✨ Member-only story

Handling Eventual Consistency with Distributed Systems

Mario Bittencourt · Follow
Published in SSENSE-TECH
9 min read · May 21, 2021

Listen Share ••• More

At SSENSE, we employ microservices to provide functionality for both our customers and back-office operators.

To operate at scale and provide resilience in a distributed environment, we leverage several patterns, including:

- Using events to communicate changes (Event-Driven Architecture)
- Using read models for specific access patterns (CQRS / Event Sourcing)
- Using replication of data between persistence models (source/replica)
- Using faster medium for frequently accessed data (caching)

The use of any of the methods above welcomes the fact that the systems are eventually consistent, which presents some challenges that are often overlooked by developers.

This article aims to discuss some of these challenges, while sharing potential approaches you can take to handle the eventual consistency aspect in your applications.

Defining Eventual Consistency

The term eventual consistency is used to describe the model, commonly found in distributed systems, that guarantees that if an item does not receive new updates,

then *eventually* all accesses to that item will return the same value matching the last update, therefore providing a *consistent* result. Systems that present this behavior are also referred to as convergent.

Perhaps the most common example is when you leverage asynchronous replication between two or more instances of an RDBMS (Relational Database Management System). In this setup, the application is scaled by driving the read operations to the replica while keeping the writes to the single main instance.

Because of its asynchronicity, when the write is performed it is only persisted locally, and later another process takes care of copying and applying the same operation on the replica(s). As this process is independent of the original write operation and is not instantaneous, there is a time where the replica does not contain the updated information.

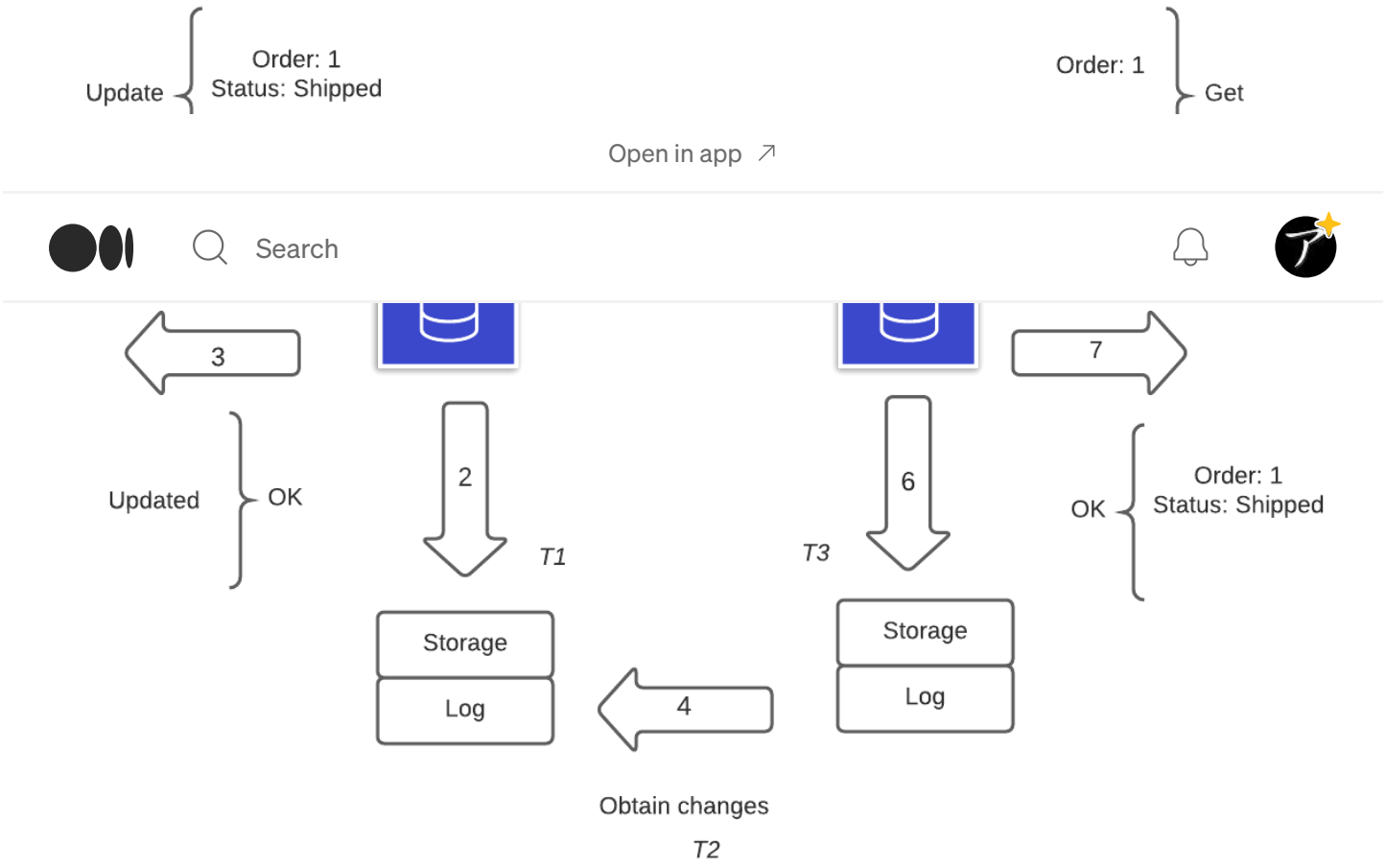


Figure 1. Simple replication operation.

Figure 1 illustrates the replication — in a simplified way — that happens between the main and its replica. In this example, if someone queries the replica in instant T3

where T2>T3, T2>T1the result it would provide would not match the one that already exists in the main instance.

Another example happens in an event-driven system, where asynchronous messages, also known as events, would be published to inform of changes just to be picked up by other parts of the system. These events would in turn trigger additional actions in the systems that consume them, such as updating local storage with copies of the data.

Similarly to the replication, there is a temporal disconnection between the part of the system that originally registered the state change and the other part(s) that are interested in those changes.

This disconnect is something intrinsic to this model and comes with a specific set of challenges. I will cover two of the most common ones next.

Common Challenges

Read After Write

This is perhaps the most common problem that we face. You execute a write operation to an entity of your system, and right after you attempt to retrieve it from the persistence, just to find out it is not there!

