How Event-Sourcing & Serverless have super-powered E-commerce systems

Santosh Hari

Santosh Hari

Sr. Consultant @ Microsoft

https://linkedin.com/in/santoshhari

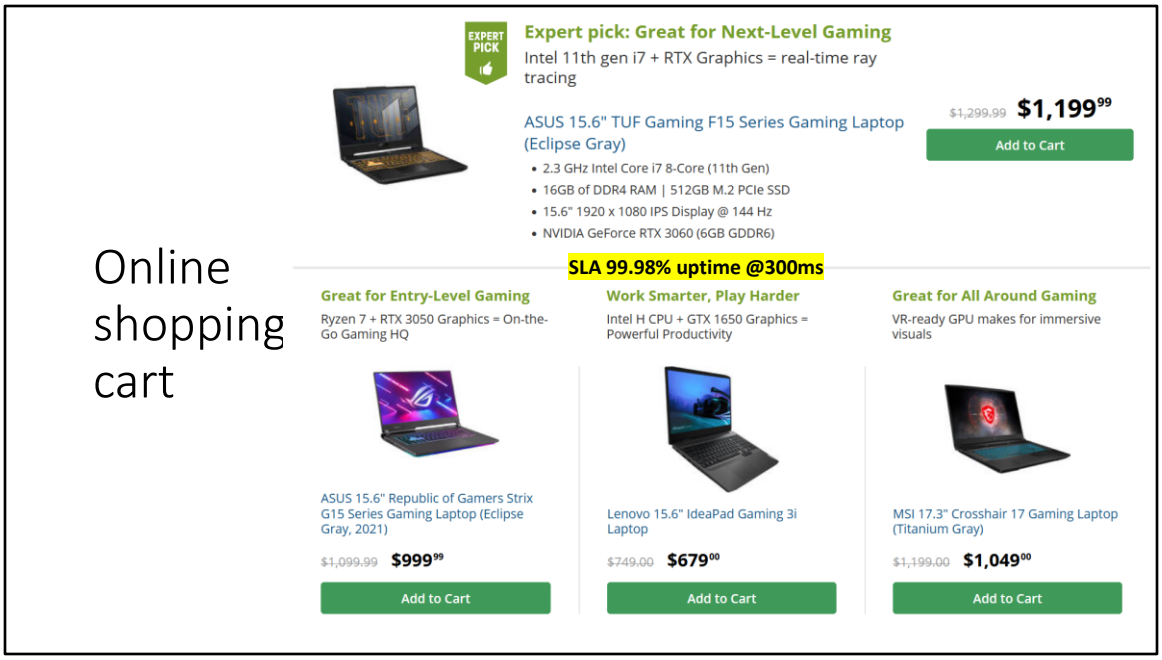
Twitter: @_s_hari

Agenda

- E-commerce: It’s complicated
- Modern principles - event sourcing, serverless, change data capture, etc
- A basic implementation in Azure
- Event sourcing discussions and questions (Other topics too)

E-commerce: It’s complicated

Online shopping cart



This is a typical online shopping cart
Has been around for over 2 decades and pretty much looks similar on most e-commerce websites
Customers browse the website for electronics, filter by laptops and then land on this page which
Shows some recommendations, probably powered by some nefarious tracking activity of the customer all over the internet

Item availability is served via an API
When she checks her fav ecomm website it can't be down or take too long to load
Because competitors website is just a tab away
SLA of 99.98% 2 hrs downtime per year, 300ms latency

