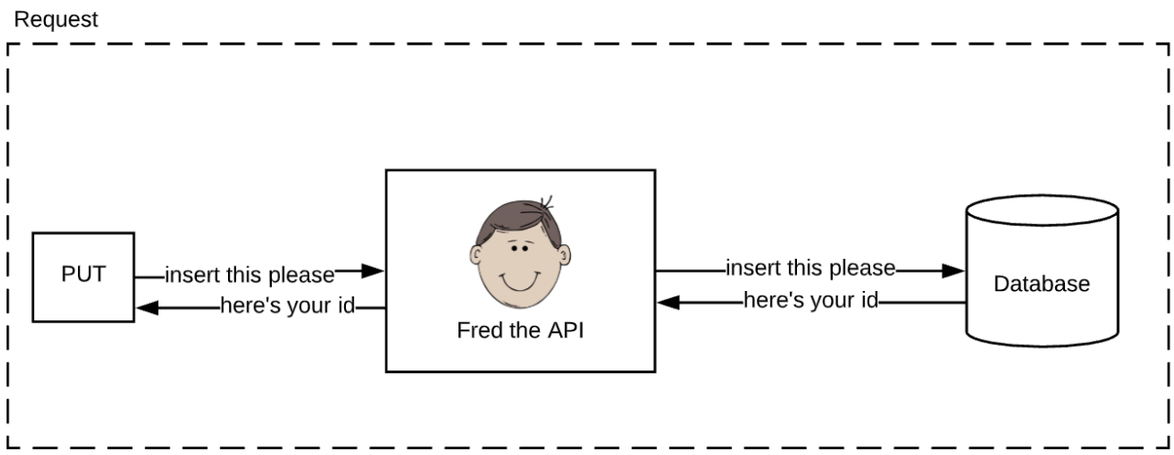
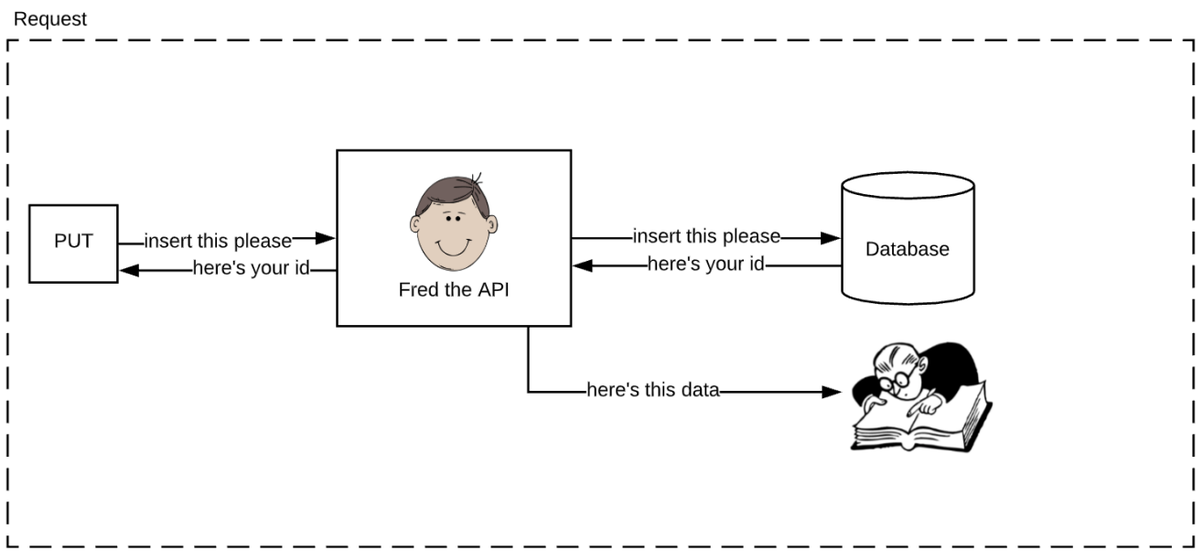
Eventual Consistency: What, How, and Why

Naguib (Nick) ihab Follow
Jan 6 · 7 min read ★

There once was an API called Fred. Fred would often receive PUT requests from a client and inserts the data into the database and replies back with the id.



Fred's life was simple until a new data auditing service started asking for any data that Fred receives to go to it as well.

