Distributed Systems (H0N08A)

Report: Project-part2

Conor Finlay *(r0977191)*
Mikkel Skovdal *(r0676230)*

December 8, 2023

How does using a NoSQL database, such as Cloud Firestore, affect scalability and availability?

In short, using a NoSQL database, such as Cloud Firestore, gives our application ease of scalability and high availability.
Cloud Firestore automatically scales data storage according to the needs of the application, maintaining the same (or near the same) query time regardless of the size of the database. As such, this allows us to be confident about the scalability of our platform, particularly where data management and retrieval are concerned.
Cloud Firestore abstracts the distribution and replication of your data, but this does not mean that it is not being replicated. Cloud Firestore replicates data over multiple regions, which ensures that your data is consistently available in many locations, as well as in the event of a disaster affecting one or more data centres.

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

Our database has 3 top-level collections: `trains`, `bookings` and, sporadically, `booking_retries`.

All data that is train specific (i.e. when categorised, can be associated with one train or another) will have the `trains` collection as its root. This top-level collection contains a list of train documents (identified by their trainID), which in turn contain the following collections: `available_seats`, `booked_seats`, `tickets`, and `times`. When a ticket is issued to a customer, a ticket document is created within the `tickets` collection and the corresponding seat is moved from `available_seats` to `booked_seats`. The latter step simplifies and speeds up the fetching of available seats to display to the user when selecting seats to add to the cart. If this were not implemented, the application would have to query the `tickets` collection for every seat on the train to determine whether the seat has been booked. Our implementation causes this action to take constant time (1 query) instead of linear time (1 query + 1 query for each seat). The `times` collection stores all the times available for the train in question, each time being stored as a separate document. While information about these times is contained within the seat objects stored elsewhere, storing the times like this allows for all the seat objects to be parsed for unique "time" values (many of them, of course, will have the same value for "time") only once - when the train data in question is first initialised - rather than every time the application requests a list of the available times. This process takes some significant time, so it makes sense to do this only once.

All bookings are stored as documents within the top-level `bookings` collection. We keep these separate from the trains collection, because bookings do not belong to any particular train but instead contain tickets for any combination of trains in the system.

`booking_retries` is the third top-level collection and keeps track of how many times a PubSub worker has attempted to make a booking on behalf of a particular customer. Without this, the

PubSub worker endlessly retries failed bookings, even in cases where the tickets will never become available (this is assuming failure to book returns an error response). Instead, we keep track of how many times the PubSub has retried in the database and allow a maximum of 3 retries (to provide for crash tolerance/deal with transient errors). If the PubSub worker fails, a booking will not be created, so it doesn't make sense to place `booking_retries` within the top-level `booking` collection, so we establish it as its top-level collection.

A small final point: our data model prepares for there to be many trains on the system (our top-level `trains` stores many different train documents, which in turn store multiple collections relating to the train in question) even though, in this implementation, there is only one train on the system. Our data model is thus prepared for scaling in the application.

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

Cloud Firestore imposes limitations on complex querying, sorting, and data filtering. It does not facilitate JOIN operations or full-text searches inherently, requiring additional considerations or external services. Aggregation functions are basic, and there are constraints on indexing. Before constructing our database schema as seen above we tried to implement the same logic using queries instead. We ended up with our solution instead as a result of the query limitations that Firestore imposes.

How does your implementation of transactional behaviour for Level 2 compare with the all-or-nothing semantics required in Level 1?

Transactional behaviour for our Level 2 cloud-distributed system is similar to the all-or-nothing semantics in Level 1. However, in our cloud environment, network latency and potential partial failures introduce additional complexities that are handled by the Google Cloud platform. The transactions on our cloud project use two-phase commit mechanisms and a distributed database that ensure atomicity, consistency, isolation, and durability across our nodes. Despite these complexities, the fundamental goal of achieving all-or-nothing semantics remains consistent between both environments.

How have you implemented your feedback channel? What triggers the feedback channel to be used, how does the client use the feedback channel and what information is sent?

We implemented our feedback channel using the SendGrid API in order to send emails to our users providing updates on the status of their bookings. The feedback channel is triggered in our system whenever a booking attempt is either confirmed or discarded due to repeated errors (e.g. when a ticket has become unavailable). Before the pubsub worker returns an OK status code, it calls a function which sends an email to the user corresponding to the type of message that needs to be given (success/failure). A feedback loop such as this is important in our application as booking confirmations are handled asynchronously by a background worker i.e. the application does not pause until it can inform the user whether their booking was confirmed or not. As such, a separate, asynchronous feedback loop needs to be implemented, which is what we did with SendGrids API.

It is worth noting that this feedback loop is not functional in our submission. We encountered the same issue as other students, where SendGrid would not approve our requests to activate an account. Because of this, we had to implement the feedback loop with a made-up API_KEY, as advised by the course administrators.

Did you make any changes to the Google App Engine configuration to improve scalability? If so, which changes did you make and why? If not, why was it not necessary and what does the default configuration achieve?