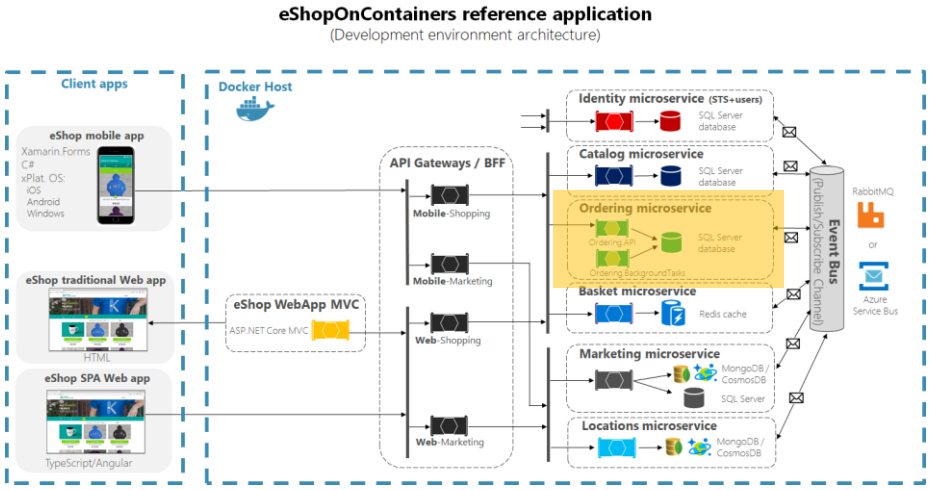
I will dig into one small piece of functionality
Customer wants to buy a laptop

The general process here is customer adds the laptop to cart

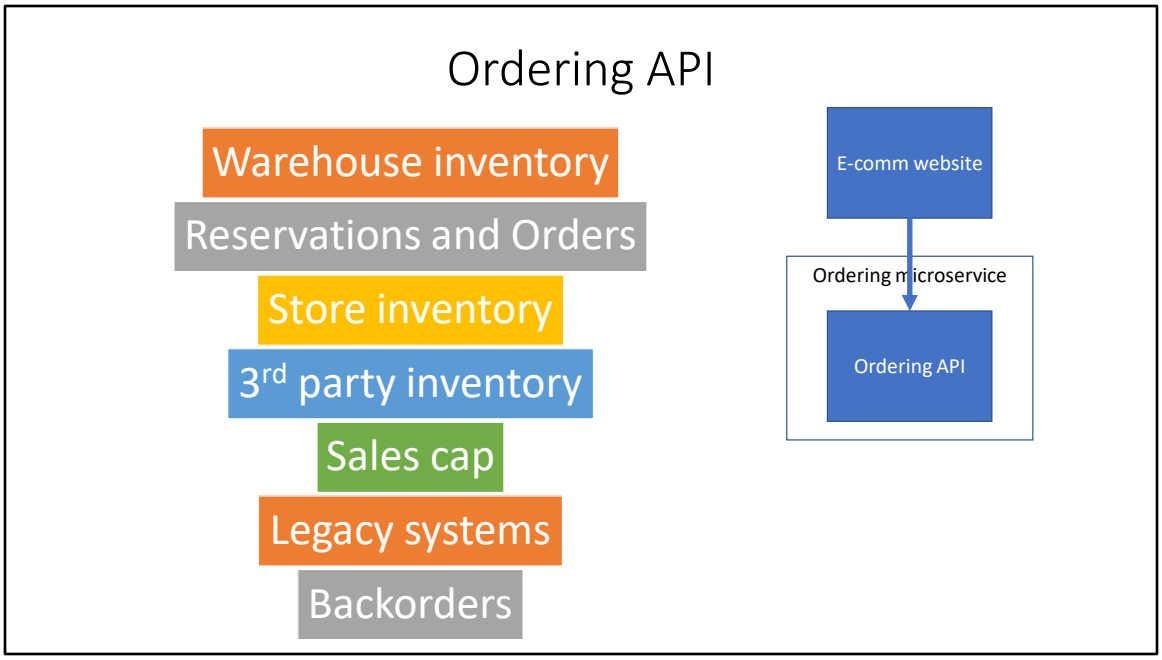
When ready customer checks out online, sets delivery address and pays for the laptop

Most e-commerce system really don’t reserve a product until you go into the checkout page
We’re freezing at the moment you’re about to checkout
IS THE LAPTOP STILL AVAILABLE?
Is it available either online or in a nearby store?

