06/06/23

Distributed Applications

Service-oriented distributed applications on a cloud platform

Arent Steeno (r0811736)
Hussain Hammad (r0767612)
Sofie Delaet (r0777254)
Vasu Kapoor (r0772430)
TEAM 20

Project ID = distributed-apps
Environmental variable: GOOGLE_APPLICATION_CREDENTIALS=/[path to /resources/serviceAccountKey.json]

Deploying on the cloud works without an environmental variable. However, to run it locally it doesn't. In the code, you will see uncommented sections in which we tried to solve this, we got close but weren't able to do it in time.

Level 1: Basic Requirements

Security: Firebase authentication

Q: How does role-based access control simplify the requirement that only authorized users can access manager methods?

A: The role-based access control defines all roles in the same location as their permissions. This means that when a user is given a certain role, they also automatically gain access to the permissions that go with it. No additional authorization checks are required and the manager methods do not need to take care of it anymore, simplifying the code. Manager methods can simply focus on what and how without having to first look at the who. Furthermore, this also simplifies and speeds up the process of changing the role of an already existing user.

Q: Which components would need to change if you switch to another authentication provider? Where may such a change make it more/less difficult to correctly enforce access control rules, and what would an authentication provider therefore ideally provide?

A: Components that would need to change are the ones relating to making the connection to the authentication provider as they may have their own unique APIs, SDKs, libraries, or protocols for authentication. Since, JWTs are a widely adopted industry standard in authentication and authorization workflows, the probability this can be kept while switching to a new authentication provider is high. However, just like it was needed to do in Firebase, claims for the roles will have to be defined as well to ensure all users get associated with the correct permissions.

Non-functional requirements: Fault tolerance

Q: How does your application cope with failures of the Unreliable Airline? How severely faulty may that airline become before there is a significant impact on the functionality of your application?

A: The code implements fault tolerance mechanisms using Spring Retry, which allows for retrying failed operations. It employs the retry strategy for certain methods that interact with the Unreliable Airline. The @Retryable annotation is used to specify the conditions for retrying, such as the exception types to consider and the maximum number of attempts. The retry attempts are triggered in the event of specific exceptions, such as WebClientResponseException, indicating a failure in communication with the Unreliable Airline. By utilizing the @Retryable annotation and specifying the maximum number of attempts, along with an optional backoff delay between attempts, the application can mitigate the impact of intermittent errors from the Unreliable Airline. The retry strategy helps to handle transient failures and improves the chances of successful communication with the airline. Additionally, a fallback mechanism is implemented using the @Recover annotation, which provides alternative behavior or default values when retries are exhausted or failures occur. The recovery methods return alternative values or perform appropriate error handling to ensure that the application remains functional even in the presence of failures.

The application in its current state handles failures of the Unreliable Airline by incorporating retry mechanisms for server 500 errors and WebClientResponseException. However, it's important to note that the code does not handle all possible errors or exceptions that may occur during the interaction with the Unreliable Airline. While it specifically deals with server 500 errors and WebClientResponseException, there may be other types of errors that can occur, such as network errors, rate limiting/throttling, invalid/malformed responses, or authentication/authorization errors, which the code does not explicitly address. The severity of the impact on the functionality of the application depends on the specific nature and frequency of failures exhibited by the Unreliable Airline. If the failures are limited to occasional server 500 errors or WebClientResponseExceptions, the retry mechanism provided in the code can help mitigate the impact and ensure a reasonable level of fault tolerance. The application will still be able to handle other requests and maintain its core functionality.

However, if the Unreliable Airline experiences severe and persistent faults beyond what the code currently handles, the functionality of the application may be more significantly impacted. In such cases, the code may need to be enhanced with additional error handling and recovery strategies to address the specific error scenarios exhibited by the Unreliable Airline.

ACID properties

Q: Is there a scenario in which your implementation of ACID properties may lead to double bookings of one seat?

A: To implement ACID properties we started by modifying the confirm quotes method in the rest rcp controller class making it an 'all or none' system. So, if one of the quotes that need to be confirmed falls through then none of them are confirmed.

In the booking manager class transactions were used for adding and getting bookings. We also added a change log which is stored on the database and logs the interactions done with the database.

Despite this there could still be a scenario in which we have a double booking. If multiple concurrent requests to the confirm Quotes endpoint are made, and each request triggers a separate call to the add Booking method. Since the calls to add Booking happen in parallel, the transactions are executed concurrently, and there is a possibility that two or more transactions check for the existence of the booking document simultaneously and mistakenly find it non-existent due to the parallel execution.

One way to fix this would be to use Zookeeper to have distributed locking.

Level 2: Advanced Requirements

Q: How have you structured your data model (i.e. the entities and their relationships), and why in this way?

A: We created an internal flights collection that has documents for each destination. The document takes the name of the destination and contains 5 fields and 1 subcollection. The fields are the ones from the json file: image path, the location (flight origin and destination), and the name; along with an airline and flightID field that are made for that flight so it is compatible with the data from the reliable airline.

The collection is a collection of the seats on that specific flight. It contains documents that are named after the seat. The document would then contain the name of the seat, price, seatID, time, and type.

