Distributed Systems (H0N08A)

Report: Project

Andi Škrgat *(r0876363)*
Daniel Rey Šparemblek *(r0883565)*

February 7, 2022

App Engine URL: https://ds-theatres.ey.r.appspot.com/

1. Imagine you were to deploy your application to a real cloud environment, so not a lab deployment where everything runs on the same machine. Which hosts/systems would then execute which processes, i.e., how are the remote objects distributed over hosts? Clearly outline which parts belong to the front and back end, and annotate with relevant services. Create a component/deployment diagram to illustrate this: highlight where the client(s) and server(s) are.

User(front-end) communicates with the back-end(our server) using REST communication. When reserving ticket there is also Pub/Sub which now acts as emulator but later will be real so our back-end will communicate with it. Firebase authentication is hosted on a separate Google server. Shows, corresponding seats and tickets are retrieved from public API((un)reliabletheatrecompany.com) and then stored as POJO at out back-end. Bookings with random UUID are created at back-end.

2. Where in your application were you able to leverage middleware to hide complexity and speed up development?

Spring Security is a powerful framework that we use for authentication and authorization. It allows customizing role-based access control; practically all complex behavior can be defined within two classes. Building rest controllers is also trivial with Spring Boot so add-ons like adding cookies support, specifying which HTTP method will be used, and redirecting users are easily done. We would also like to point out Web Client and its easy configuration for hypermedia API.

3. At which step of the booking workflow (create quote, collect cart, create booking) would the indirect communication between objects or components kick in? Describe the steps that cause the indirect communication, and what happens afterwards. We think that the most beneficial would be to introduce indirect communication when creating bookings because this is the moment when quotes have to be saved to persistent storage which is a slow operation. Creating quotes and collecting carts are both computational tasks that can be executed quite fast. Indirect communication is done in the following steps:
1. When the application is started, a new topic is created for all messages related to confirming a cart.
2. Push subscription is created - Pub/Sub broker will send all messages to link /pubsub
3. Sometime during the application's lifetime, the client confirms the cart by executing a POST request on route /confirmCart.
4. The message is written to persistent storage.
5. The Pub/Sub broker sends a message to its subscribers
6. Once the message is acknowledged, it is deleted from persistent storage.

4. Which kind of data is passed between the application and the background worker when creating a booking? Does it make sense to persist data and only pass references to that data? A list of quotes is passed, together with information about the customer. It would make sense to persist the data, but Pub/Sub already persists data because it guarantees at-least-once semantics.

5. How does your solution to indirect communication improve scalability and responsiveness (especially in a distributed/cloud context)? In a real-world scenario, there would be thousands of time-consuming requests from different parts of the world. Our system has to be capable of handling a high volume and spread of users so the load can be distributed as equally as possible. If we would use direct communication, this can be quite a big overhead, and the Pub/Sub broker solves that by using a broker which is capable of choosing channels without unnecessarily overloading nodes because more workers can be deployed dynamically. This way, users are always provided with the last updated information. Front-end also benefits from it because UI remains responsive.

6. Can you give an example of a user action where indirect communication would not be beneficial in terms of the trade-off between scalability and usability? One of the greatest advantages of a pub/sub system is the ability to handle a high volume of data by directly on-demand scaling. However, sometimes it is simply unnecessary to use for some smaller operations so brokers could focus on scheduling more important tasks and subscribers on solving. For example, a user accessing his account is a trivial, simple-scale operation that will not influence the overloading of some server node. Using indirect communication for such operations could lead to system instability and network saturation. Another reason why e.g user accessing his account should be handled by direct communication is because indirect communication cannot provide user with direct response.

7. Is there a scenario in which your implementation of all-or-nothing semantics may lead to double bookings of one seat? There is no scenario in which double bookings of one seat can occur. First reason why it cannot happen is because as soon as POST REST request for reserving seat succeeds another request will not be sent again. Pub/Sub can create problem because it guarantees at-least-once semantics but we took care of it in the way that ids of all received messages from Pub/Sub in corresponding controller are saved in the Set data structure and if message with same id is recognized message is discarded and not send to model class for further processing.

8. How does role-based access control simplify the requirement that only authorized users can access manager methods? It simplifies it in the way that it is enough to specify a claimed role, which can be done in the Firebase emulator, and read it in doFilterInternal. By using role access it is enough to specify role permissions; and not for each user separately, which greatly reduces the time needed for adding a user or e.g. changing the logic of giving access. It also significantly reduces time needed for specifying fine-grained access for many users separately.