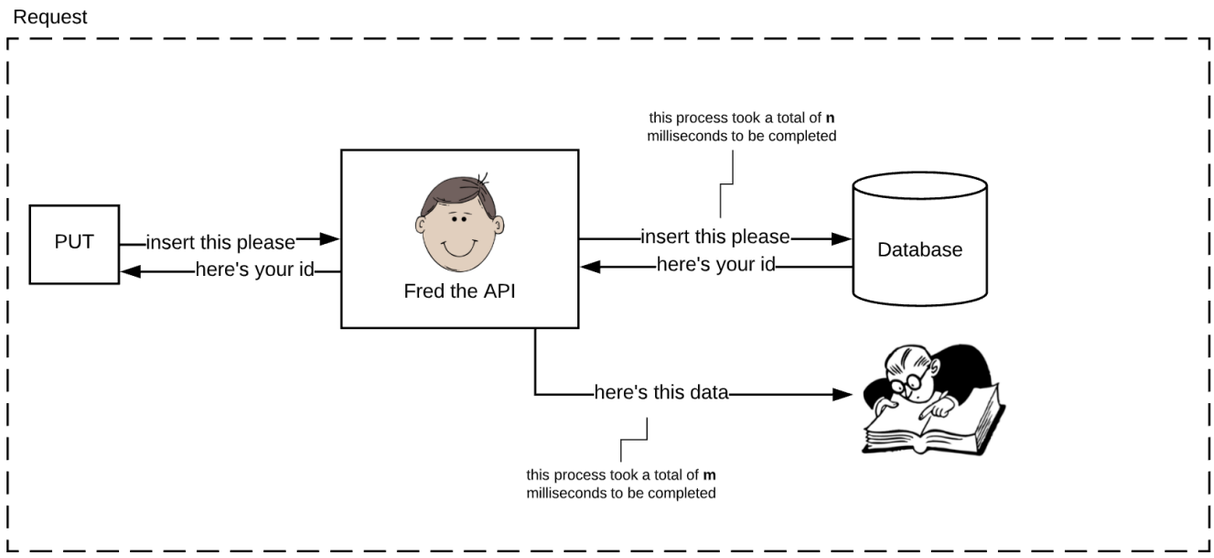


This creates two distinct problems.

Problem #1

Imagine inserting the data into the database took around **n** milliseconds, which meant that in the first scenario the client making the request would take **n** to receive the id and finish the request. Let's also assume that the auditing service requesting the data takes

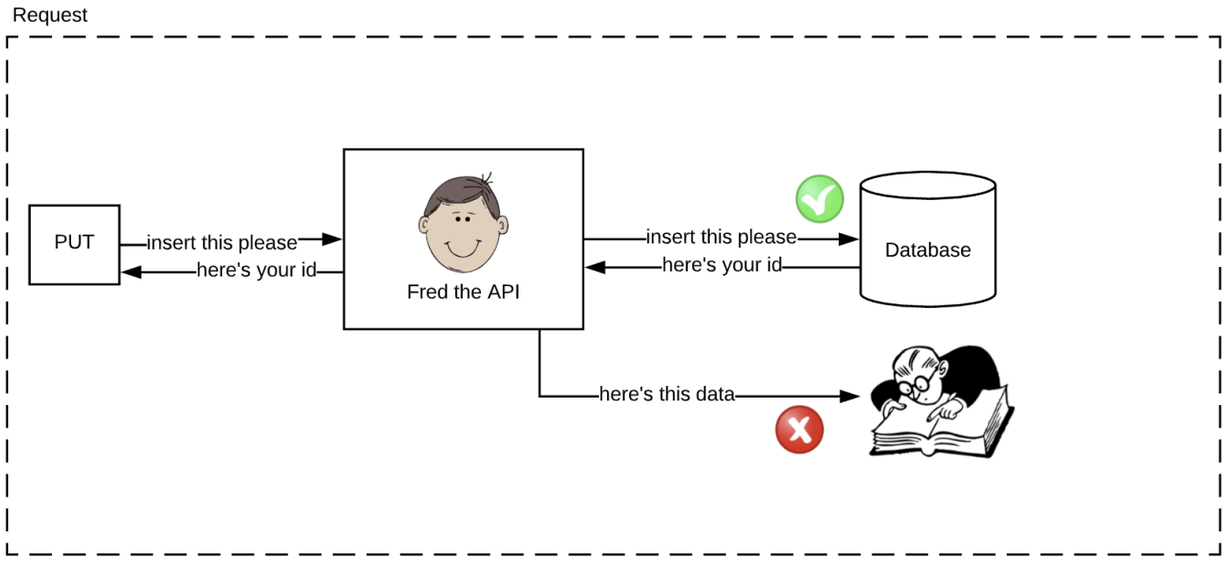
around **m** milliseconds to consume it, so after adding that service the client making the request now takes **n+m** to finish the request.