Q: Compared to a relational database, what sort of query limitations have you faced when using the Cloud Firestore?

A: Since there are no Join operations between collections then you need to perform multiple queries to get what you want. An example would be getting flight times which are stored as a field in a document in the seats collection which is a subcollection in another document for flights. In a relational database you would do a single query and

link the primary key of the flight as foreign key in the seats table. However here you need to query multiple times to get the information. If you wish to perform a single query, then you would need to denormalize the data by duplicating relevant information across multiple documents or collections. This leads to increased storage size and extra code to maintain consistency.

Q: 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?

A: Firstly, it is much easier to debug and test code locally during development and early production. Especially given the powerful tools provided by modern IDEs.

Migrating large programs and their respective local databases to the cloud can be slow and difficult. All information must be correctly transferred, and the program must perform as expected.

Developed applications may rely on several dependencies available in the local environment but don't perform as expected when running on the cloud.

Something essential to take into account when migrating to a platform such as Google Cloud is their pricing. The costs linked to data usage and storage will undoubtedly be an essential point of discussion during the project.

As seen with our Google credentials authentication implementation, a cloud platform comes with unique security models and requirements. These have to be taken into account when working with role-based access systems.

Q: 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?

A: There are several changes necessary when migrating to another cloud provider.

Firstly, changes related to the authentication (as answered in the first questions) and security. Authentication is used for sign-ups, logins, API request interaction authentication, role assignment and every single transaction. In other words, all of the above-mentioned features will have to be revisited when switching cloud providers to ensure compatibility

Secondly, changes related to the database implementation. This has been done in FireStore. All the application's bookings are stored and managed using Firestore, which is part of the Google Cloud Platform.

In summary, when switching to a new cloud provider, the code must be altered at some locations. All calls to the database and authentication functionality would have to be substituted for the new platform's implementations.

Q: Include the App Engine URL where your application is deployed. Provide a login and password for the teaching team and mention it in your report. Make sure that you keep your application deployed until after you have received your grades.

A:

App Engine URL: https://distributed-apps.ew.r.appspot.com

Email for login: exampleprofile@gmail.com

Password: PASS8989

Roles and Tasks

Arent

3.1.1 Functional requirements: business logic – Arent and Sofie worked together on this section. Provided the basic functionality for the booking website. This included dissecting the relevant API structure for booking the flights, making calls to the API, and providing functions for all different methods together with a local database. (8 hours)

3.1.2 Security: Firebase Authentication – Arent and Sofie worked together on this section. Provided a very basic validation structure. Decoded the given certificate, created a new User instance based on the promises and executed the correct actions based on its assigned role. (4 hours)

3.2.2 Deployment to Google App Engine – Arent and Sofie worked together on this section. Made the necessary changes to the program to refer to the Google Cloud Platform. Created the correct web app on the Google Dashboard. Were successful in deploying the program. (3 hours)

Changed the environmental variable to a . JSON file. 30 min

3.2.3 Security: Firebase Authentication (extended) – Extended the Firebase authentication with validation. Fetched the relevant Google certificates and decoded them into PublicKeys. PublicKeys are stored in a hashmap, from which the appropriate id is fetched and compared to. (6 hours)

Hussain

3.1.4 - Adding bookings to firestore as well as getting bookings from firestore. Including deserialization of data to be able to create the objects in the code. Incorporating the firestore database with the rest of the methods in the booking manager class. (12 hours)

3.1.5 - Atomicity of confirming quotes with an all or nothing approach. Transaction when adding and getting booking to and from firestore. Creating a change log. (6 hours)

3.2.1 - Parsing the data from the json file and adding it to firestore. Worked on extending the APIs to also receive data from firestore along with the webclient but was unsuccessful. (5 hours)

Total time spent: 23 hours

Sofie

3.1.1 Functional requirements: business logic – Arent and Sofie worked together on this section. Provided the basic functionality for the booking website. This included dissecting the relevant API structure for booking the flights, making calls to the API, and providing functions for all different methods together with a local database. (8 hours)

3.1.2 Security: Firebase Authentication – Arent and Sofie worked together on this section. Provided a very basic validation structure. Decoded the given certificate, created a new User instance based on the promises and executed the correct actions based on its assigned role. (4 hours)

3.2.2 Deployment to Google App Engine – Arent and Sofie worked together on this section. Made the necessary changes to the program to refer to the Google Cloud Platform. Created the correct web app on the Google Dashboard. Were successful in deploying the program. (3 hours)

3.2.3 Security: Firebase Authentication (extended) – Extended the Firebase authentication with validation. Fetched the relevant Google certificates and decoded them into PublicKeys. PublicKeys are stored in a hashmap, from which the appropriate id is fetched and compared to. (2 hours)

Vasu

3.1.4 Tried to add and get bookings from firestore with Hussain but my approach didn't work properly so he deserves full credit for this part. (3 hours)

3.1.3 Modified the system to be more fault tolerant so that system can handle ‘server 500’ failures appropriately and not crash due to them. (12 hours)

Total time spent: 15 hours