9. 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? Currently, we use the Firebase authentication provider. Some of the alternatives are Backendless, Amazon Cognito, Okta, Azure AD... If we would switch to another authentication provider we would have to change the login component of our system, which is currently called in the Firebase backend, and the code where we set the authentication context that Firebase Authentication uses. An authentication provider would ideally be able way to easily configure claims, which is a core idea when using access control rules. It should also be compatible and easy to set up with Spring-Boot middleware, specifically Spring Security module.

10. How does your application cope with failures of the Unreliable Theatre company? How severely faulty may that company become before there is a significant impact on the functionality of your application? It copes with well with any kinds of failures. It tries a REST request five times and if it doesn't succeed, it'll just return an empty list for that specific request, ie. time, show, seat, etc. so we can ensure that application keeps running even if API responds in the harmful way. Because of this, no severe faults can significantly impact the functionality.

11. How does using a NoSQL database affect scalability and availability?
NoSQL databases are both more scalable and more available than SQL databases. They are horizontally scalable, unlike SQL databases that are vertically scalable, meaning they can handle an increase in traffic just by adding additional servers. SQL databases, on the other hand, are less scalable, they handle an increase in traffic by adding additional components such as RAM, SSDs, and CPUs.

NoSQL databases are also highly available, meaning there is a guarantee that every request receives a response about whether it was successful or failed. Projects that require high scalability and availability should consider using a NoSQL database.

12. How have you structured your data model (i.e. the entities and their relationships), and why in this way?
The data model is structured in a way to ensure a low amount of requests need to be made to the database for the application to properly function. The data model within the back-end can be divided into two parts. The already created entities or data classes and the newly created *ShowDTO* and *SeatDTO* classes. *DTO* being "Data to Object" which maps the internal theatre data within data.json to its corresponding DTO classes. These classes are then used to put the data into the database in such a manner: *shows/showID/data*. The data can further be divided into the necessary fields and a nested *seats* collection, which is comprised of n documents, corresponding to n seats for that show.

The bookings are stored differently. The *users/userID/bookings* collection stored all the bookings. n documents in this collection correspond to n bookings that this user has made. Each booking consists of the necessary fields and an array of tickets in that booking/quote.

This structure enables us to quickly fetch a booking from the database and/or any of its field values, such that we just need to put a request to the database once, avoiding any calls within a loop.

13. Compared to a relational database, what sort of query limitations have you faced when using the Cloud Firestore?
There are several query limitations that we faced while using Cloud Firestore compared to a traditional relational database. The first is that there are no aggregation queries. For e.g. when we wanted to count which user has the most bookings, we had to do that "manually" within Java using a counter, whereas a relational database using SQL could do these kinds of aggregations using a simple COUNT aggregate query. Aggregation queries are expensive and this is why they aren't in-built into Firestore.

Another limit we faced is the read/write speeds and size constraints. Although we didn't necessarily face these limitations, it is easy to see how one of these limitations could be reached under load and while scaling. The maximum sustained write rate to a document is 1 per second and the document size limit is 1 MB. Although the size shouldn't be an issue if you organize your data correctly, we do have to take into account that things start to break down when using map and array type fields.

Lastly, queries aren't as flexible as they would be in a relational database. Mentioned above is the point of aggregation, but this also applies to JOIN and other similar SQL functions that will help you query your data in many ways.

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

It has its similarities but is quite different. The way we implemented all-or-nothing semantics in Level 1 was that we ensured that the number of successfully returned tickets (available seats) is the same as the quote size, this would mean that even if one seat is unavailable, the booking will not succeed, disabling two users from booking the same seat at the same time. The transactional behavior is implemented in a way to ensure that the bookings will only be stored in the database once we check that all seats are available. The transaction encompasses one read and two writes, the availability checking, creating the booking, and setting the seats to unavailable. This ensures that user A is not able to book the seats while user B is booking them; and when user B finishes the transaction, they set the seats as unavailable, thus making user A's transaction fail as the seats are no longer available. Both level 1 and level 2 implementations check whether the seats are available, but level 2 uses a built-in system within databases (transactions) to ensure correct blocking.

15. 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?

The feedback channel is implemented using SendGrid. Every time the user makes a reservation, indirect communication (pub/sub) is used to process this request, and SendGrid is triggered to send a new mail clarifying whether a request was successfully executed or not. If the execution passed successfully, the customer receives an email in which it says which seats are reserved, for which show, for which customer, and for which hosting company. This is formatted nicely using an HTML table. If execution didn't go well then it receives an email that all tickets have been canceled and that they should repeat the operation or try to contact our company.