Since we're building architecture that scales, this time will keep increasing as we add more services.

Problem #2

It's a fact of life that sometimes requests fail for any number of reasons: the client is not ready, validation failed, too many requests..etc..etc.. So what would happen if one of the requests fail, i.e. if the data was inserted into the database but didn't get through to the auditing service



We can't tell the client that the request failed because it kind of didn't and if we try to re-insert the data, the database would complain (we can get around that by upserting but we'd be bending a business rule)

. . .

One solution to both problems is **eventual consistency**, but to understand that we need

to first cover **CAP**

The CAP principle

CAP stands for consistency, availability, and partition resistance. The CAP Principle states that it is not possible to build a distributed system that guarantees consistency, availability, and resistance to partitioning. Anyone or two can be achieved but not all three simultaneously.

> *Consistency means that all nodes see the same data at the same time.*
>
> *Availability is a guarantee that every request receives a response about whether it was successful or failed.*
>
> *Partition tolerance means the system continues to operate despite arbitrary message loss or failure of part of the system.*

From the book "Practice of Cloud System Administration, The: DevOps and SRE Practices for Web Services, Volume 2" By Thomas A. Limoncelli, Strata R. Chalup, Christina J. Hogan

Eventual consistency (EC), is able to achieve Availability and Partition tolerance at the expense of Consistency, what that means is that we can't guarantee that all of the recipients of the data will see it at the exact same time but we do guarantee that we'll always able to have the service available because we're tolerant to any dependencies failing.

We guarantee:
- **A**vailability
- **P**artitioning tolerance (AKA failing dependencies)
We don't guarantee:
- **C**onsistency between data stores

. . .

Eventual consistency explained with coffee

Imagine you go to a coffee place and order a latte and a scone. The cashier takes your order, you pay and then you wait for your coffee but you take your scone immediately.

As you move to the next window waiting to pick up your coffee you ponder on what just happened:

- The cashier was available to take your order and gave you a receipt of your transaction
- You don't know if the espresso machine was working at the time of ordering, but it doesn't affect your order

- You have a guarantee that you will *eventually* get your coffee, but you don't know the exact time

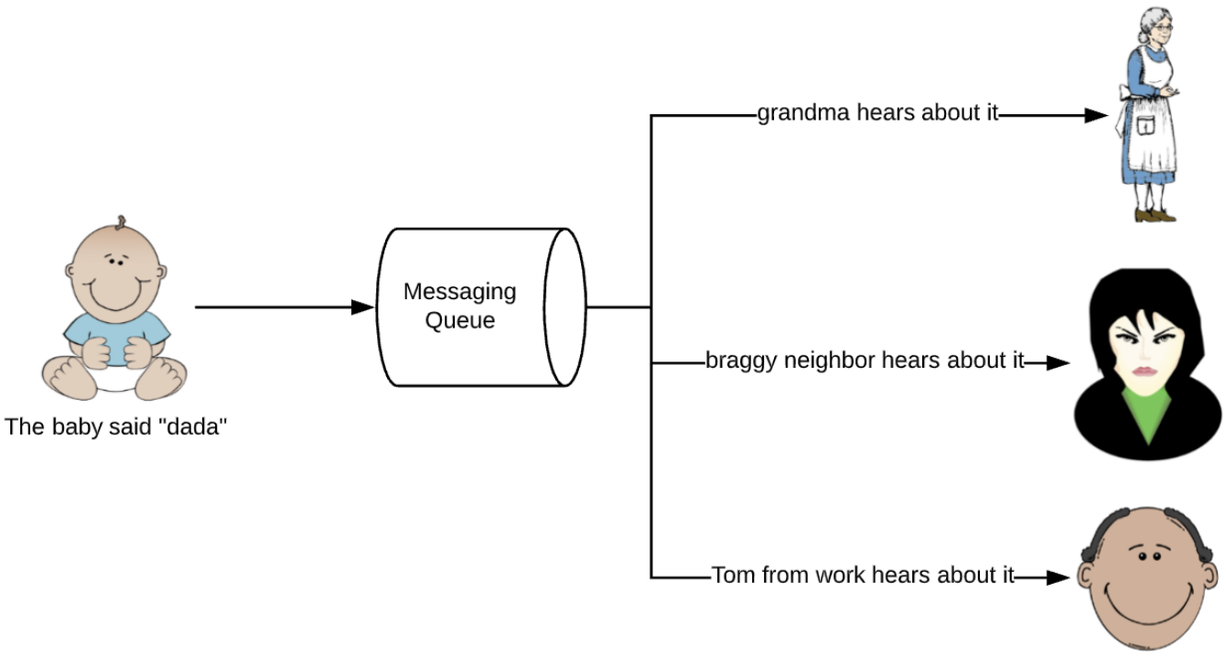
Your latte is now ready, you take it and enjoy it with the scone.

