker through REST calls to retrieve product information and complete purchases.
- **Management Functionality:** The broker also provides a management interface for administrators to manage product bundles, view customer information, and monitor orders. Similar to the webstore, this functionality is implemented through a web interface and interacts with the broker using REST calls.

The broker is a Spring Boot application. It uses Firebase to manage user authentication and authorization for both the webstore and the management functionality. It further uses Firestore to store data such as product bundles, customer information, and order details. This deployment choice allows for scalability and automatic management of the broker's infrastructure. In the figure below, the broker node is shown.

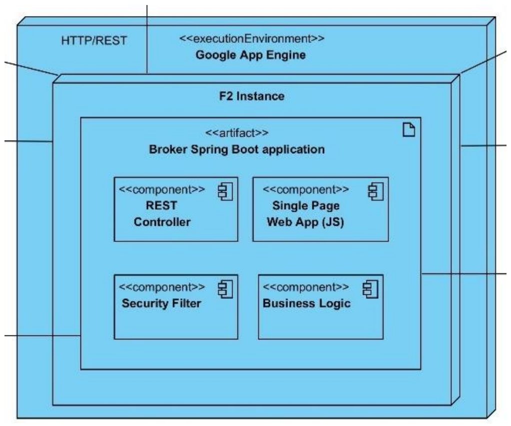

Figure 2: Broker node

As seen above, the broker application is deployed on Google App Engine (GAE) on an F2 instance.

2.3 Deployment diagram

The full deployment diagram is shown below.