Reference e-comm architecture



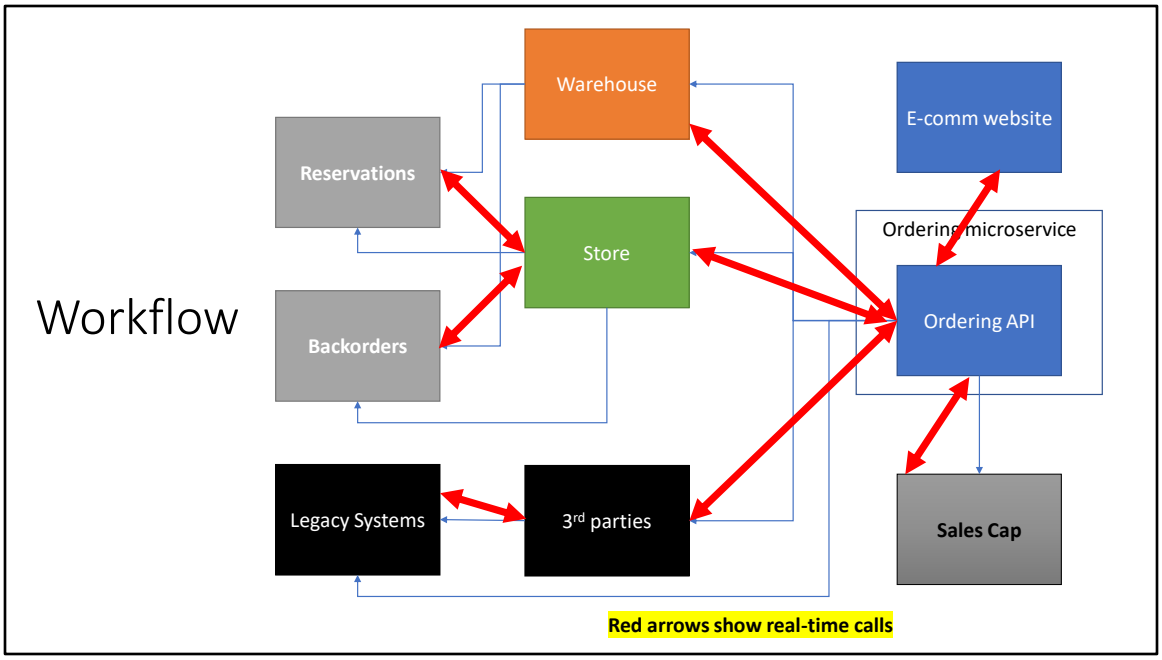
We're used to looking at architectures like eshopcontainers
which are pretty awesome for small to medium size businesses
You have identity, catalog, ordering, basket and other microservices.
But when it comes to larger e-commerce systems each component gets more complicated
And if you're not careful even these can get real gnarly
For instance let's drill into the ordering microservice



In this example the real life version of the ordering microservice is actually an entire org
You hear about supply chain these days all the time:
When the e-comm website invokes the global ordering API
It has to check with all these microservices:

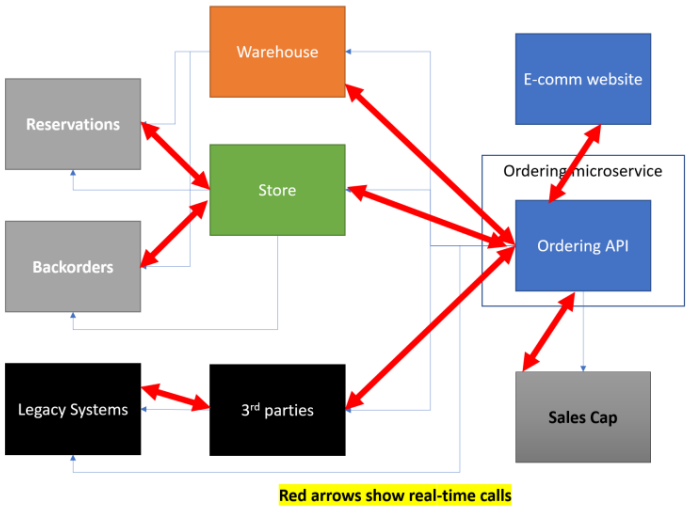
Warehouse inventory
Existing reservations and orders
Store inventory – floor and backroom
3rd party inventory – partners and suppliers
Sales cap – like seasonal limits for XBoxes
Legacy systems which may also contain duplicates that need to be checked
Backorders that were not fulfilled

So what does this look like?