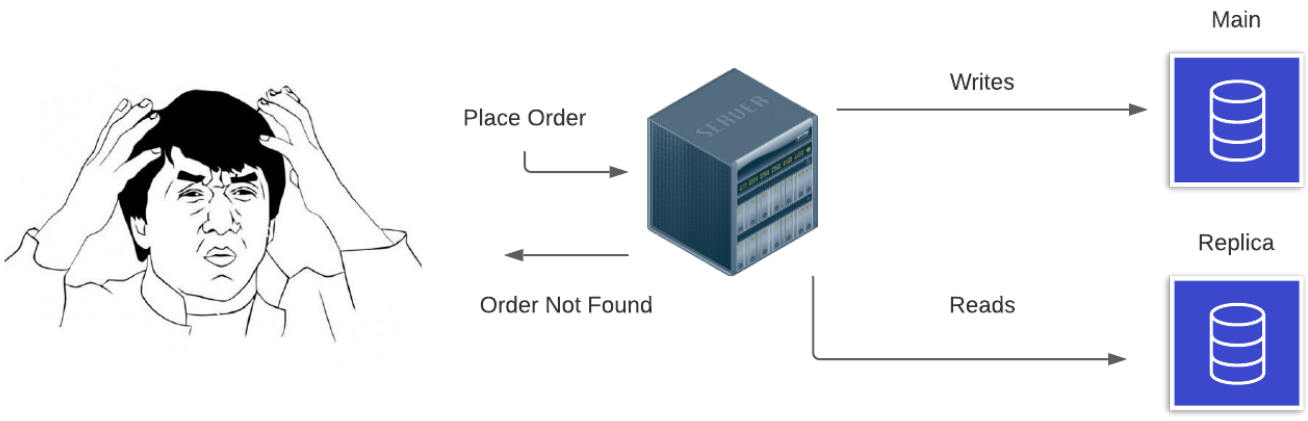


Figure 2. A read after write issue.

Figure 2 illustrates the case where your application is writing — creating or updating — an entity and in the same execution flow attempts to read it back. Because the replication takes time, depending on the current load of your application and persistence layer, by the time you execute the query, the changes have not yet been propagated.

The “funny” part is that during the development this will not manifest itself. You normally do not have a replication setup in your development environment, hence the reads and writes go to the same persistence. By the time you deploy your application and have real users interacting you will find out ‘strange’ behaviors that you can’t consistently reproduce.

Concurrency Control

Almost like a corollary of the read after write, with eventually consistent systems you often find yourself retrieving an entity from a projection, a read-only model, only to perform additional changes. All goes well if the projection is up-to-date with the source, but if that is not the case you may end up losing previously updated information.

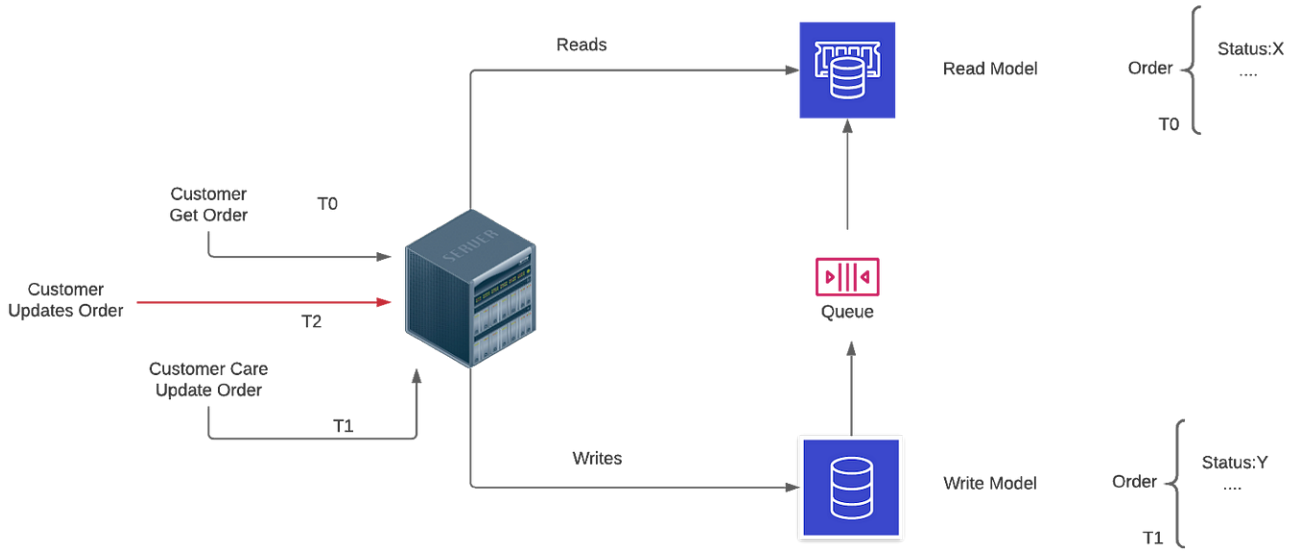


Figure 3. The change in T2 would overwrite the one in T1 (T2>T1 >T0).

Figure 3 illustrates that your order has been updated by a Customer Care Agent, but before the change has been propagated to the read-only model, you decide to edit your order. The system will end up in an inconsistent state because it does not take into account the concurrency aspect of the access.

Of course, this is not exclusive to eventually consistent systems, but it is an important aspect to not be neglected.

Solutions

The solutions I will explore next try to address the challenges in different ways. Depending on the case, you may use one or more combined to handle eventual consistency issues in a graceful way.

Fake It

Arguably the best way to avoid the read-after-write issue is not to perform a read at all. Imagine the situation illustrated in figure 4.

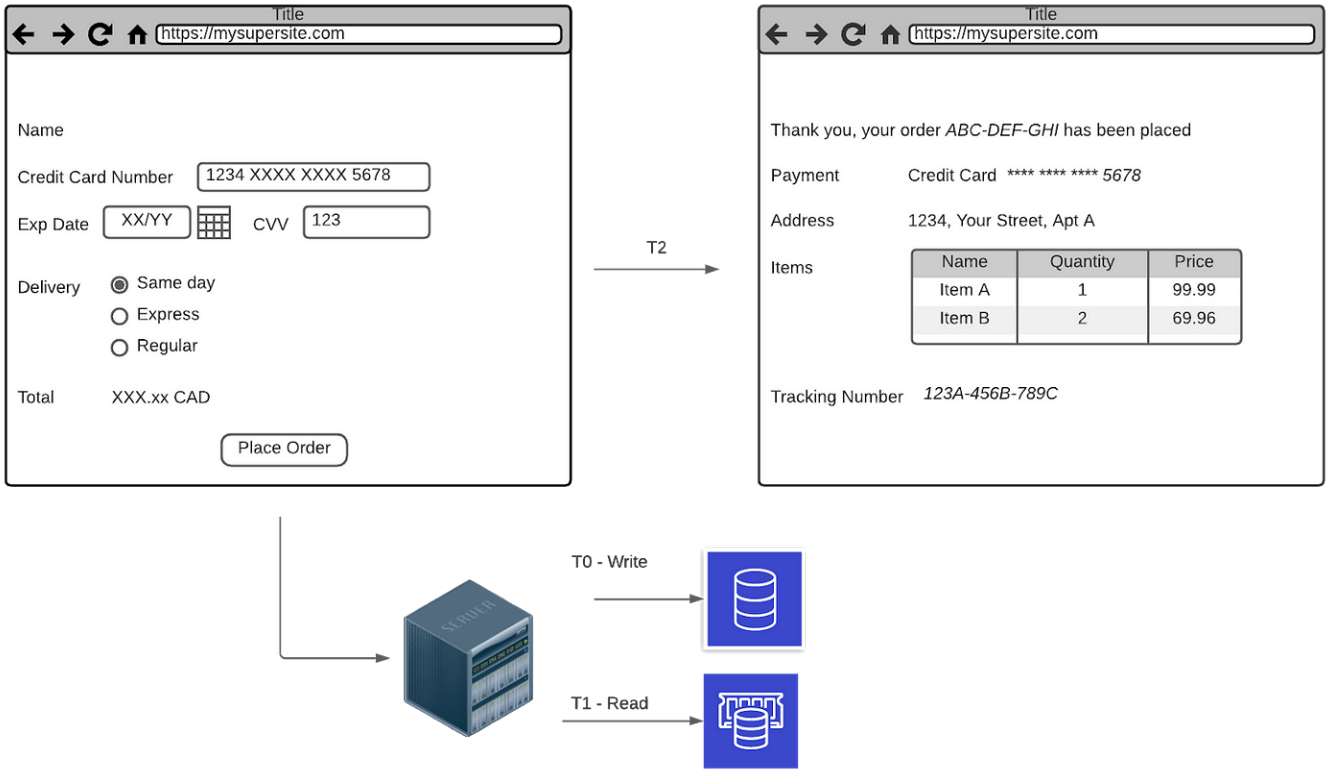


Figure 4. Place an order and display the summary of the order that has been created.