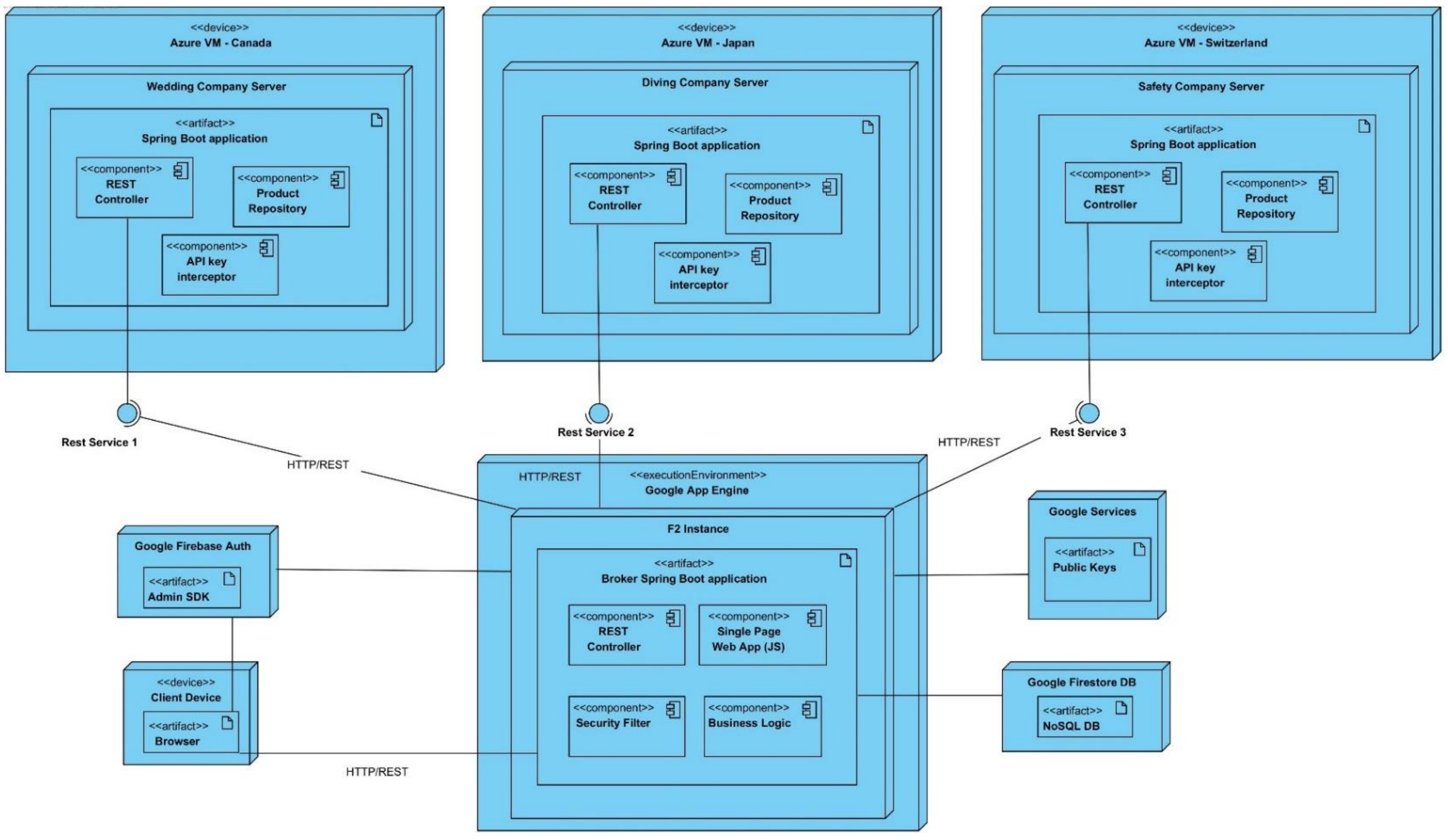

Figure 3: Full deployment diagram

There are 3 suppliers, each providing a REST API. The broker application relies on these APIs. The broker application is provided to the client device, and updates information through REST calls. The broker further contacts Google Services in order to receive public keys for token verification. Finally, it connects to Firestore and Firebase as previously described.

3 Access Control

3.1 JWT Verification

JSON Web Tokens (JWTs) form the backbone of our authentication and authorization system. They are digitally signed, compact, and self-contained tokens used to securely transmit information between parties. Here's a breakdown of the JWT lifecycle within our system:

3.1.1 JWT Issuance

1. **User Authentication:** When a user logs in using their email and password, the Firebase client-side SDK handles the authentication process.
2. **Token Generation:** Upon successful authentication, Firebase generates a JWT for the user. This JWT contains the user's email address and any additional claims associated with their account, including the crucial "roles" claim for role-based access control.

3.1.2 Token Usage

1. **Client-Side Storage:** The client (browser) receives the JWT from Firebase and stores it locally, typically in local storage or session storage.
2. **Including the Token in Requests:** For every subsequent request to the protected API endpoints on the broker platform, the client includes the JWT in the Authorization header of the HTTP request. This header typically follows the format: `Authorization: Bearer {JWT}`.

3.1.3 Token Verification

1. **Security Filter:** On the server-side, the `SecurityFilter` component intercepts all incoming requests to protected endpoints which are defined in the `WebSecurityConfig` class. More information on the `WebSecurityConfig` class' handling of role-based endpoint authentication is presented in Section 3.2.2.
2. **Dynamic Public Key Retrieval:** Since the JWT is signed by Firebase using asymmetric cryptography, the `SecurityFilter` needs to retrieve the corresponding public key to verify the signature. This public key is not hardcoded. Instead, it is fetched dynamically from Google's public key endpoint.

3. **Signature Verification:** The `SecurityFilter` uses the Auth0 Java JWT library, employing the RSA256 algorithm, to verify the JWT signature against the fetched public key. This verification ensures that the JWT has not been tampered with and originated from a trusted source (Firebase). If the verification fails, the request is immediately rejected with a 401 Unauthorized response.

4. **Decoding JWT and Context Setup:** After successful verification, the JWT is decoded, extracting the user's email address and roles from the "email" and "roles" claim. This information is used to create a `User` object representing the authenticated user. If the "roles" claim contains the value "manager", a `SimpleGrantedAuthority` object with the authority "manager" is created.

   The `SecurityFilter` then creates a `FirebaseAuthentication` object, which encapsulates the user's details and authorities, and sets this object into the Spring Security context. This step makes the user's authentication and authorization information available to other components of the application, especially the controllers.

3.2 Role-Based Access Control

Authorization, which determines what actions a user can perform, is implemented in our system using Role-Based Access Control (RBAC).

3.2.1 Custom Claims and Role Assignment

In addition to the "email" claim, the JWT issued by Firebase can contain custom claims. In Level 1, these claims are set using the Firebase emulator UI. However, in Level 2, set custom claims programmatically using the Firebase Admin SDK. The `SecurityFilter` is then able to extracts this claim during JWT decoding.

The value "manager" within the "roles" claim signifies that the user has administrative privileges. This role is programmatically assigned to specific email addresses during application startup in `Dsgt4Application.java`. Moreover, as mentioned previously in Section 3.1.3, the `FirebaseAuthentication` class maps the "roles" claim to a Spring Security authority. If the "roles" claim is "manager", the user is granted the "manager" authority, which Spring Security uses to make authorization decisions.