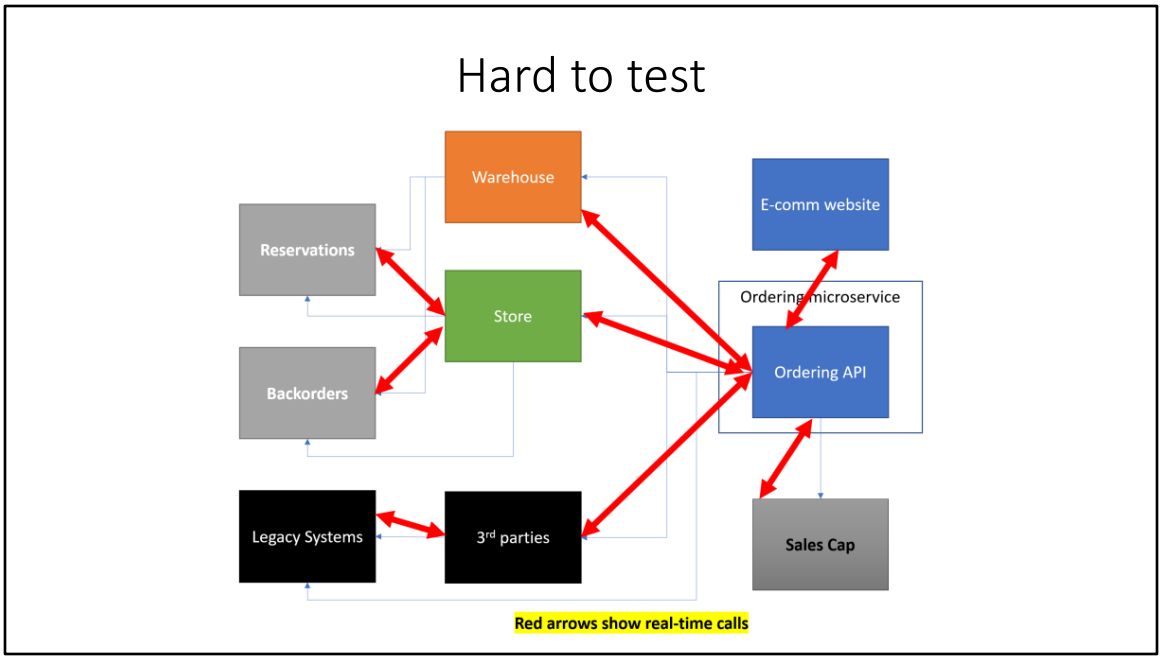
This is somewhat it looks like
Each is represented by a service (each box is a system)
Each may be maintained by one or more teams,
Each has it's own service or more likely multiple services and databases
What happens when customer wants to place the order for the the laptop
Ordering API call
Which calls warehouse inventory
Not finding anything it calls the store inventory where it finds a match
But we immediately have to check the reservations and backorders to make sure the item is still available
And then we have to check 3rd party API which has to make a call to legacy systems in this case to check for duplicates
After all said and done we still have to check to make sure the item falls within sales cap
We make multiple calls including a ton of synchronous API calls until we can finally compute the answer

Each service is a SPoF



Some of the problems with this architecture
System has 99.98% uptime and 300ms response means each of these systems has to be about better than that
Which means 5 9s of uptime and around 20-50ms latency