You just placed an order and want to show a confirmation page to your customer. Because we are trying to read the order right after placing it, we will not see it if there is any delay — replica lagging due to higher load, message broker with a backlog of messages, etc.

A solution would be not to read the orders, and simply show the order information you have in memory at the moment of placing it.

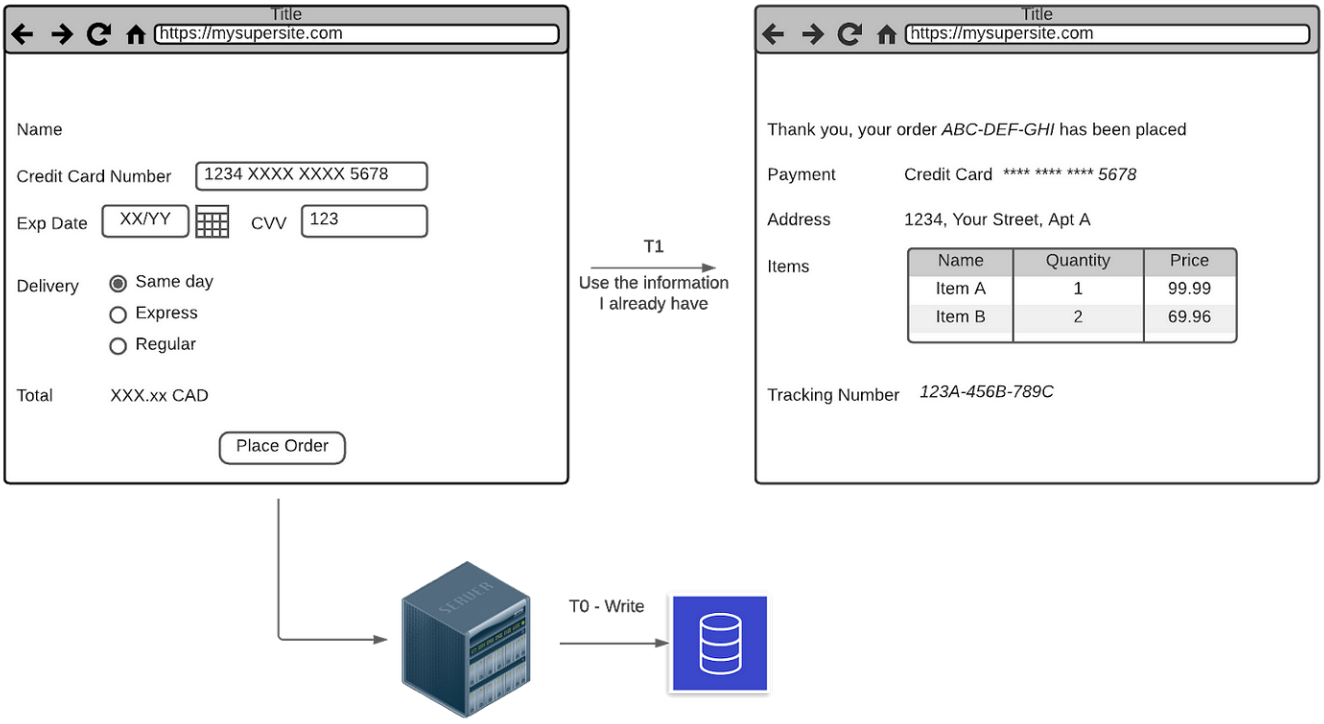


Figure 5. After placing the order just show the customer the same information you have.

If your use case generates more data while executing the operation you will have to consider showing just the information you already have and signaling that the operation is still in progress.

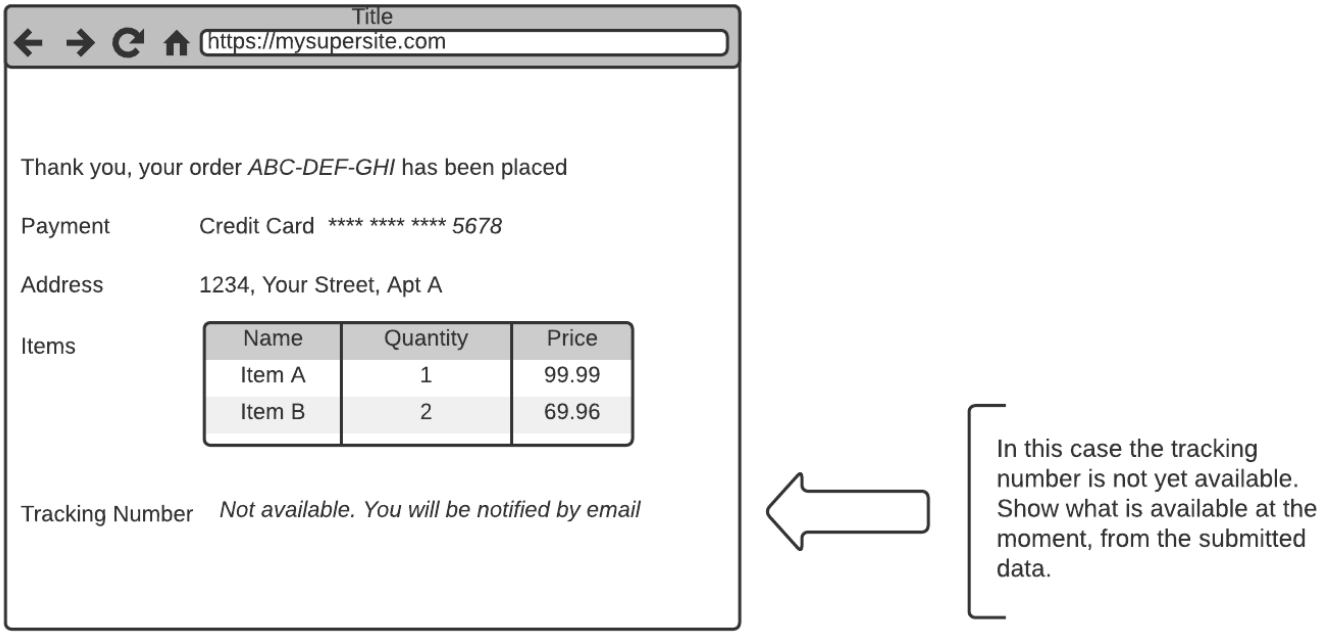


Figure 6. Use the information you have and indicate that portions are not yet available.