. . .

We've covered the Why, now let's cover what it is and how to implement it.

The What?

EC is delivering the data to a message queue that takes the message and delivers it to whoever is listening.



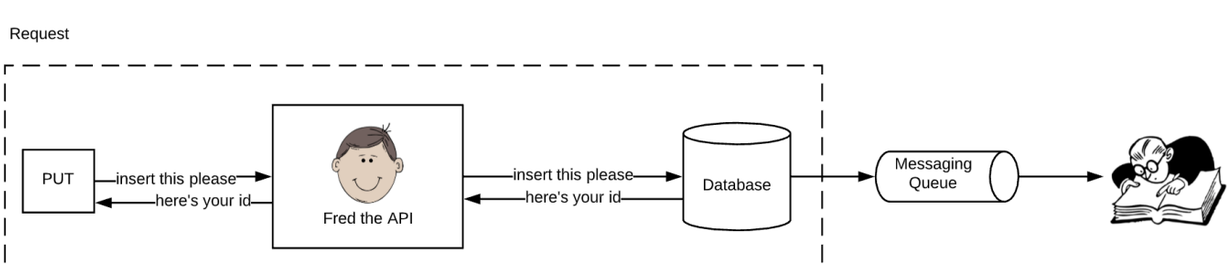
Some messaging queues push to the receivers, while others just hold the data and the receivers keep a record of which message they are at.

However, the main feature in most messaging queues is that if the receiver does not consume the message for any reason, the message queue will keep retrying to send the message or the receiver will keep retrying to consume it.

The How?

Using our previous example, Fred the API, we can implement EC in two ways:

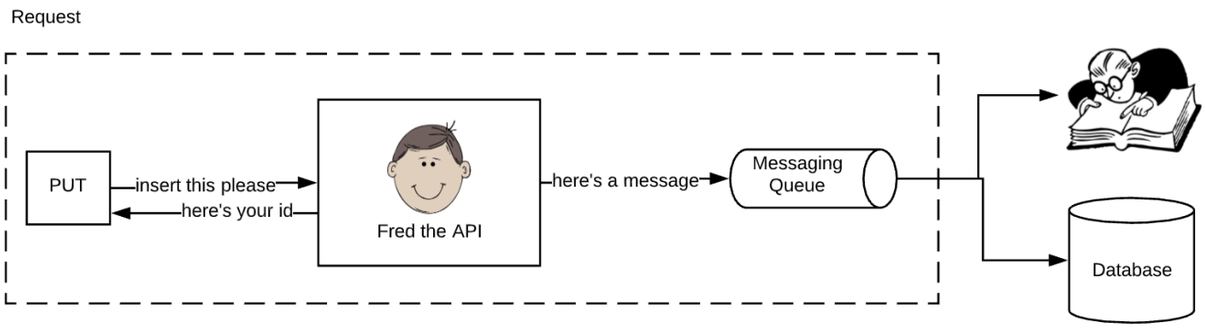
Way #1 database first



The request is limited to inserting the data in the database; after it's inserted Fred tells his client that the data is in and the request ends there but the data's journey doesn't. The database triggers the MQ and sends it the data that it has just received, the auditing service picks up the data and consumes it without affecting the reliability or the time it takes to complete the request.

In that method, we guarantee that the data is in the database by the time the request ends but we can't guarantee that it is in the messaging queue.