3.2.2 Role-Based Endpoint Authorization

The `WebSecurityConfig` class uses Spring Security's `antMatchers` method to specify URL patterns and the hasAuthority method to define the required authority for accessing those URLs. Our application has specific endpoints that require the "manager" authority:

- `/api/getAllCustomers`
- `/api/getAllOrders`
- `/api/addBundle`
- `/api/updateBundle`
- `/api/deleteBundle/**`
- `/api/getProducts`

The WebSecurityConfig explicitly defines these endpoints and enforces that only requests from users with the "manager" authority can access them. When a request reaches any of these protected endpoints listed abiove, Spring Security checks the user's authorities against the authorization rules defined using the `@PreAuthorize` annotation, where `@PreAuthorize("hasRole('manager')")` will only allow requests from users who have the "manager" authority in the Spring Security context. If the user has the necessary authority, the request is allowed to proceed; otherwise, a 403 Forbidden error is returned.

For all other API endpoints under the `/api/**` path, authentication is required, but the specific "manager" role is not mandatory. This configuration ensures that any logged-in user can access these endpoints.

3.3 API key interception for Suppliers

These API endpoints of the suppliers are protected with an API key to prevent outsiders from accessing these endpoints. Only known brokers can access these API endpoints. For simplicity, each supplier service in our system has a hardcoded API key defined in the `ApiKeyInterceptor` class. The `ApiKeyInterceptor` component acts as a request interceptor, examining incoming HTTP requests *before* they reach the controller methods. The interceptor specifically checks for the presence of the API key in the Authorization header of the request. The expected format is `Authorization: Bearer {API_KEY}.`

If the Authorization header contains the correct API key, the request is allowed to proceed. However, if the header is missing or contains an invalid API key, the interceptor immediately rejects the request with a 401 Unauthorized response. This mechanism effectively blocks any unauthorized access to the supplier services.

4 TRANSACTION

4.1 Level 1

4.1.1 All or nothing Semantics

Our system implements all-or-nothing semantics for composite orders by employing the composite transaction pattern. When a user attempts to purchase a bundle, the following steps occur:

1. **Reservation Requests**: Reservation requests are sent to each relevant supplier's server for each product in the bundle. A successful reservation means that the (specified amount of) products are reserved i.e., temporarily unavailable, to other users.
2. **Failure Detection and Rollback:** If any product reservation fails (due to a supplier server failure, the product being unavailable, or other reasons), the system detects this failure. As a compensating transaction, a request is sent to each supplier to cancel the previously reserved products. This effectively rolls back the entire order.
3. **Purchase Confirmation:** If all products are successfully reserved, a second set of requests is sent to the suppliers to confirm the purchase, making the order permanent.

4.1.2 ACID Properties

Atomicity: The composite transaction pattern ensures atomicity. Either all products in a bundle are reserved and confirmed, or the entire transaction is rolled back, leaving no partial changes.

Consistency: Data consistency is maintained because all data related to the order (status, product reservations, etc.) is updated atomically. If a failure occurs, the system reverts to a consistent state by rolling back the transaction.

Isolation: Concurrent orders are handled by reserving products first, ensuring that no conflicts arise during the reservation phase. This prevents multiple users from reserving the same product simultaneously.

Durability: Cloud Firestore's inherent durability ensures that order data is persisted even in the event of failures. This means that data is written to disk, making it resilient to server crashes or other failures.

4.1.3 Failure Scenarios and Handling

Supplier Server Unavailable

If a supplier server becomes unavailable during the reservation process, the all-or-nothing semantics ensure a rollback of the entire order. However, if the server is unavailable during the purchase confirmation phase (Step 3), we implement a retry mechanism. In this, the confirmation requests are performed in separate threads, with each request being repeated until a successful response is received. This ensures that the purchase is ultimately completed.

Broker platform crash

Each user maintains collections in Firestore representing bundles in their basket, bundles being processed, and bundles that have been purchased. If the broker platform crashes during the reservation phase, the system is in a relatively safe state, as no permanent changes have been made to the supplier's systems. However, if the broker crashes during the purchase confirmation phase (Step 3), the system handles this as follows:

1. On restart, the broker checks the "processing" collection for each user to identify orders that have not yet been confirmed with suppliers.
2. For any unconfirmed orders, the broker sends confirmation requests to the suppliers, completing the purchase of the bundle.