Set an Expected Version

If you have a use case where you are making changes to an existing entity and trying to retrieve the updated entity right after, one solution is to define what the expected version of the entity you want to receive is.

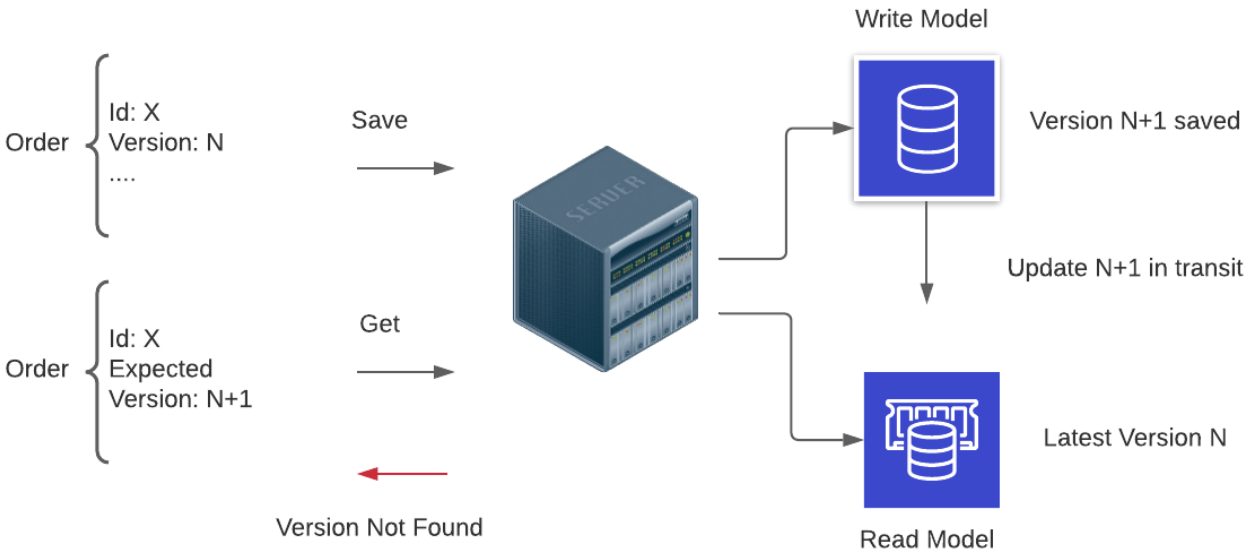


Figure 7. Using the expected version allows detection of the entity that has not yet been updated.

In this solution, you are embracing the fact that your system may not have converged by the time you requested the entity. If you do not receive the version you expect or a later one, you know you have to do something and can't use the information you just received. At this moment you can show a custom message to the user or use a spinner — see the UI Poller approach next — and retry fetching until you succeed.

This is a more sophisticated approach that requires you to use some form of versioning to the entities you are manipulating, but still simple to implement.

Having the version also helps with the concurrency control because at the server you can simply reject a change that expected a version that is older than the one that already exists.

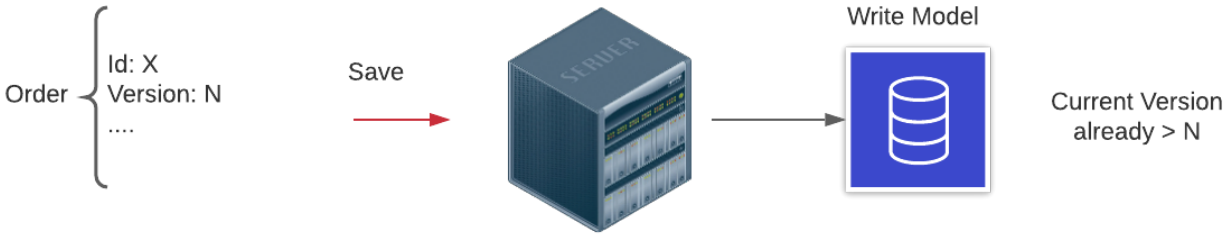


Figure 8. Using the version helps to detect changes already happened to your entity.

If your use case allows, you may want to consider even merging changes or using conflict-free replicated data (CRDT).

UI Poller

In the two previous solutions, I attempted to avoid the eventual consistency issues by not retrieving any information or detecting if the one I retrieved was good to be used.

Although those can be applied to many situations, what happens when it is not the case? For example, you want to return a previously placed order and the system generates a return code after processing the request.

You can't use the Fake it approach because this code is not part of the submitted data and the version you expect is still not available. A simple approach is to use the (in)famous spinner approach and behind the scenes keep polling for the updated information.

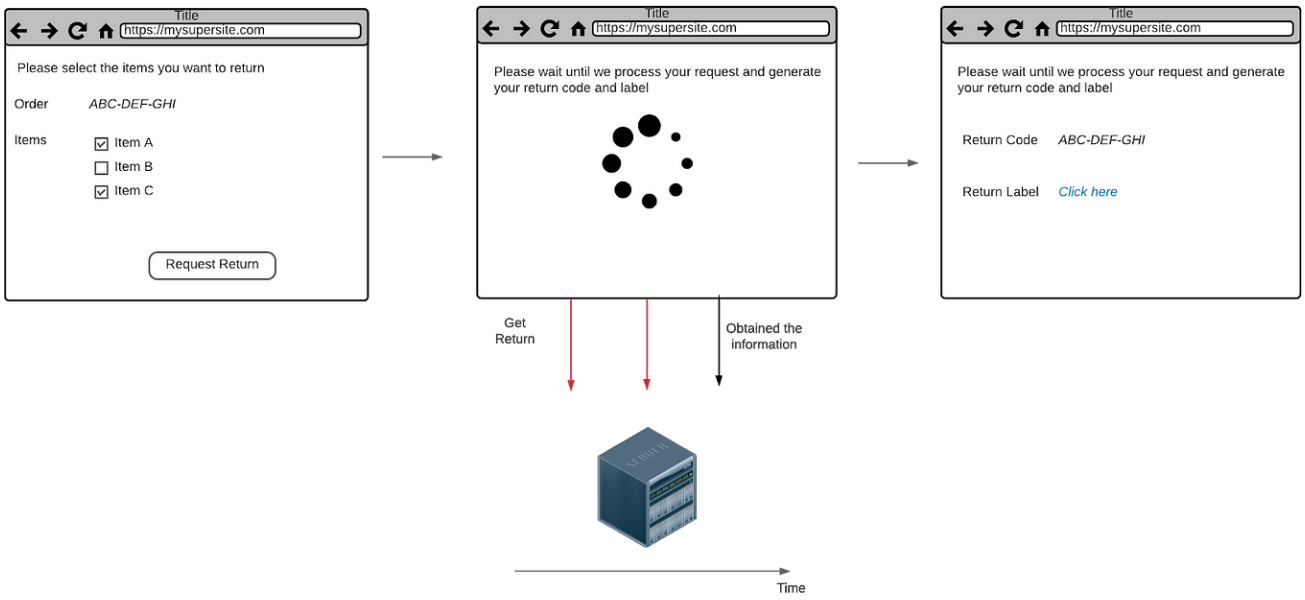


Figure 9. A retry poller attempts to locate the information you need.

