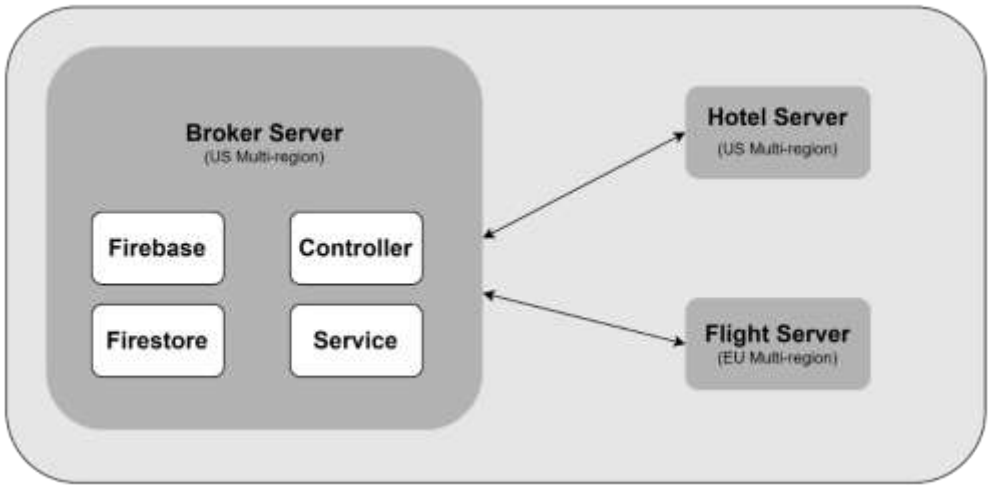
Overall Architecture



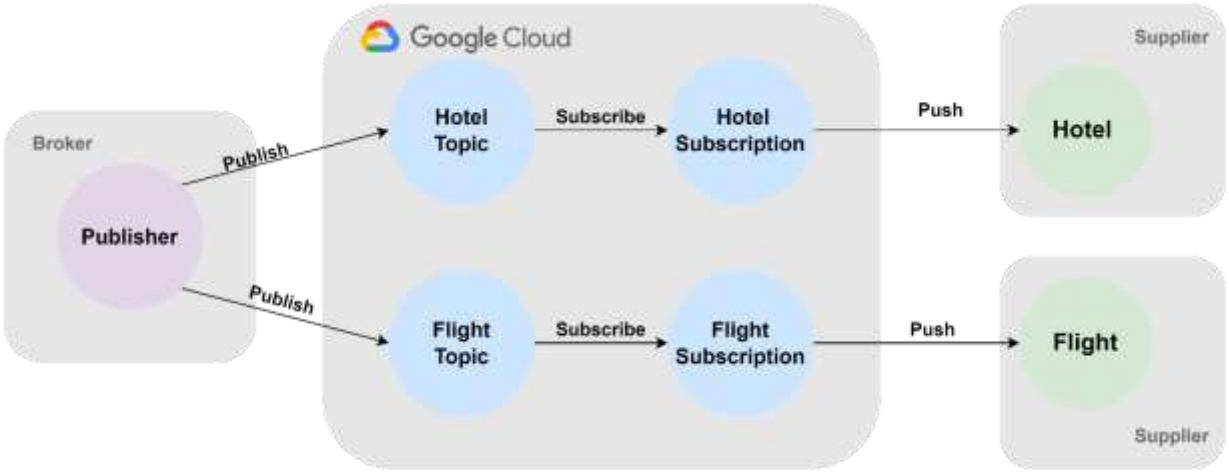
Our distributed system architecture consists of a Broker Server, Hotel Server, and Flight Server. The Broker Server, deployed in the US multi-region, handles authentication with Firebase, stores data in Firestore, and manages API requests and business logic. The Hotel Server (US multi-region) and Flight Server (EU multi-region) provide localized booking services.

The Broker Server acts as the coordinator, communicating with Hotel and Flight suppliers via Google Cloud Pub/Sub for reliable message delivery. It handles operations such as pushing requests, confirming actions, committing transactions, and rolling back if necessary, ensuring robust and scalable transaction management across regions.