4.2 Level 2

4.2.1 Broker Product Integration

Our system seamlessly integrates broker products with external supplier products. In Level 1, all external products present in active bundles, are stored on our Firestore as seen in Figure 4, to which we now simply inject our own products. We require one more collection of “reservations” which allows us to track the reservation status of our own products, mimicking the structure employed by the external suppliers. To update the status of a locally reserved product, we query our Firestore database instead of using external server endpoints.

Our products are injected onto firestore using the deployed cloud function `initializeFirestore`.

4.2.2 Transactional Behaviour

Expanding upon the Level 1 all-or-nothing semantics, we adjust our strategy to accommodate internal products where we just have to do a local method invocation. When a purchase is made:

1. We first attempt to reserve our own products.
2. Only after successful reservation of our own products do we initiate reservations with external suppliers.
3. This strategy minimizes unnecessary external requests in the event of a rollback, improving efficiency.

5 DATA MODEL

The figure below shows the ERD for the Firestore database.

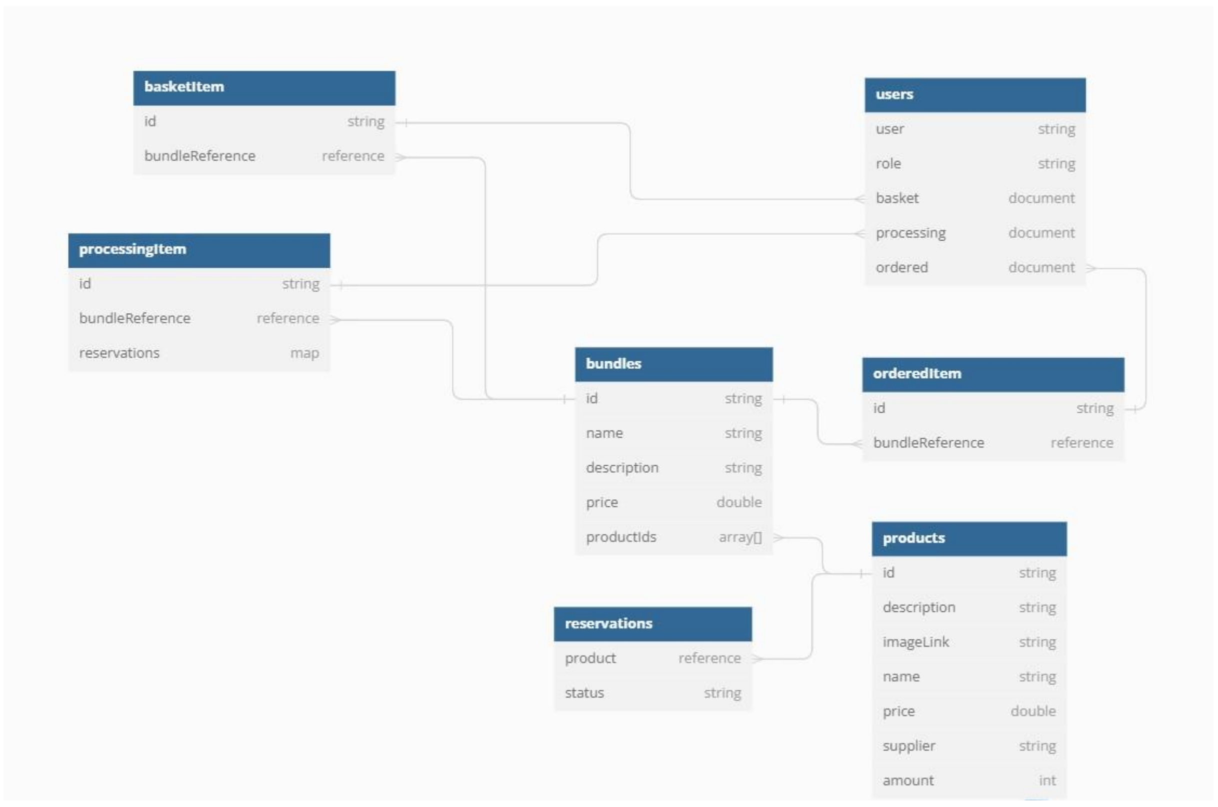

Figure 4: Database ERD

5.1 Data Model Structure

The core of the data model revolves around two main collections: the bundles collection and the user collection. A reservation collection was also added to allow for the management of the broker products implemented at Level 2. These collections will be further discussed in detail in the following sections.