Way #2 messaging queue first



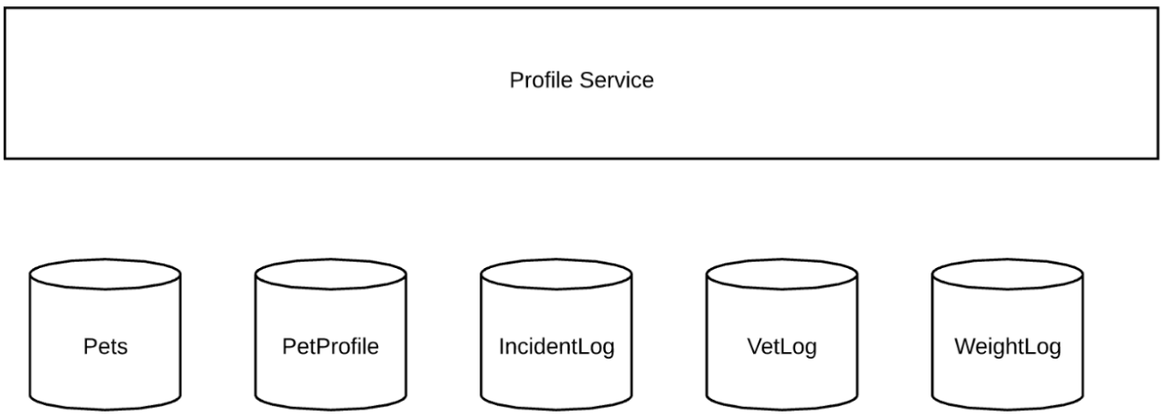
In that method, we guarantee that the data is in the messaging queue but not in the database. The API is also responsible for creating the identifier for the new data.

You'd use that method if having your data in the messaging queue is more important than having it in the database.

. . .

A real-life example of eventual consistency

In My Guinea Pigs app, we split up the pet's profile data on different DynamoDb tables.



The *PetProfile* table contains a summary of that profile but not the whole data, i.e. the name, age, gender and some of the logs.

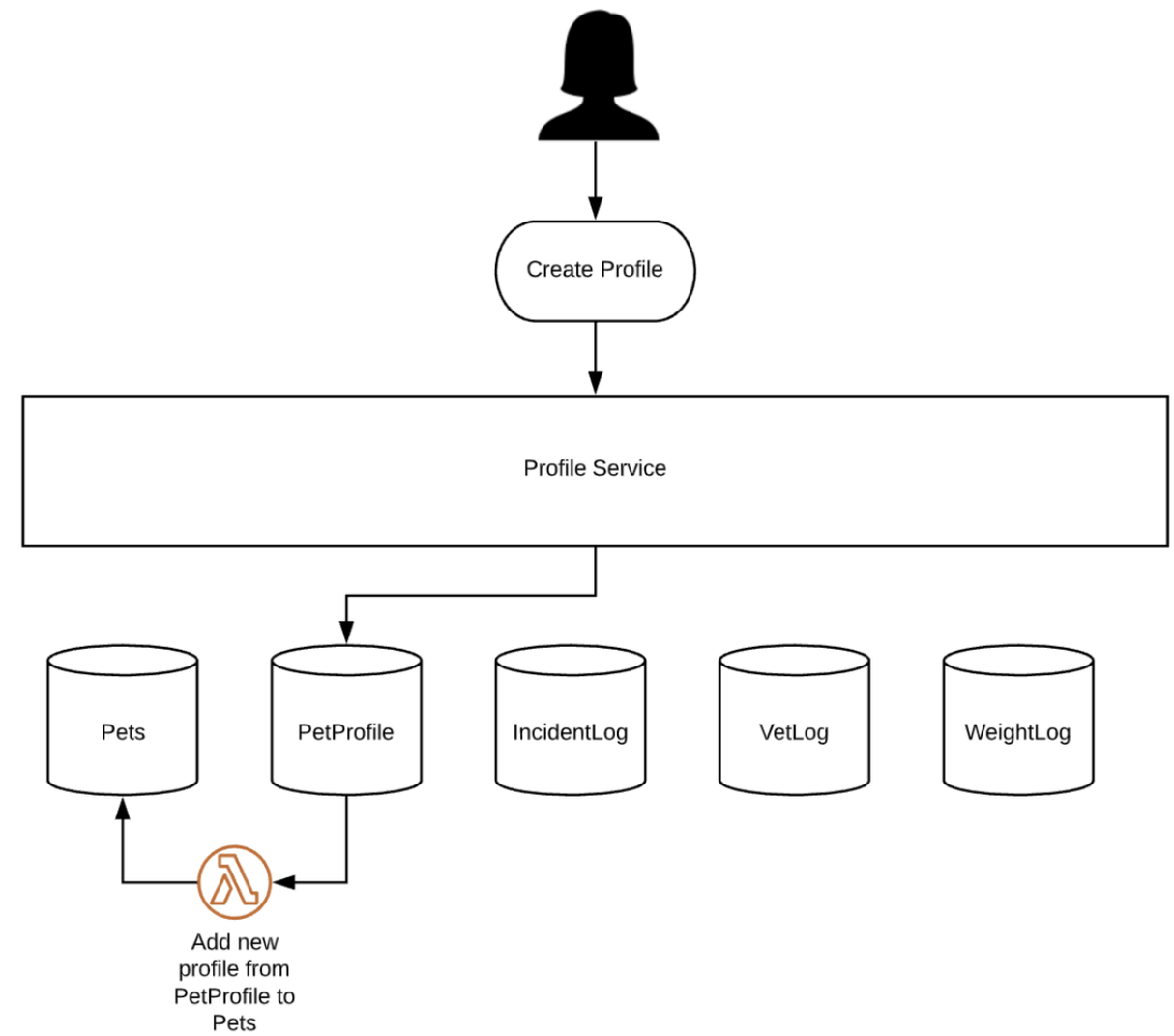
The *Pets* table contains fewer data about the profile but all of names of that user's pets.