A brief summary of the different remote interfaces and their operations

Endpoint	Method	Parameters	Description
/hotels/test	POST	None	Simple test function to validate subscriber function.
/hotels/all	GET	key (query param)	Retrieve a list of all hotels with HATEOAS links.
/hotels/{id}	GET	id (path), key (query param)	Retrieve detailed info about a specific hotel by ID.
/hotels/pubsub/push	POST	None	Receive and process a Pub/Sub message for hotel booking.
/hotels/commit/{id}/{rooms}	POST	id (path), rooms (path)	Commit a hotel booking.
/hotels/rollback/{id}/{rooms}	POST	id (path), rooms (path)	Rollback a hotel booking.

Distributed B2B communication: Pub/Sub

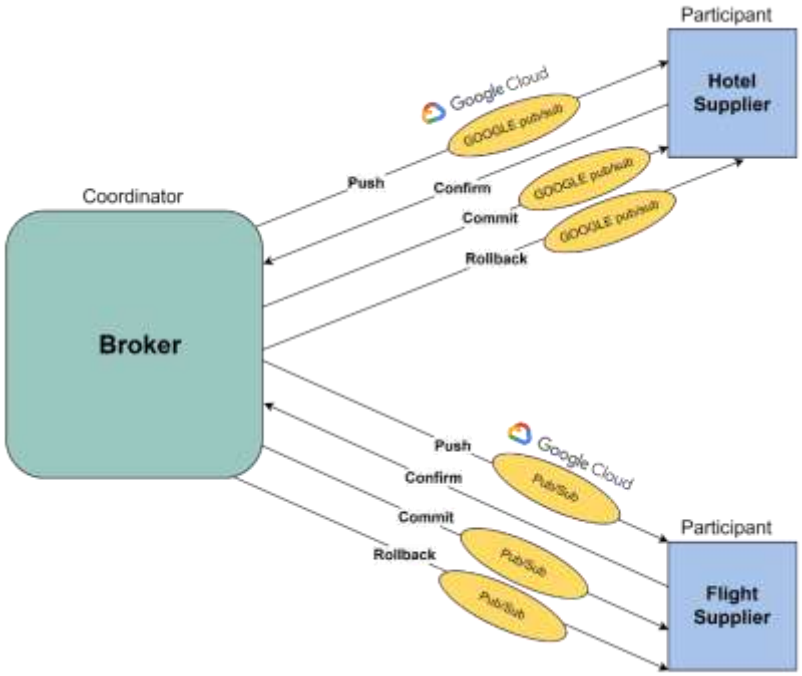


In our B2B communication setup:

- Broker to Supplier: Using **WebClient** or **Pub/Sub**:
  The broker can initiate orders and manages 2PC coordination by using **WebClient (Level 1)** to communicate the suppliers or using **Pub/Sub (Level 2)** publishing messages and transaction signals to topics.
  In **Pub/Sub**, messages are pushed asynchronously to suppliers with **at-least-once-semantics**, ensuring reliable delivery with retry mechanisms. We set the acknowledgement deadline for 100s with a retention duration of 1 day. This way, we can **cope with failures** of the external supplier services, in case where the supplier fails to response, it will not cause a total failure of the broker platform.

- Supplier to Server: Using **WebClient**:
  Suppliers confirm orders and communicate transaction vote (commit or abort) back to the broker using WebClient.

Distributed atomic transaction: Two-phase commit



We use two-phase commit for the transaction behavior:

1. The broker first sends the order details through **Pub/Sub** to initialize the transaction.
2. Upon receiving the message, the participants make **REST request** to Broker to confirm the transaction
3. Upon receiving the confirmation, broker update the database, once all components are confirmed, it will again send **commit** messages through Pub/Sub
4. If any error happens, like the message is not handled and expired, the broker sends **rollback** messages through Pub/Sub.