5.1.1 Bundles

A user with the manager role is able to create bundles by combining different products from different suppliers, as discussed earlier in the system architecture. The information of each created bundle was then stored in the database in a document in a collection called “bundles”.

Data on any product that pertains to a bundle is also stored on the Firestore database in the products collection. This decision was driven by several factors.

The first factor is system persistence. Local storage ensures that users can view bundles and their related products at any time, unaffected by the state of the supplier. As such, even in the case of a supplier outage, users will be able to access the site and continue browsing without interruption.

The second factor was network traffic optimization. When a manager is creating a new bundle, a GET request is sent to each supplier to obtain all available products. This is fine, as this process is seldom done by many managers at a single point in time. However, it is highly likely that a large volume of concurrent customer users will be accessing the broker site. As such, constantly querying each supplier for product information on user request would lead to unnecessary network congestion. Essentially, by locally storing relevant product details, the system would avoid overwhelming suppliers with frequent product inquiries during customer browsing.

Only products pertaining to a bundle were stored locally to improve scalability and take into account the dynamic nature of product catalogues. In the case of there being numerous suppliers with each having hundreds of different products, storing all products can quickly become cumbersome. Supplier product offerings are also constantly evolving, with products being added or removed. For example, many fashion brands implement “fast fashion” thus seasonally changing their entire product catalogue. Storing a complete, ever-changing catalogue locally would require continuous updates, potentially causing inefficiencies.

5.1.2 User

The user collection contains a document for each of the authenticated users. Within each user document, the information about their role (manager or customer) is stored, along with sub-collections pertaining to the orders the user has made.

Each user has a dedicated document storing their role (manager or customer) and email address. The user document also contains three nested sub-collections to track their orders at different stages: the basket, processing, and ordered sub-collections.

The basket sub-collection contains documents for bundles a user has added to their shopping cart but not yet bought. Each document references the corresponding bundle document from the "bundles" collection and includes a unique identifier for the cart item.

When a user clicks "buy," the system sends reservation requests to the suppliers for the bundled products. Upon successful transactions, the document representing the cart item is moved from the "basket" sub-collection to the "processing" sub-collection. It remains here until all suppliers confirm product reservations for the complete bundle.

Once all supplier confirmations are received, the document is transferred from the "processing" sub-collection to the "ordered" sub-collection. This indicates a successful purchase, and the order is finalized.

This three-tiered sub-collection structure provides a clear and organized way to track user orders throughout the buying process. This structure also ensures long-term data retention, as information on past orders is never deleted and remains stored in the database. This can be useful for customer support services and be used for customer data analysis to better understand customer behavior and product performance.

5.1.3 Reservations

Level 2 of the project introduces a new complexity, with the broker also acting a supplier. The broker now offers its own products that can be added to a bundle. These products are stored within the same collection as the relevant products from external suppliers. The supplier field allows for the differentiation of the product origin of the products.

The expansion of the broker’s scope now requires an additional function to allow it to manage reservations for its own products. To accommodate this, a new collection named "reservations” was created. Each reserved product has a corresponding document within this collection, containing details about the reservation's state (pending or confirmed).

5.2 Limitations on Queries

5.2.1 Lack of JOIN Queries

Relational databases allow for JOIN queries to establish connections between data in separate tables based on shared attributes. However, this type of query is not supported in Firestore. As such, it was necessary to find an alternative solution to combine data from different collections.

In this particular application, a JOIN would have been particularly useful in relating the “basket" and "bundles" collections. By joining these collections based on the bundle ID, it would be possible to efficiently retrieve all bundle details along with the corresponding cart items in a single query. The same applies to the “products” and “bundles” collections, based on product ID.

As a workaround, a solution utilizing reference fields were implemented. This approach allowed for each cart item to directly reference the corresponding bundle document. By establishing this reference, the complete bundle details can be easily retrieved from within the cart item document, eliminating the need for a JOIN query.

5.2.2 Lack of Exclusionary (NOT) Queries

Relational databases also allow for exclusionary (NOT) queries to retrieve all data that do not meet a specific criterion. This type of query would have been beneficial for filtering during product retrieval. The broker had its own set of products that implemented a different method of processing compared to products from external suppliers. Ideally, the query would retrieve only products associated with regular suppliers.