If the client requests to see all the pets that the user has we use the *Pets* table, if the client requests to see the summary data about a particular pet then we query the *PetProfile* table, and if the client needs to update the age of the pet then we go to the table that has this data which in that case the *PetProfile*.

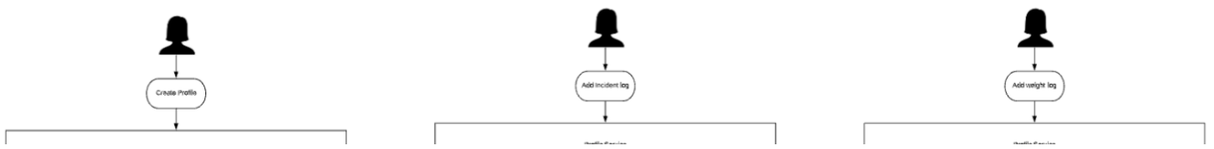
However, if the client needs to add a new profile, that data needs to be stored in both tables and even though we can batch update both tables at the same time in a single transaction we don't *have* to because the *Pets* table doesn't need to know about that information as soon as it is available but will eventually need to know about it so we went with EC.

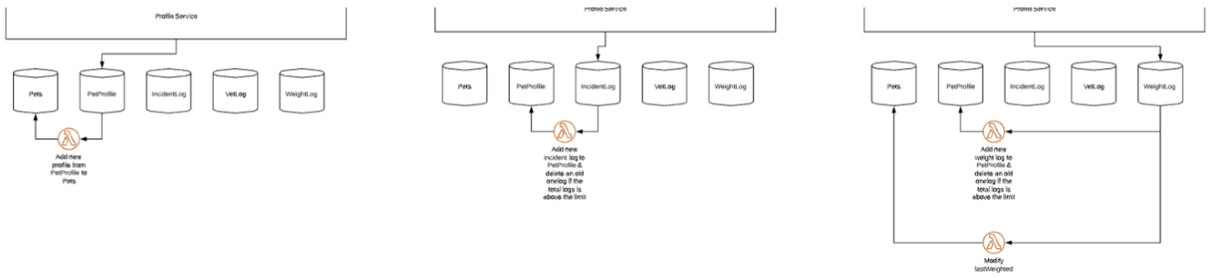
Eventual Consistency in DynamoDb

DynamoDb offers a feature called Streams, which allowed us to trigger a lambda function that would put data in another table:



And the same goes for the rest of the tables, at some we have the stream triggering multiple functions.





What is wrong with this picture?

As the CAP principle explains, we can guarantee that the service will be available and that the data will be inserted in at least one of the tables, but we can't guarantee that all the tables will consistently have the same data at any point in time. In our case, that was an acceptable scenario, and in a healthy system, the client wouldn't notice because the data transfer happens nearly instantaneously.

. . .

Further readings

I hope the article explained in a nutshell what eventual consistency is and how is it used, but I do encourage you to do further reading before deciding EC is for you:

Microservice Trade-Offs

Many development teams have found the microservices architectural style to be a superior approach to a monolithic...

martinfowler.com

Using Eventual Consistency and Spring for Kafka to Manage a Distributed Data Model: Part 1

Given a modern distributed system, composed of multiple microservices, each possessing a sub-set of the domain's...

programmaticponderings.com

Transactional Outbox Pattern - Pradeep Loganathan

A microservice often needs to publish messages or events as part of a transaction that updates the database. For...

pradeeploganathan.com

Microservices Eventual Consistency Software Architecture Programming Software Development