As illustrated in figure 9, after receiving the information you stop the spinner and provide the information the user needs to see.

The downside of this solution is that you will increase the load on your backend with potentially useless requests. In order to alleviate the situation, you should consider at least setting at the client-side a retry mechanism that uses a back-off strategy and a maximum number of retries before giving up.

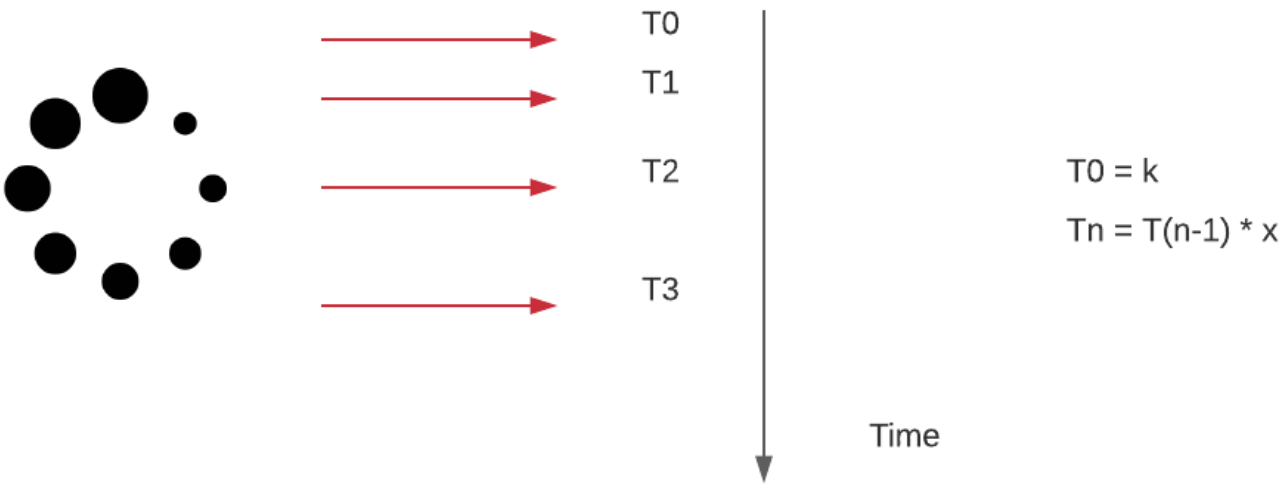


Figure 10. Increasing the interval between retries.

If you are monitoring your application — like you should — you can even set the initial retry interval based on the Nth percentile for the use case you are trying to handle. If figure 11 illustrates the execution time for the use case, you can choose, for example, the 75% and set the first attempt after 500ms, the second with 750ms, and the third and final one at 1300ms.

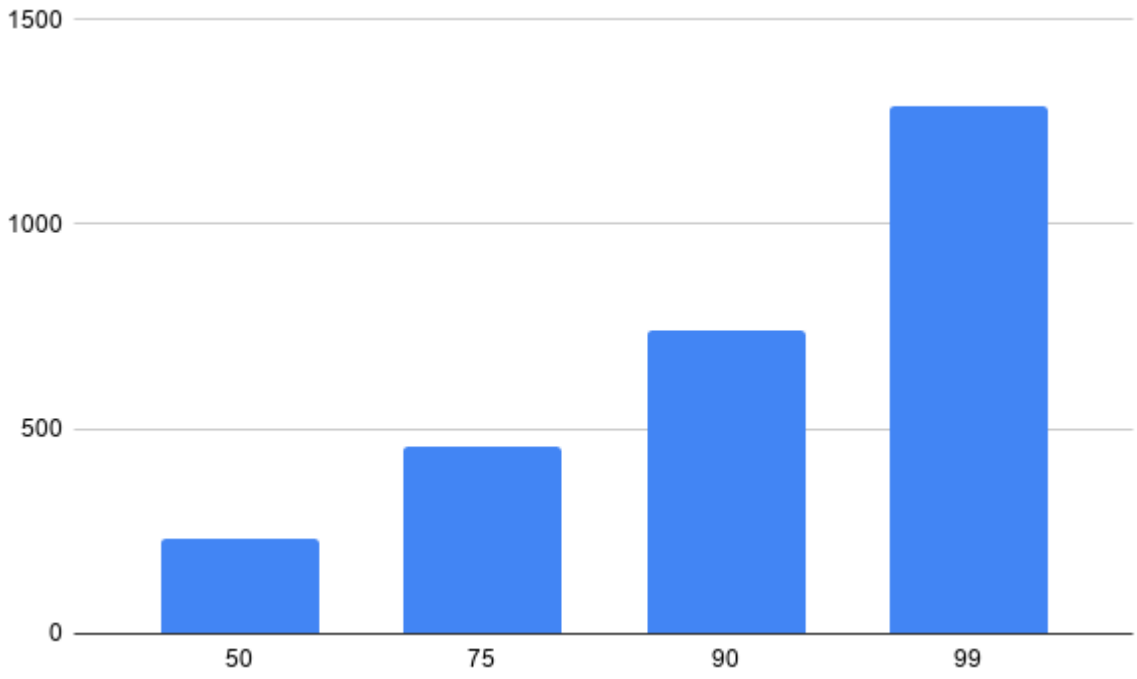


Figure 11. Using the execution time to determine the retry interval.

If your use case is more complex, or the number of potential polling requests unacceptable due to the scale, the next solution can help.

WebSockets

This is by far the most complex solution of the list so far, but also the most powerful and likely flexible. It consists of leveraging Websockets and the way you communicate with the client via the backend of your application.

WebSockets or the WebSocket API is a way to enable bi-directional communication between the client — usually the browser — and a server. This way, the client can send messages to the server and receive responses without having to rely on polling the server.

The problems it tries to solve are the same as the UI Poller, but addresses the limitations when it comes down to scale — such as too many clients — and the potential variability of the execution time.

Figure 12 illustrates the solution using WebSockets and has the following flow:

- The client provides the information for the operation to take place
- The client establishes the WebSocket connection with the backend
- The backend starts the operation
- The backend finishes the operation and sends the information back to the client

This solves the limitations of the previous solution. There are no additional and useless polling requests from the client, which means once the information is ready, the server will push it to the client. Additionally, it also enables more complex cases where a single operation can have many individual stages as seen in Figure 12.