Transactional behavior for ACID properties

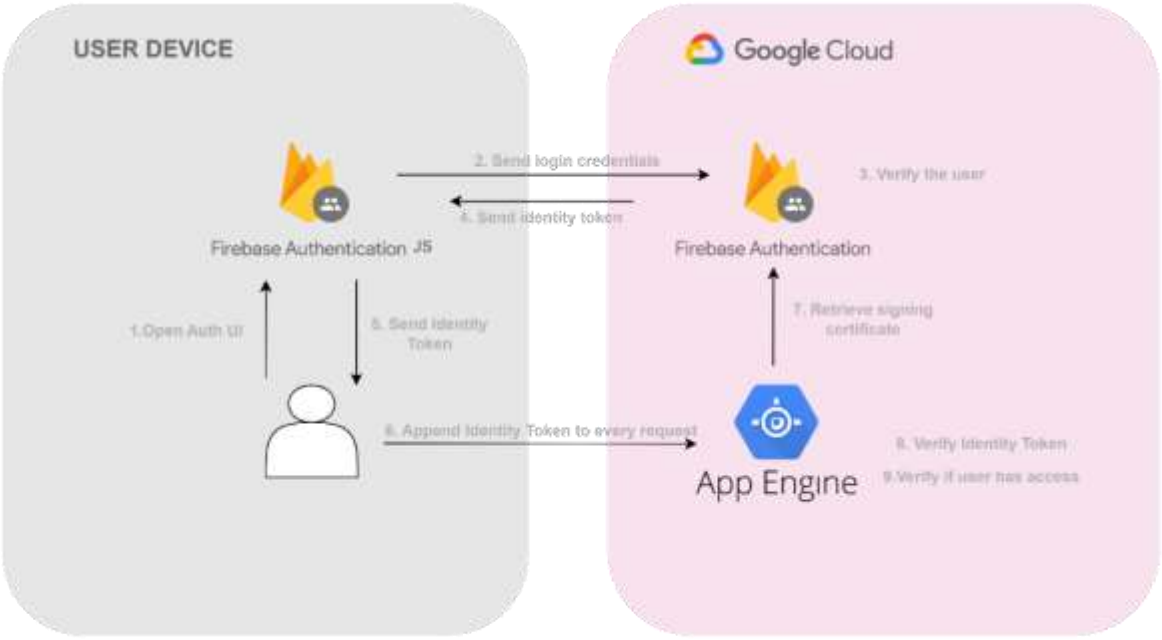
With the 2PC:

1. If part of the order fails, the broker would roll back the changes that has happened, which ensures atomicity and consistency
2. The suppliers use the Concurrent HashMap to store the data, while the functions are using 'synchronized' keywords, so isolation is ensured.
3. After reserving or successfully ordering a package, the data are always stored in the cloud fire store, which ensures durability

However:

1. The communication for rollback may fail, which violates atomicity.
2. Network issues or service disruptions could affect the ability to persist data reliably.

User Authentication



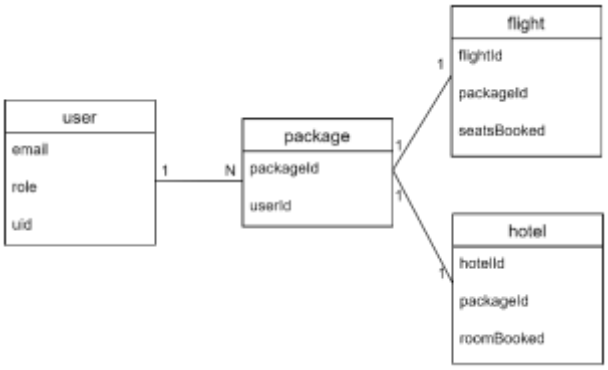
In user authentication, we follow the standard authentication flow. The **security filter** is used to verify the token, and a **JwkProvider** is used to dynamically request the public keys from the endpoint provided by Google.

