Distributed Systems (H0N08A)

Report: Google App Engine

Dries Janse *(r0627054)*
Steven Ghekiere *(r0626062)*

December 4, 2019

1. Which hosts/systems execute which processes, i.e. how are the remote objects distributed over hosts, if run in a real deployment on the App Engine platform (not a lab deployment where everything runs on the same machine)? Clearly outline which parts belong to the front- and backend. Create a component/deployment diagram to illustrate this: highlight where the client and server are.

Figure 1 shows the deployment diagram of the application. This diagram shows how the different classes can be deployed in a real distributed deployment.

2. At which step of the workflow for booking a car reservation (create quote, collect quotes, confirm quotes) would the indirect communication between objects or components kick in? Describe the steps that cause the indirect communication, and what happens afterwards.

The indirect communication takes place at the 'confirm quotes' step in our application. The actual confirmation and creation of the reservations is the most demanding process. This part is the bottleneck activity of the application. By decoupling this part of the application, it allows the back-end to introduce an additional delay for the processing. The front-end will achieve better availability and performance because it does not have to wait for the actual creation of the reservations. The CarRentalModel, which contains the confirmQuotes method, will push a task to the task queue service. The task queue service will then send a http post request to the worker with the worker url and the serialized parameters. The worker will execute the task. In this case it tries to create reservations for all the quotes, making use of a transaction.

3. Which kind of data is passed between the front- and back-end? Does it make sense to persist data and only pass references to that data?

Instances of the 'PayloadWrapper' class are passed between the front- and back-end. This class has a list of quotes, the name of the renter and the email of the renter. The confirmQuotes method in the CarRentalModel class will create an instance of the PayloadWrapper class. This instance will be serialized to a byte array and given to a TaskOptions object. This TaskOptions object will be pushed to a queue and the worker will execute the task.

In our solution we made use to pass this data by value. Passing this data by value requires network traffic for sending the data to the Worker servlet. It is possible to persist data and only pass the reference, but in this context it makes less sense. This will result in storing the content of the PayloadWrapper class in the database which will have to be done by the front-end. Because this is more work for the front-end, it will downgrade the user experience. A second disadvantage is that the worker to interact more with the database. This can result in lower performance.

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

We used an email service to let the customer know if their reservations were made. When a client confirms a quote, the client-side will create a task in the queue and the user is redirected to another page which shows the message that the reservation is received by the server. When the task has been executed, the worker will send another email to the customer. This could either be a success or failure email, since there could be a conflict with the current reservations and the requested reservations at the point of execution. Either way, we supply the list of reservations in this mail to make sure the customer has all the details.

We chose this kind of backchannel instead of a status message on the website, since we feel like a customer will never leave the website open for hours/days to track their progress.

5. How does your solution to indirect communication improve scalability?

Since the bottleneck transactions are executed on a different servlet (which could run on a totally different server) and are simply called with an URL and a payload, the load on the client-side server is reduced. This way we can separate the load on servers, which improves scalability.

6. Workers in GAE's Task Queues are by default set to run in parallel. While parallelism is usually a desirable property of a cloud service, as it enables scalability and thus faster overall processing, it may also endanger the application's state consistency. Assume a scenario in which two different clients try to confirm a couple of tentative reservations, i.e. their quotes are queued to be processed by the back end. Both include a tentative reservation for the last available car of a certain car type, so that, assuming correct behaviour of the car rental application, it should fail to confirm the quotes for one of them.

(a) Is there a scenario in which the code to confirm the quotes is executed multiple times in parallel, resulting in a positive confirmation to both clients' quotes?

Yes, there is a possible scenario. The most simple scenario is that the application makes use of one queue and two workers which can execute tasks in parallel. If the confirmation of two lists of quotes are executed in parallel on the workers, an inconsitent state can occur. This happens when both workers request the status of the same car, while no actual reservation is already made. Both workers will make the reservation, while one reservation should have failed.

(b) If so, can you name and illustrate one (or more) possibilities to prevent this bogus behaviour?