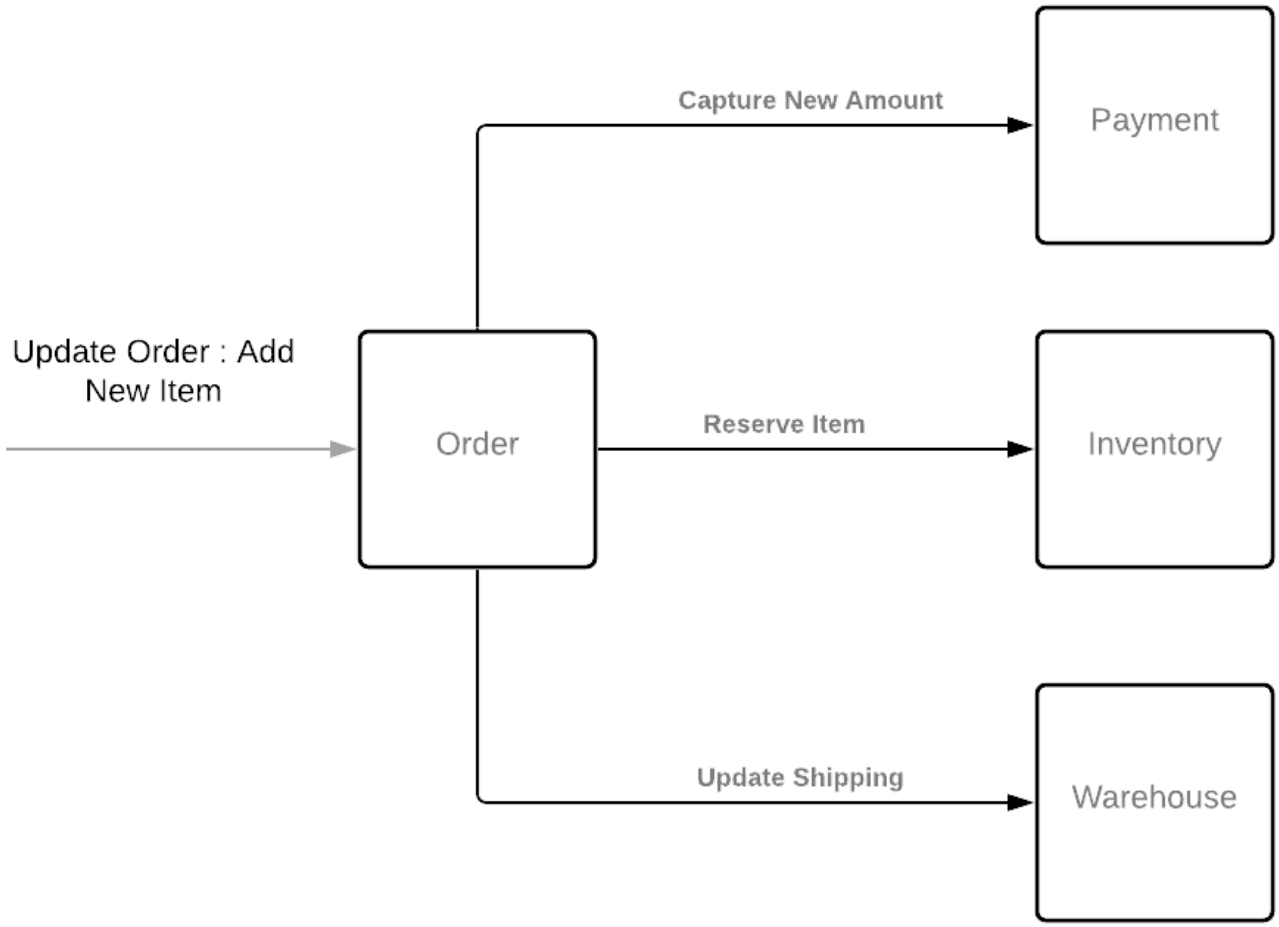


Figure 12. A complex case involves several steps.

In this example, a Customer Care Agent is updating an Order and as part of the operation, you are expected to capture a new amount, reserve the stock, and update the shipping information at the warehouse.

Now let's look at the components that could be associated with enabling this solution while leveraging AWS.

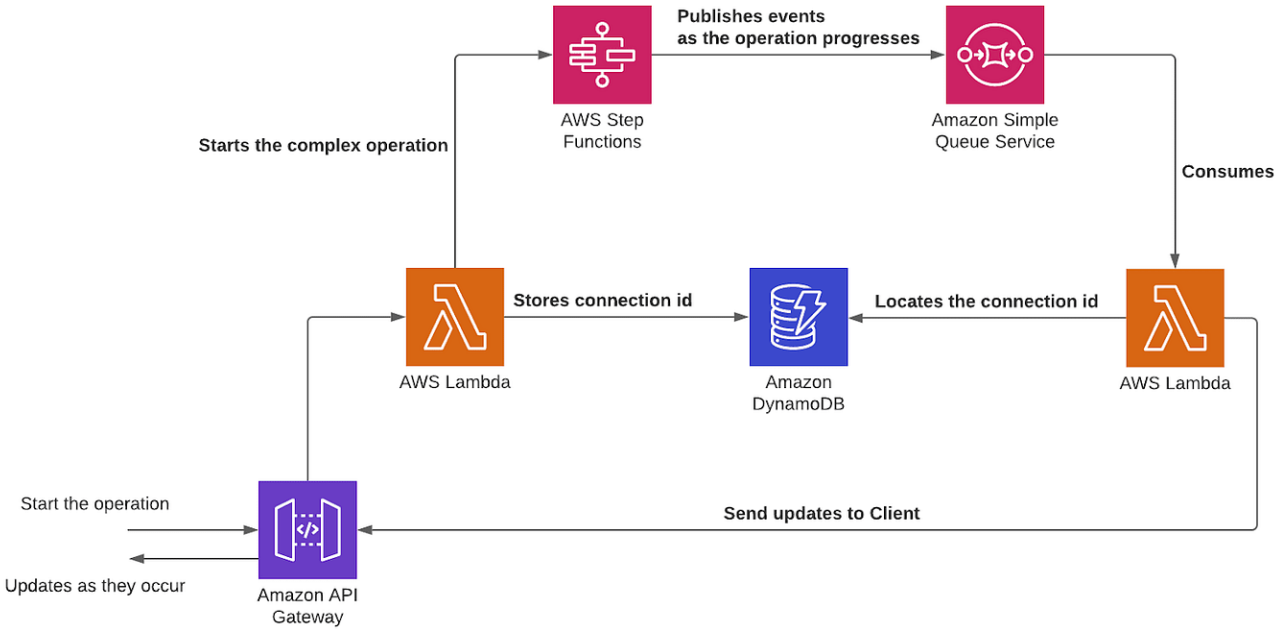


Figure 13. One way to leverage WebSockets in a complex operation.

API Gateway

- Receives the request and manages the WebSocket
- Routes the request to the appropriate Lambda

Lambda A

- Validates the request
- Stores the connection id in DynamoDB
- Starts the Step Function

DynamoDB

- Stores the connection id associated with the WebSocket

Step Function

- Coordinates the execution of the microservices
- Inform of the updates via SQS

SQS

- Contains the messages representing the communication the service wants to provide back to the client

Lambda B

- Receives the messages from SQS, locates the connection id, and sends the content back to the Client via the API Gateway.

As you can see, it is a much more complex solution than the previous ones but extremely powerful as it allows you to send continuous feedback to the client as the execution takes place.

It is important to note that although not shown, the code behind a production application is expected to handle failures, such as when the client is no longer connected or attempts to reconnect.

Wrap Up

Eventual consistency is a reality to our systems and in most cases unavoidable. I presented some approaches you can leverage that do not try to remove the eventual consistency, but instead acknowledge its existence and incorporate ways to handle it.