Since Firestore doesn't support NOT queries, the current solution involved retrieving the "supplier" field for each product and then employing an if-else statement within the application code to manually filter out the broker's products.

6 CLOUD

What are the pitfalls when migrating a locally developed application to a real-world cloud platform? What are the restrictions of Google Cloud Platform in this regard?

Migrating a locally developed application to a real-world cloud platform like Google Cloud Platform (GCP) offers significant advantages, but it's not without its challenges.

One major issue involves the discrepancies between local development and the actual GCP setup. Local development relies on emulators, like the Firebase emulator for users. These emulators can provide a simplified experience compared to the full functionality of GCP services. For instance, the local emulator allows for easy user role management through a GUI, whereas the deployed version requires programmatic changes. This necessitates maintaining a separate development branch that doesn't include these emulator-specific alterations. Additionally, local emulators might not fully replicate all functionalities of the actual GCP service. For example, token verification, which is crucial for a secure deployed application, is not included in the emulator. This means additional code specific to the deployed environment needs to be written.

Further, deploying an application to GCP can be time-consuming. This makes it difficult to test code specific to the deployed environment. Debugging issues that arise during deployment can be more challenging due to the added complexity of the cloud environment compared to a local setup.

How extensive is the tie-in of your application with Google Cloud Platform? Which changes do you deem necessary for migrating to another cloud provider?

Our application leverages Google Cloud Platform (GCP) for several essential services, resulting in a moderate level of tie-in with the platform. Here's a breakdown of our integration and the potential challenges and changes involved in migrating to a different cloud provider:

Google Cloud Platform (GCP) Services:

- **App Engine:** Our primary application, the broker platform, is deployed on Google App Engine (GAE). GAE provides a managed platform for deploying and scaling Java web applications, handling infrastructure management, load balancing, and auto-scaling.

- **Cloud Firestore:** We use Cloud Firestore as our NoSQL database to store bundle information, reservations, etc. Firestore offers a flexible, scalable, and real-time database solution with robust features for data management.
- **Firebase Authentication:** We rely on Firebase Authentication for user authentication and management. Firebase provides a secure and easy-to-integrate authentication system that manages user accounts and generates JWTs for authentication.

We believe our application is *pretty strongly* tied to GCP. Core functionalities like authentication, data storage, and application hosting are heavily reliant on GCP services. However, much of our business logic is platform-agnostic. Moreover, the supplier services, for instance, are deployed on Azure VMs and communicate with the broker platform through standard REST APIs and they could potentially be deployed in.

Migrating to a different cloud provider would require several changes and present certain challenges:

1. **Service Equivalents:** Finding equivalent services on other cloud platforms for App Engine, Cloud Firestore, and Firebase Authentication. While most providers offer comparable services, there might be feature disparities or differences in API implementations.
2. **Code Modifications:** Our code directly interacts with GCP-specific APIs and libraries. Migration would necessitate adapting the code to use the APIs and libraries of the new cloud provider, potentially involving substantial code refactoring.
3. **Deployment Configuration:** Our deployment process and configuration are tailored to App Engine. Moving to another platform would require reconfiguring deployment settings, potentially rewriting deployment scripts, and adapting the application to the new environment's constraints.
4. **Data Migration:** Migrating the data from Cloud Firestore to a different database service. This process might involve data transformations, schema mapping, and ensuring data consistency during the migration.

7 LINKS

Website URL: https://dsgt-little-endian.ew.r.appspot.com

Github link (including supplier code): https://github.com/mariakare/littleEndians/tree/deploy-main

Accounts

Manager account: email = manager@manager.com; password = 123456

User account: email = user@user.com; password = 123456

8 TEAMWORK - EVERYONE

Maria: Broker REST Services, Firestore Modelling and Management, Function Deployment, Broker/Supplier Crash Fault Tolerance **80 hours**

Ivan: Frontend development (JavaScript, CSS, HTML), deployment on GAE and Token verification. **80 hours**

Sudarshan: Supplier REST Services, Tokens, Roles, and Endpoint Control, deployment on GAE. **80 hours**

Benedicta: Broker REST Services, Firestore Modelling and Management, Broker Crash Fault Tolerance. **80 hours**

FACULTEIT INDUSTRIËLE INGENIEURSWETENSCHAPPEN
CAMPUS GROEP T LEUVEN
Andreas Vesaliusstraat 13
3000 LEUVEN, België
tel. + 32 16 30 10 30
iiw.groept@kuleuven.be
www.iiw.kuleuven.be