(c) In case your solution to the previous question limits parallelism, would a different design of the indirect communication channels help to increase parallelism? For this question, you may assume that a client will have quotes belonging to one car rental company only.

A different design could be that we create a queue for each car rental company, with one worker on each queue. This will allow parallelism between companies so that we can process the 'confirmQuotes' method of each company at the same time. Furthermore, the single worker guarantees that there is never a inconsistent state.

7. How does using a NoSQL database affect scalability and availability?

Most standard SQL databases are vertically scalable, which means the system admins can increase the load on the server by adding or improving hardware components (RAM, SDD, CPU, ...). The fact that NoSQL have very little functionality built in, makes the operations very basic and very quick. Since the database doesn't have to do a lot of compex work and handling difficult locks, it is really easy to partition the data sets across many servers/disks. Thus, NoSQL databases can be scaled horizontally, which means multiple servers can handle the same database. Obviously a NoSQL database can be scaled vertically aswell. NoSQL is preferred when choosing for large or constantly evolving data sets.
Since we can replicate the data very easily, this means that availability will go up aswell. Since multiple copies on different servers are stored, these will decrease the overall risk of losing the data.

8. How have you structured your data model (i.e. the entities and their relationships)? Compared to a relational database, what sort of query limitations have you faced when using the Cloud Datastore?

Figure 2 shows the structure of the data model. Since the only type of relationship between two entities is a one-to-many (child/parent) relation, we created a parent root entity for each CarRentalCompany. Each company has a list of CarTypes, which on his turn has a list of Cars. And each car has a list of reservations. Quotes are not stored in the database.
Not being able to join tables made the querying harder, for example, we needed to specify each parent in each key to query the child of these parents. Above that, since we're using Google App Engine, we are dependent on the restrictions which Google specifies. For example it is impossible to create a query with more than one quantity comparison (greater/lower).

9. What is the most critical difference between confirmQuote and confirmQuotes from the viewpoint of transaction management? Explain an alternative to your solution for the all-or-nothing semantics of confirmQuotes which is essentially different from the viewpoint of transaction management. Given the underlying distributed storage layer, what are the implications of each solution for performance and data consistency?

The single 'confirmQuote' does not need an implicit transaction, because when we try to confirm the quote, it will look for available cars. If there are no available cars found, the server will never create a Reservation. Thus we never need to cancel the transaction, since no data is stored in the store. The multiple 'confirmQuotes' does need a transaction for obvious reasons. If any of the reservations fail, all of them should revert and the data is never committed.