When choosing which one — or which ones — to apply, always take into account your context and select the one that best serves the business needs. It will help you to avoid introducing unnecessary complexity or cost to your solution.

Distributed Systems | Eventual Consistency | AWS | Websocket | Serverless

Follow

Written by Mario Bittencourt

949 Followers · Editor for SSENSE-TECH

Principal Software Architect

More from Mario Bittencourt and SSENSE-TECH

Mario Bittencourt

Achieving Idempotency: There Are More Ways Than You Think

Software development is a complex endeavor and one of the certainties is that as soon as you expose your application/system to real traffic...

· 7 min read · Oct 16

272

Pablo Martinez in SSENSE-TECH

Hexagonal Architecture, there are always two sides to every story

The Hexagonal Architecture, also referred to as Ports and Adapters Architecture

7 min read · Jul 9, 2021

2.1K 25

Pablo Martinez in SSENSE-TECH

Dependency Injection Vs Dependency Inversion Vs Inversion of Control, Let's set the Record Straight

Dependency Injection, Dependency Inversion, and Inversion of Control are 3 terms that, although related, are commonly confused and...

8 min read · Feb 4, 2022

727 4

Mario Bittencourt in Better Programming

API vs Messaging—How to Choose Which One to Use?

We have been living in a distributed world for quite some time, and when considering how should you implement the communication between...

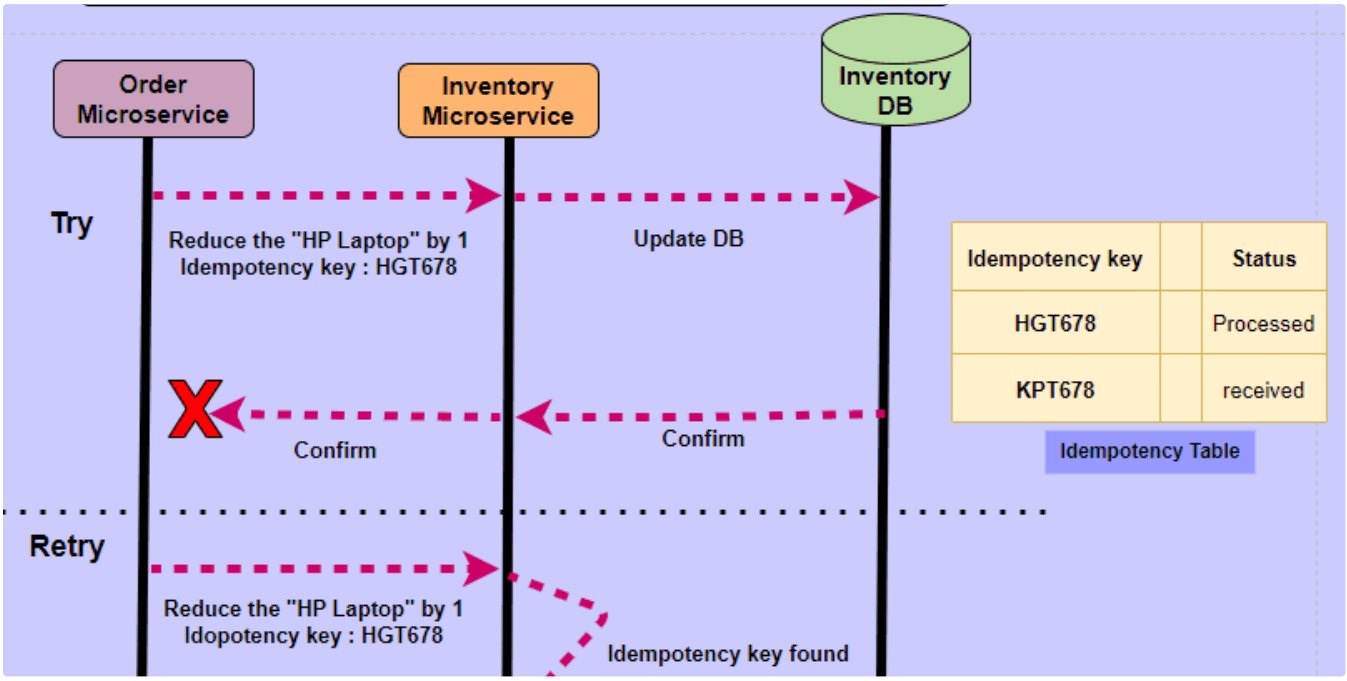
· 7 min read · Jun 13

225 3

See all from Mario Bittencourt

See all from SSENSE-TECH

Recommended from Medium



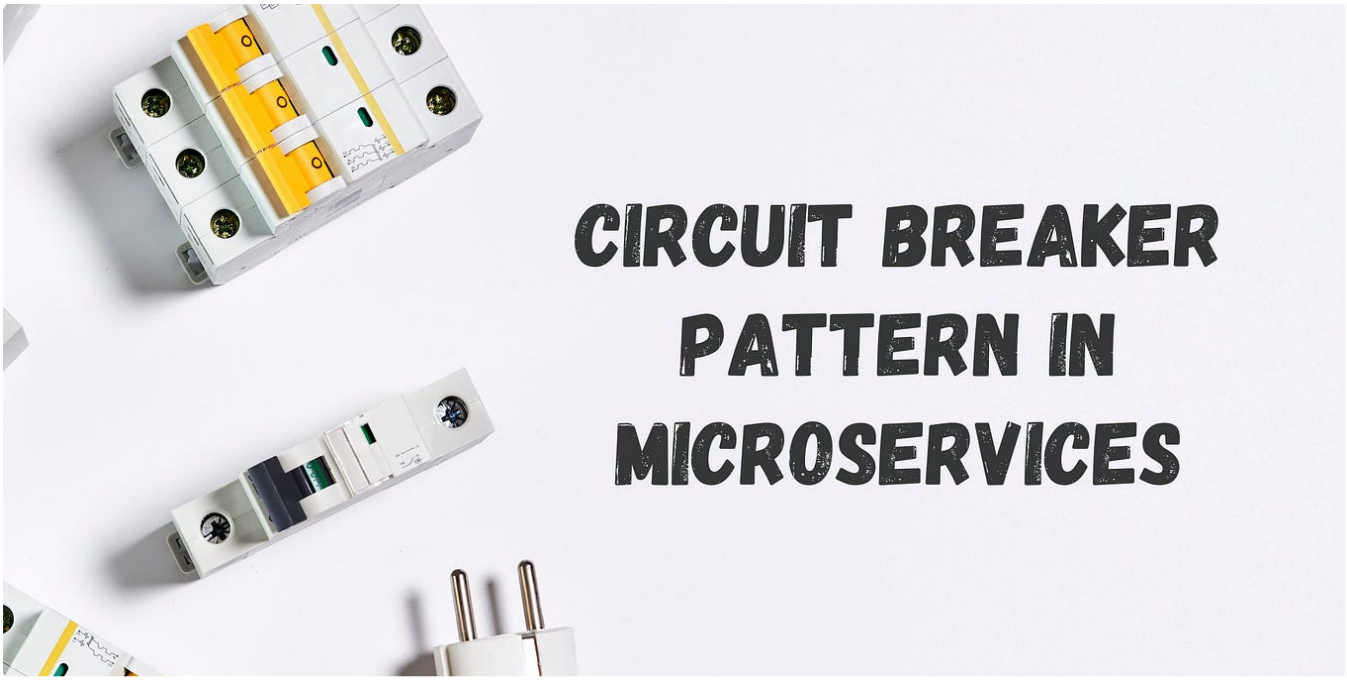
Debasis Das

Idempotency in Scalable Distributed Architectures—Example

1. What is Idempotence :

3 min read · Sep 28

36 1



Chameera Dulanga in Bits and Pieces

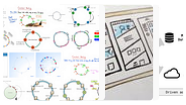
Circuit Breaker Pattern in Microservices

