**Reservation Management System**

**Cloud Computing Systems Project**

**2020/2021**



**Authors:**

António Beirão

Henrique Realinho

Nuno Pereira

**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.

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 fully use it.

**Design**

Our project was 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.

**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**

// BIG TODO

testes completos:

Add\_image

Delete\_Entity 204

Add\_reservation 200

Add\_Entity 200

Add\_Calendar 200

Testes Imcompletos:

Add\_Forum error in code

Add\_Message error 405

Add\_mesage\_reply erro because message

Check entry not done

Get\_All\_Entity 500

**Add Calendar – 500**

**Add entity – media 200, entity getting 500**

**Add forum – error (artillery > TypeError: Cannot read property 'capture' of null)**

**Add image – 200 OK, post and get**

**Add message reply – error**

**Add reservation – 200 OK**

**Check entry – error**

**Comment – TODO**

**Delete entity – 204 not working?**

**Get all entity – 500**

**Caching**

Create entity: 352 ms // this is not using artillery ….. TODO

Get entity: 221 ms

Delete entity: 233 ms

Post calendar: 136 ms

Get calendar: 147 ms

Delete calendar: 107 ms

**No caching**

**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.