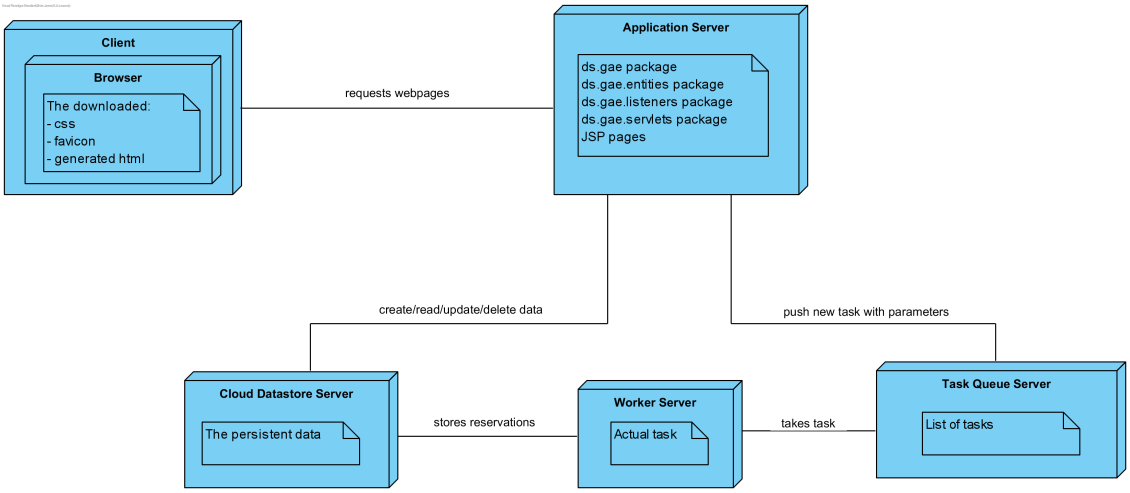

Figure 1: Deployment diagram

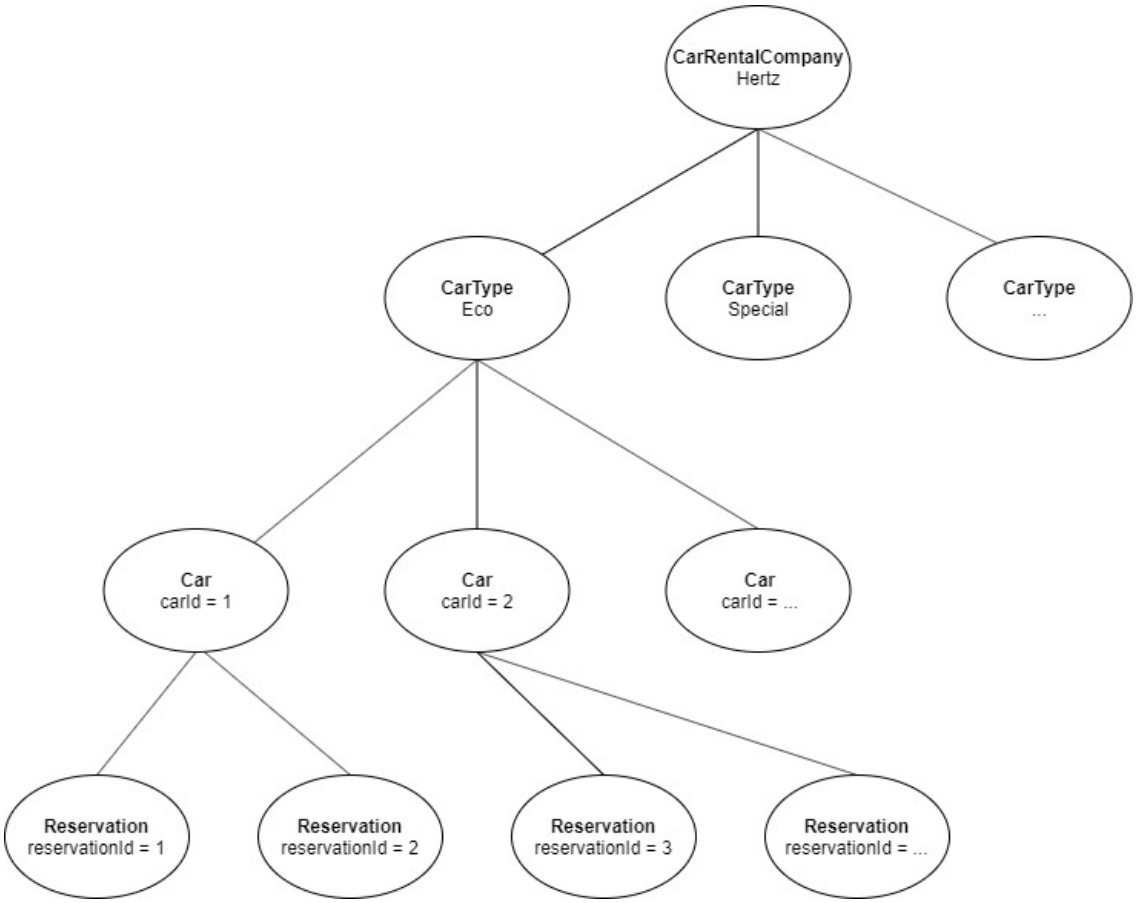

Figure 2: NoSQL Entities Structure