How to Use the Circuit Breaker Software Design Pattern to Build Microservices

6 min read · Jan 11

 465  2

Lists



General Coding Knowledge

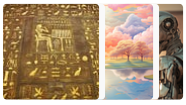
20 stories · 509 saves

Leadership

37 stories · 136 saves

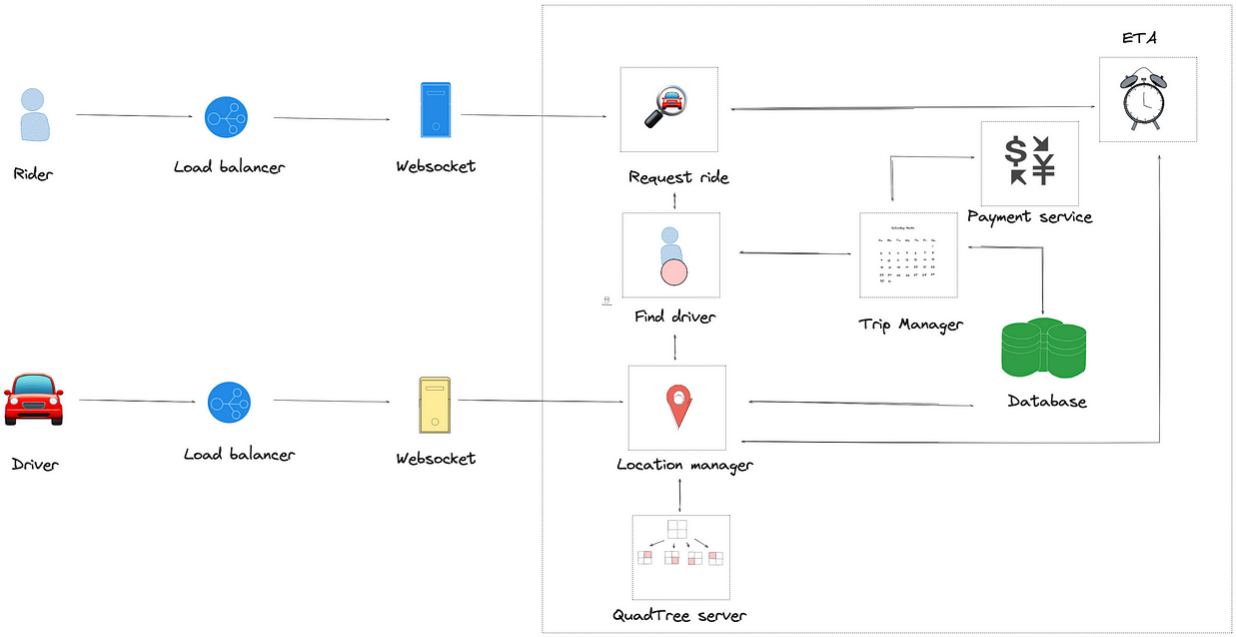
New_Reading_List

174 stories · 169 saves



Natural Language Processing

767 stories · 348 saves



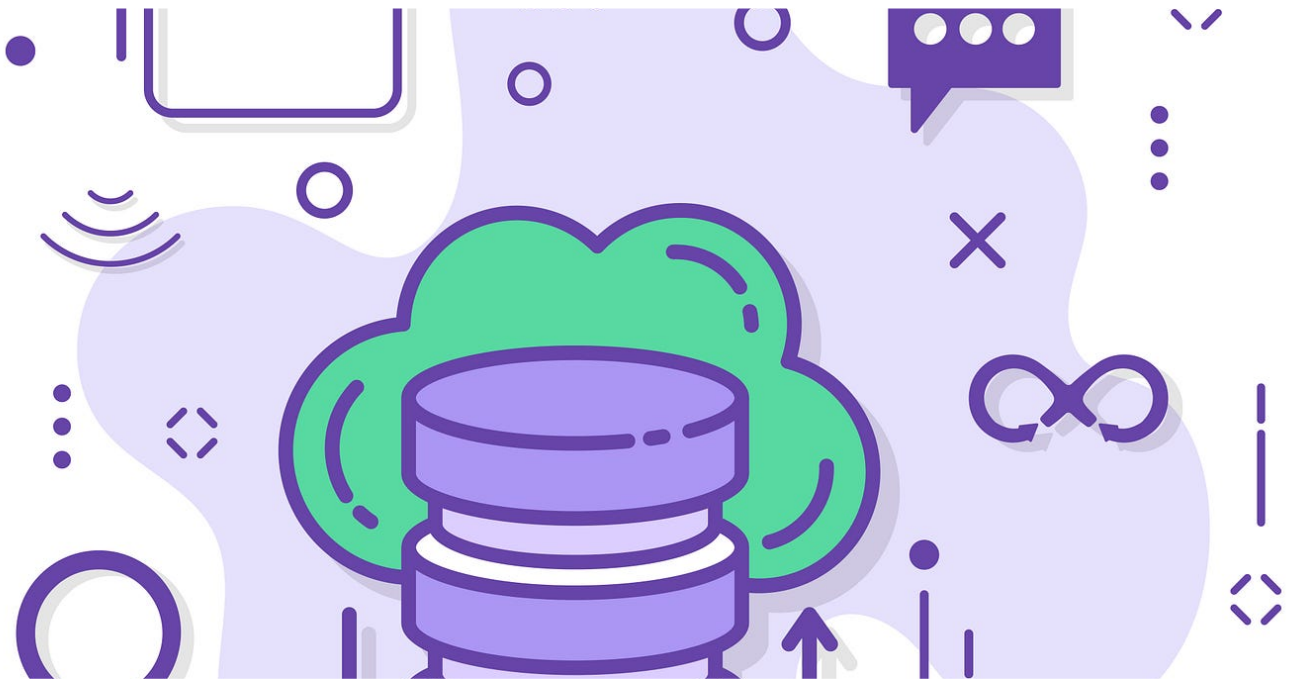
Suresh Podeti

System design: Uber

Introduction

7 min read · Oct 13

32 1



Héla Ben Khalfallah in ITNEXT

Deep dive into database internals

Architecture, storage and data structure

14 min read · May 4

129

Mahith Narayanan in Cloud Native Daily

Choosing Between Event-Based Architecture and Request-Based Architecture in Microservices...

Factors to consider when deciding between Event-Based Architecture and Request-Based Architecture for developing your microservices.

4 min read · Jun 26

77 5

mayank bansal

Hotel Booking App—Design

Design for Airbnb, Agoda, Booking.com

7 min read · Aug 25

35 2

See more recommendations