16. 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?

For this task, we decided to try only automatic scaling options. First we set *min_pending_latency* to 1000ms and *max_pending_latency* to 3000ms. This means that as long as the latency is below 1000ms, App Engine will not create a new instance. When the latency is between 1000ms and 3000ms, App Engine will try to reuse some of the existing instance, and only if no instance can respond to the incoming requests, it will create a new instance. When latency gets higher than 3000ms, App Engine will try to create a new instance. Default *max_pending_latency* was set to automatic which meant that requests were in the queue for at most 10s. After this, we set *target_cpu_utilization* to 0.6 and with this, we wanted to get a better balance between performance and cost. With this setting, new instances will start to handle traffic when CPU usage reaches 60%. Lastly, we set *min_idle_instances* to 1 so there is always at least one instance ready to respond to the user's request. This will, however, increase the cost of the application because we are charged per number of instances running, not traffic they receive.

17. 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?

A native application is available only from the system from where it is installed whereas web applications can be accessed from anywhere using the Internet. Web applications are quicker and easier to build and maintain than native apps. The problem with native applications is also that they are system-specific, and not all programming languages are platform-independent. E.g. Java is but problems can occur even with Python while using certain packages. Running the application on the Cloud allows us to use the resources as needed, not like in the web server where you have them fixed. Scaling the application can be done extremely fast and without the need to restart it, whether we need more RAM, bandwidth, or memory. The great thing is also that the site owner pays only for what they are using, which is very important because when hosted on a web server, resources are often unused but still paid. Hosting on the Cloud also means high availability which is one of the key components when building distributed systems. Backups can be done in very short time intervals which minimizes the chance to lose data - one of the possible problems when applications are hosted.

18. 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?
When deploying to a cloud platform, several things need to be taken into account. The security needs to be handled appropriately and the best way to do that is to use outer frameworks and libraries that already implement that. In our case, this includes catching, decoding, and verifying JWT tokens using an RSA asymmetric cryptography algorithm. Emulators need to be replaced by real services and this often means setting complicated configurations, e.g. Pub/Sub, Firestore. A problem can also happen because deployed web applications should also be able to run locally if we would like to add new features in the future. Google Cloud App Engine has several restrictions that we also need to take into account. The maximum request size is 10MB, the maximum request header size is 16KB, the maximum response size is 32MB and the maximum deadline for a request is 60 seconds. Also one of the pitfalls is that the developer doesn't have access to the file system.

19. 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?
For changing to another cloud provider, we would need to change the storage provider (Firestore), the indirect communication platform (Pub/Sub), and the authentication (Firebase). All these components can be used externally so for e.g. it is possible to use Amazon AWS with Google Firestore but then there is the possibility to have performance loss because of the need to transfer data from one data center to another. Indirect communication should also be changed from Pub/Sub to e.g. Amazon SWS, Lambda for AWS, or Azure Web Pub/Sub for Microsoft Azure. In general, one can conclude that the application is very tied with the Google Cloud Platform.