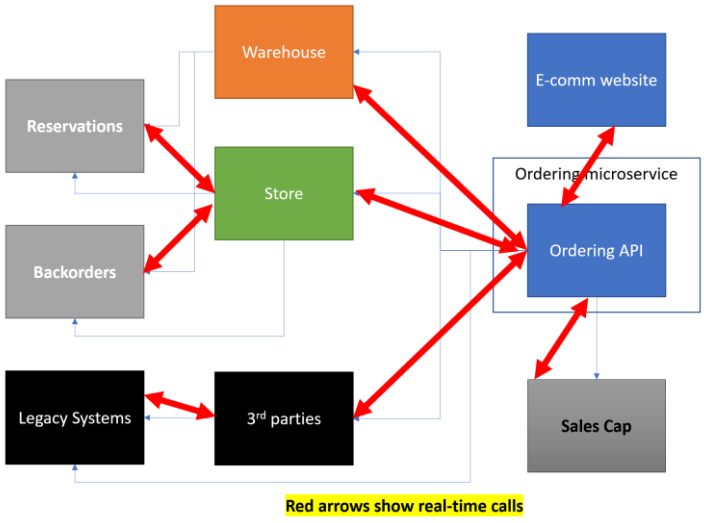
Additionally for every call, each of these services needs to work correctly
Even if one service goes down you risk giving the wrong answer to the customer
Aka one service takes down the entire system



Hard to test the entire dependency tree

Sure you can do unit tests which covers a small unconnected surface
the integration tests are a nightmare, hard to execute, complicated and expensive

Current state is inefficiently stored



Each of these systems was designed internally in a traditional manner

As changes happen the state of each system, usually in relational databases has to be mutated in place
We're assuming that there are no outages and everything stays up all the time

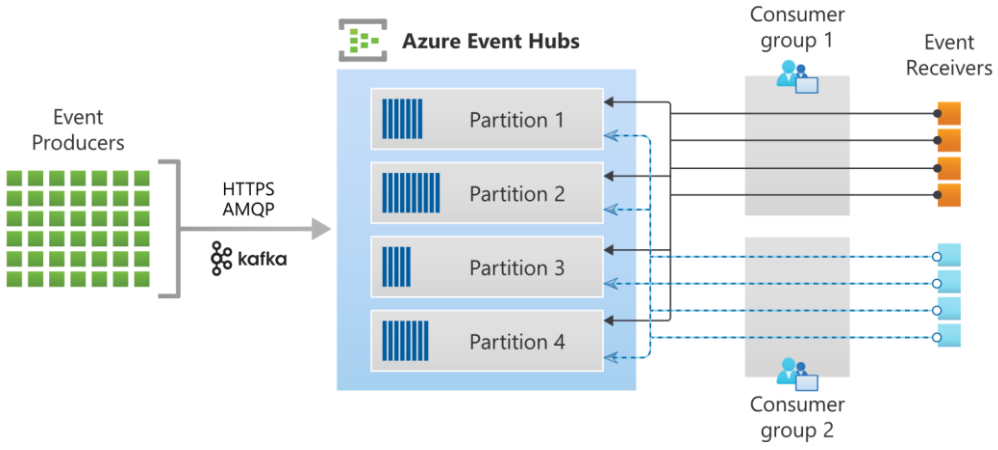
What happens when a component goes down or a bug is introduced

This is code, distributed unreliable, expensive code

To put it bluntly, servers are not reliable and we have to account for downtime

Modern principles - event sourcing, serverless, change data capture, etc

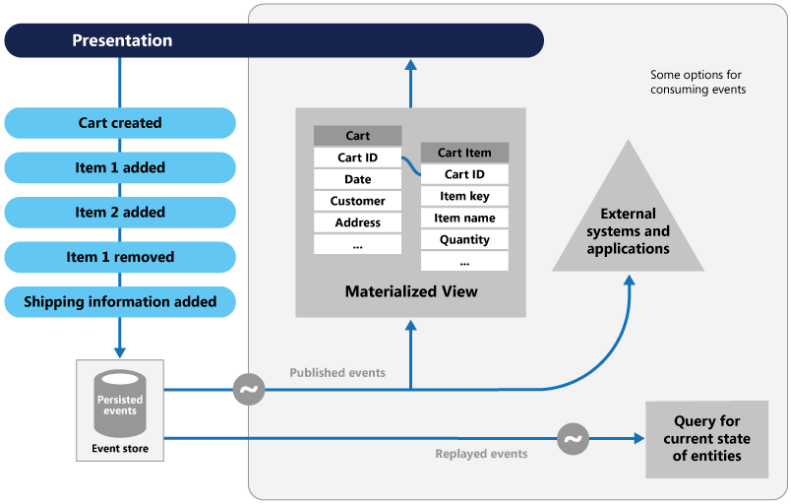
Switch to message and log based communication