Implementation to enforce and implement access control such that only authorized users can access manager methods:

1. In “WebSecurityConfig”,“.antMatchers("/api/**").authenticated()” ensures any request to endpoints under “/api/**” requires the user to be authenticated.
2. “@EnableGlobalMethodSecurity(prePostEnabled = true)” annotation enables method-level security
3. After sign in, in “FirebaseAuthentication” class, the getAuthorities method checks if the user is a manager by calling user.isManager(). If true, it assigns the ROLE_MANAGER authority to the user.
4. In the controller, “@PreAuthorize("hasAuthority('ROLE_MANAGER')")” annotation is added for the /api/getAllOrders and /api/getAllCustomers, so only managers can access, while users are forbidden

Firestore Database

Structure of data models



Our data model consists of 4 data models as shown above, This structure efficiently captures the relationships between users, their travel packages, and the individual flight and hotel bookings within those packages.

User: Contains **email**, **role**, and **uid**. A user can have multiple packages, establishing a one-to-many relationship.

Package: Contains **packageId** and **userId**. It links to both flights and hotels, each in a one-to-many relationship.

Flight: Contains **flightId**, **packageId**, and **seatsBooked**. Each package can include multiple flights.

Hotel: Contains **hotelId**, **packageId**, and **roomsBooked**. Each package can also include multiple hotels.

Query limitations using Cloud Firestore

Compared to a relational database, Cloud Firestore has limitations like the lack of complex join operations, constraints in performing aggregate functions (e.g., COUNT, SUM), and limited support for complex transactional queries. This can make querying interconnected data more challenging and may require denormalization or multiple queries to retrieve related data.

Business Logic

The main function of our broker is to allow users to order air tickets and hotels together in one travel package, which corresponds to two different suppliers respectively, and then the user can package the two together to order.

For flights, the user can select the **destination** and **date**, then they can select one flight from an available list, then user needs provide the required **number of passengers** and the **passenger names**. For hotels,

the user can choose the **destination** and check-in **date**, then they can choose one from all available hotels, after that user needs to provide **number of people** and **number of nays** for this hotel.

Deployment

Challenges when migrating a locally developed application to a real-world cloud platform:

- Hardcoded configurations

Specifically, server port numbers for different supplier services, which were specified in the *application. properties* file had to be reconfigured. To address this, we removed these hardcoded settings from the *application.properties* file and implemented dynamic configuration management to accommodate the cloud environment.

- Firebase authentication
    - The *firebasedmin-sdk.json* should be securely stored using Google Secret Manager, otherwise, it may be exposed to unauthorized access.
    - A better practice we actually implemented in the end is dynamically requesting public keys, which complies with OAuth 2.0 and OpenID Connect standards, to enhance security in cloud environments.
- Our services as separate App Engine projects

By deploying each service as a separate App Engine project, we avoided the complexities and potential issues of a single project configuration.

Google App Engine Advantages

- It automatically creates buckets in Google Cloud Storage to store the application, and these buckets can be **multi-regional**, providing higher availability than regional storage classes. This ensures higher availability and data redundancy across regions asynchronously, enhancing failure prevention and data durability.
- Google Cloud App Engine offers default **auto-scaling**, adjusting the number of instances based on the application load.

Google Cloud tie-in

Our application is heavily integrated with Google Cloud Platform, utilizing several of its key services. We use Google App Engine for deploying our services, Google Cloud Storage for storing application data in multi-regional buckets and Firebase for authentication and real-time database services. The dynamic configuration management and auto-scaling features provided by GCP are critical to our application's performance and scalability. However, we assume that migrating to another cloud provider would present a significant configuration problem rather than a solution/architectural design problem.

App Engine URL and example emails

App engine URL:

https://broker-da44b.uc.r.appspot.com/

User example:

Email: wearegonnawin@gmail.com

Password: 666666

Manager example:

Email: manager@test.com

Password: 1972800290

Task distribution

Name	Task	Time
Geng	Firebase Authentication B2B communication Two Phase Commit – supplier side	60
Jeffee	Firestore Database B2B communication Two Phase Commit – broker side	60
Ivan	Front-end Deployment Business Logic – supplier side	60
Jiaao	Front-end Business Logic – broker side	60