Reservation Management System

Cloud Computing Systems Project

2020/2021

Authors:

António Beirão, nr 50816

Henrique Realinho, nr 50415

Nuno Pereira, nr 50698

Introduction

The proposed project was challenging and very interesting to us, allowing us to explore some important cloud concepts and models, while developing a system that is pertinent and useful.

The system needed to be implemented was a reservation management system that could theoretically be integrated into any company's systems, if the company wishes to provide a reservation system for its clients. This proposed project is a perfect use case for leveraging the cloud infrastructure and environment, therefore being a suitable project for the Cloud Computation Systems class, allowing us to experiment with and use many of the services and resources we learned about in the lectures and laboratories.

Design

The design of our project was thought and implemented in order to achieve a stable and reliable architecture that would support the required features for the project.

This makes the architecture of our project being designed using 4 main resources: Entities, Forums, Calendars and Reservations. Each of these resources has its own container in Cosmos DB. We also have a blob storage container for the Media images.

Each Entity can hold multiple media files, but only one forum and one calendar. Each Calendar has the information on the respective Reservations, only allowing one Reservation per day. Calendars also have a list of available days which is computed at the creation of the object, and updated every 24 hours with an azure function. Each reservation also holds information about the entity for which it was booked. A Forum keeps the message objects contrary to the rest of the resources that only hold ids. This was done because the message resource does not have its own container.

Regarding the page endpoint we were only able to design and implement three endpoints. These endpoints are used to calculate several important things, namely, list the most recent entities in the system (which is done by retrieving the entities present in cache); Check if an Entity is listed; Update the listed variable in Entity object.

It is also worth mentioning the use of Azure functions in our project. These were done so that we can replicate data as well as update information in our system on a time basis. How we used Azure's serverless capabilities is described in greater detail in the Implementation topic.

Implementation

We used Java with JAX-RS for the REST endpoints. We chose to go with the JAX-RS libraries to build our RESTful web services instead of going with another framework such as Spring as we were all already familiar with it, having used it in other projects before and also because it was officially supported by the professor, which made it easier to debug and ask questions regarding our implementation.

We followed a MVC model, separating each resource into its own Interface, Controller and Service. A resource's interface is the resource's API which defines the resource's endpoints, the controller is the class that implements the interface and sends data to and from the services, with the service being the main logic of the resource. It is the service that accesses the databases and the cache, retrieving and processing the data to then return it to the controller, which in turn returns it to the client-side.

We used Redis caching for all resources except Media, meaning that in each GET for every resource we first try to retrieve the data from the cache, and only if the data is not there the database is accessed. In each create and update operations we put (or update) the data from the cache.

The server is deployed in the West-Europe Azure availability zone.

We implemented two azure functions. One with a Timer Trigger, which runs every 24 hours and updates the available days on each calendar, subtracting one day (the day that just passed) from the available days list that each calendar holds. The other Azure Function implemented is Blob Store function that runs whenever a new blob is inserted into the system. As we only use the blob storage for the Media resource, this Azure Function runs whenever a new media file is uploaded into the system and it takes that media file and replicates it into another availability zone (US central).

The Cosmos DB resource is in the West-Europe region, with geo-replication enabled, also saving data to the East-Europe and US Central Azure regions, through multi-master replication.

We set the consistency level to SESSION in all our Cosmos DB containers. We decided to use the SESSION consistency level as it is appropriate for our use case. The trade-off between the SESSION consistency level and a stronger one are just not worth it for this use case, though obviously a STRONG consistency level would be ideal for the reservations container at least, but given that it could, and in most cases would, lead to a big decrease in performance, we opted for the SEASON level. As for a weaker level, one could be used in the entities or forums containers possibly (CONSISTENT PREFIX), but we thought the SESSION level was still the most appropriate and reasonable one, given the performance trade-off.

The database is using Last Write Wins conflict resolution, as we think it is appropriate for our use case, with no need to write a custom merge procedure.

Each create operation on every resource generates a new id for the new object, disregarding the id which came in the request, if given one. We do this in order to facilitate operations on the services and to help mitigate any security flaws that might appear.

Evaluation

Our evaluation to the system was done via artillery, with the application deployed with a S3 pricing tier. To have a consistent evaluation we made the artillery tests create 30 objects for 60 seconds. Every test seen in table 1, was made following this criteria. For table 2, we have some tests for the delete and get operations. The values presented in the tables correspond to the median response times (in ms) of the corresponding requests for 30 objects during 60 seconds, in order to achieve a fairer and more reliable comparison.

	Media	Entity	Media and Entity	Calendar	Forum	Message	Message Reply
With Caching	332	119	218	343	254	135	181
No Caching	347	65	174	295	261	144	209
With Cache USAEast	879	1641	923	1347	1037	853	731

Table 1 - Median times of creating some objects in the platform

As we can see from the results above it is possible to conclude that creating objects without caching is faster when compared to the implementation with caching. This can be explained by the fact that there are less operations being made without the need to insert objects in cache. On the other hand the creation of a message and a message reply is faster with cache because the functions involved in its implementation will update an already existing object without the need to fetch information from the database which translates in a more efficient response time. It is worth noting that if the object is not in cache the response time might be equal or even higher.

As is expected the response times from the USA based server are significantly higher than the ones from the EU server due to its distance difference (given that the requests are made from Portugal, in western Europe).

	Delete	Delete	Get	Get	Get
	Entity	**Forum**	**All entity**	**Entity**	**Forum**
With Caching	349	270	320	275	61
No Caching	269	250	321	312	103
With Cache USEast	2023	780	1234	533	372

Table 2 - Median times of get and delete some objects in the platform

With regards to the second table the results were expected for both operations. As we can see from the table it is possible to see a trade-off between the creation and retrieving time since the creation would take a bit longer but the retrieval is faster when using cache. This translates in a more efficient system for the simple fact that the GET operation will be the most used overall. It is worth mentioning that the operation *Get All Entity* gets the same result in both architectures because the operation will always retrieve the information from the database. As for the DELETE operation, the results while using caching were worse since the operation not only deletes the information in the database but also from the cache, resulting in a slower response time.

Conclusions

The system implementation and deployment gave us the perfect opportunity to better and practice using the various cloud resources we learned about in the lectures, more specifically the Azure resources and services. The implemented system could be used by real companies interested in integrating a reservation system with focus on availability and clients all over the world.

Our work in this project came with some natural difficulties, specially trying to model the data classes as best as we could given the database used and the project description. The unit tests of our endpoints using artillery were also somewhat difficult to implement for us, given we had never worked with such a tool, but it also gave us the opportunity to learn this essential part of testing a system deployed on the internet, specially a cloud system such as ours.

We implemented all the services that we set out to implement. The project was not implemented for the full grade (20), given that we did not implement support for advanced search or computation with Cognitive Search or Spark, and also did not test for geo-replication, though we tried to implement it. We tried to begin using the advanced search feature for our Forum messages but ended up not being able to implement it given the time frame.

From our tests, we can see that the results are more or less as expected beforehand. The get operations benefit greatly from caching, while the create, update and delete operations suffer in terms of performance. Overall, it is worth it to use caching in our use case given that the benefits in performance outweigh the disadvantages in the referred operations, as described above.