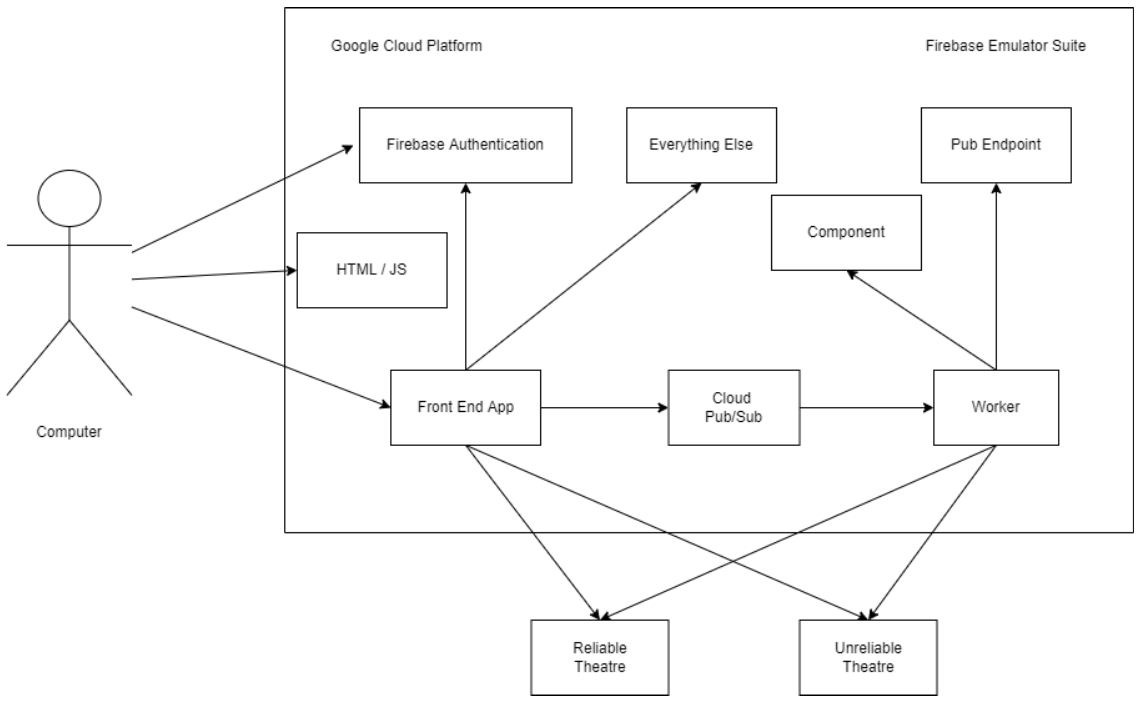
20. What is the estimated monthly cost if you expect 1000 customers every day? How would this cost scale when your customer base would grow to one million users a day? Can you think of some ways to reduce this cost?
For estimating the cost, we need to take 3 components into account: Firestore, Pub/Sub, and App Engine. For estimating the price for the Pub/Sub, we tested what the size of the request sent to Pub/Sub broker is and we obtained a value of 181 B. We will assume that every customer on average purchases 2 tickets which means that for a 1,000 customers a day approximately 1,000*2*181B=0.362 MB are sent through the network. From Google Price Calculator we conclude we won't have any costs for this number of customers. With a similar calculation for a 1,000,000 customers, approximately 362 MB will be sent a day. This gives the cost of \$0.44 a month.

For the AppEngine cost calculations are a little more complicated: When having a 1,000 customers, we think 1.5 instances is enough for handling all of the requests. We took 1.5 because there is always one idle instance in the system with the tweaking we made so an additional half instance (on average) should be enough to handle all requests. This gives a price for the App Engine ≈ \$66.92 a month For having a 1,000,000 customers a day, we think we should scale with approximately 100 instances. We think that the biggest problem that needs to be handled is sudden peaks in the incoming requests which is why there would be at least 50 idle instances in the system ready to immediately respond to the user's request. This gives a price for the App Engine ≈ \$7300 a month. One possible way of reducing cost is setting target_cpu_utilization to a very high value which will ensure that new instances are started later. Another thing we could try is setting max_pending_latency to automatic(10s) and max_concurrent_requests to high values which will again defer creating new instances. One should note that by having these kinds of settings, performance penalty will occur and some other parameters of auto_scaling like min_idle_instances should then be set to better balance performance and cost.

For Firestore, the calculation is also a little bit more complicated. For a 1,000 users interacting with the database: we have 6 shows on display with 2 of them being stored in our database, so only 1/3 of the shows are being queried in our database. Hence only 1/3 of our user base will actually book our internal shows, but we could estimate that another 1/3 would look at them before choosing another show to book. There are 6 queries to the database per booking, 5 of them being reads and one being write. Four of the five reads are only for people that look at the show, and the last one is for the people that book the shows. Meaning if 1/3 of the user-base looks at the show and 1/3 of the user-base book the show, the total number of document reads for a 1,000 users is 669,300, assuming there are 500 seats available in each show. The total number of writes need to include the writing of the actual booking, which is only 1 document per user and only 1/3 of the user-base. Meaning there are only 666 document writes to the database. That would conclude a cost of \$11.30 per month. For a 1,000,000 customers, the calculation is the same, only multiplied. Meaning there are 668,997,000 reads to the database per day and there are 666,000 writes to the database per day. Assuming the large estimate of the data taking up a 1000 GiB, the total price would be \$12,435 per month.

Total cost for a 1,000 customers per day is around \$78.
Total cost for a 1,000,000 customers per day is around \$19,735.

Remarks After the first delivery deadline, we did not implement deleting tickets because we did not see that feature in the API. However, this is now handled so there is no case in which only a partial number of tickets can be reserved.


Google Cloud Platform
Firebase Emulator Suite
Firebase Authentication
Everything Else
Pub Endpoint
Component
HTML / JS
Computer
Front End App
Cloud Pub/Sub
Worker
Reliable Theatre
Unreliable Theatre