We did not find it necessary to make changes to the Google App Engine configuration in order to improve scalability in our application. The configuration that our application was set to by default was F2, which offers a good balance of cost-efficiency to performance.
When multiple instances of our application on various accounts were opened (as many as was feasibly possible for our testing) we did not notice any degradation in the speed/responsiveness of our application despite this increased load.
The default F2 configuration is intended for applications with moderate traffic, which is suitable for our application which does not demand intensive computing or handle extremely high traffic. As such, the default configuration for Google App Engine achieves strong availability in our application given our performance requirements.

What are the benefits of running an application or a service (1) as a web service instead of natively and (2) in the cloud instead of on your own server?

Firstly, there are several benefits to running an application as a web service instead of natively. The most important, in my opinion, is accessibility. Potential users of your application do not need to download the application in its entirety in order to use it, all they need to do is navigate to your domain. Furthermore, web applications are generally less platform-specific than native applications i.e. they will run on many different platforms without having to develop different versions of the application. This further adds to the accessibility of the web service.
Another benefit of running an application as a web service is that it is far easier to update the application than it would be if it were running natively. The burden is not on the users to update the application - the next time they access the web service, it will be up-to-date.

Secondly, there are also several benefits to running an application in the cloud instead of on your own server. Running an application in the cloud reduces the burden on the developer to manage the infrastructure necessary for maintaining the application. As the application scales and/or the needs of the application change, a cloud environment provides a level of flexibility, making these changes less challenging. Difficult challenges such as predicting future data storage needs and maintaining security are made easier or even totally relieved by modern, large cloud providers like Google. And, of course, a cloud environment provides a level of constant availability which is unmatched by a personal server.

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 Google Cloud Platform can invoke various challenges. Some issues we had to consider were data transfer costs, latency issues, and ensuring we used the cloud-native version of every service we deployed locally. Some other issues that can come up when migrating from a local project to the cloud are scalability, security, and compliance considerations. This involved organising our data model to maximise query efficiency, as well as implementing a more robust security/authentication system whereby we check the signature on identity tokens to ensure their authenticity.
The Google Cloud Platform can be restrictive to the extent that we become dependant on their services. Each of the services we use are specific to the Google Cloud Platform, so migration to another provider is more difficult than it would otherwise be. In addition to this, Google Cloud's billing can be overly complex, making it difficult to estimate the costs of migrating a locally deployed application to the cloud.

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's tie-in with the Google Cloud Platform is quite extensive, and makes use of a broad range of Google Cloud services such as Pub/Sub, Firestore, Google App Engine, and Firebase Authentication. Our application is built on the assumption that these services will integrate

seamlessly with one another, so migrating to another cloud provider would pose some challenges.

In the event of a migration, aspects of our application which rely on the aforemention 4 services would need to be refactored.
Switching from Google Pub/Sub to a similar service from another provider might require a changes in the message patterns of our booking system.
Switching from Firestore to another database provider would also likely require a change in our data model. Firestore handles scalability very well, so it is likely that this would need to be considered upon transition to a new provider.
The security of our system would have to be reassessed upon transition to a similar authentication service to Firebase Authentication, so as to avoid exposing vulnerabilities in our system
And, finally, migrating from Google App Engine to a similar platform-as-a-service would involve reconsidering the computational and scaling requirements of our application, as well as adapting the deployment process, amongst other considerations.

What is the estimated monthly cost if you expect 1000 customers every day? How would this cost scale when your customer base would grow to one million users a day? Can you think of some ways to reduce this cost?

The Google Cloud Platform charges based on variety of relevant metrics from whatever of their services you use, which can be viewed on the Google App Engine dashboard. As such, the following costs need to primarily be taken into account to give a proper estimate of cost:

- The App Engine itself charges based on instance hours
- Firestore charges based on total storage use, data transfer costs (especially if your application communicates frequently with external services or serves content globally), and read operations.

The charges for Firestore read operations dominate our billing sheet and make up the majority of our costs. When taking the cost incurred by our own personal use as an indicator, we calculated that we cost the application \$0.043 per hour on average (per user).

Assuming each of the 1000 users per day use the application for 20 minutes, the total cost estimate is calculated as follows: $\$0.043 \div 3 \times 1,000 \times 30 = \$430/\text{month}$

Given that our costs are dominated by a factor which scales linearly with usage (firestore), we can assume that the costs as a whole will scale linearly as well. As such, the following is the calculation for 1 million customers: $\$0.043 \div 3 \times 1,000,000 \times 30 = \$430,000/\text{month}$

Some generic ways to reduce cloud costs include optimising firestore queries, reducing the number of emails sent with SendGrid, and considering downgrading to the F1 configuration on Google App Engine. For our billing sheet, which is dominated by Firestore, our focus should really be on optimising firestore queries. It may not be necessary, for example, to store and query ticket objects in the database, and instead we could rely on seat objects alone to provide information on which seats are available. Some of these cost-saving measures may involve changing the definitions of some of our core classes, however, and so this is beyond the scope of this project.

Include the App Engine URL where your application is deployed. Make sure that you keep your application deployed until after you have received your grades.

Our App Engine URL is the following: `https://ds-part-2.ew.r.appspot.com/`