Systems comm with each other over http or messages using queue
Log based pub sub systems like Event Hub and Kafka messages –
order and retained for later consmption using checkpoiints
If you have an outage or bug, you can reset checkpoint to point in time before the issue
Replace http API calls with event hub or kafka

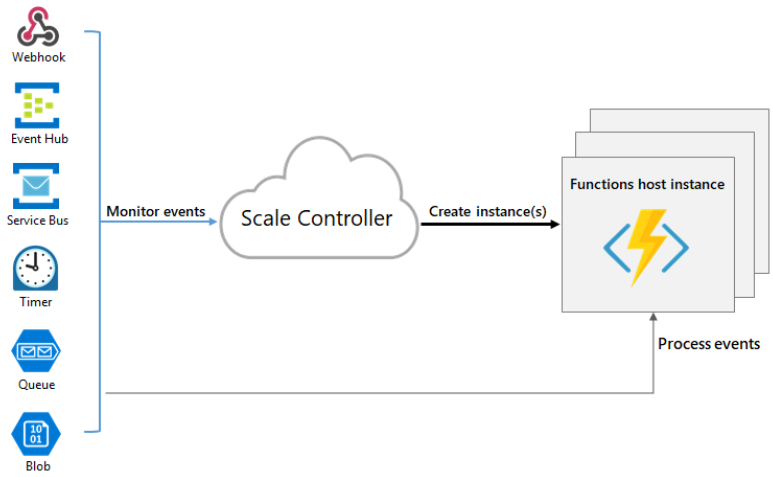
What problem does this solve? Drastically improves mean time to recovery

Implement Event Sourcing



Events are facts about something that happened in the world
Doesn't change by fact already happened
Events are 1st class citizens
Dat source is ordered set of events
Current state is ordered set of events – aggregating stream
Bank account is result of every depostit and withdraawael
Time machine you can see the state for any point in time and walk step by step for every state
What problem does this solve? Easy to replicate state at any point in time. Good for debggin and analytics

Leverage Serverless functions for Stateless computations



All business logic is stateless funitons with zero external dependencies

Collect all state you need up front and pass into code as parameters
The function is Predictabile and atomic in nature
No random outcomes or partial results
Real world failures will keep code from running but won't cause uncertain state

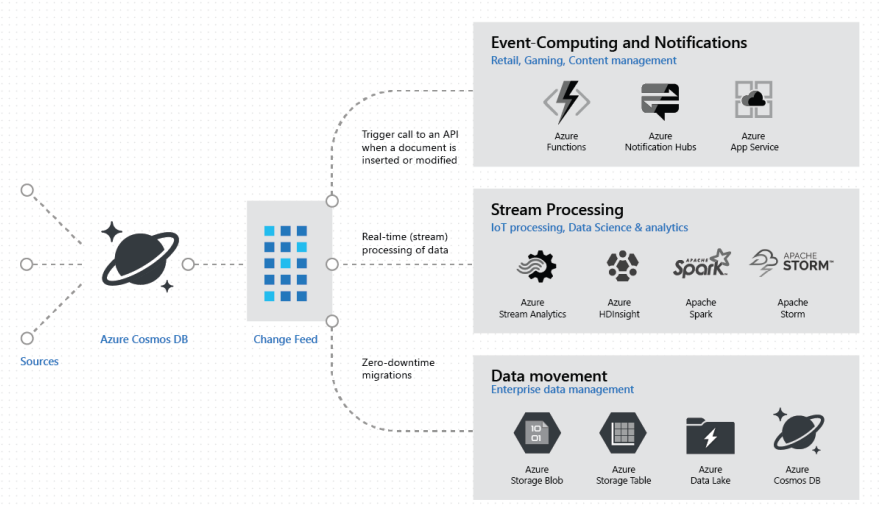
Domain logic Easy to unit test wide set of scenarios
Stateless code is easy to test
Integration tests only needed for testing connection between services

You must also consider a service like Function for the following reasons:

1. Atomic in operation
2. Scaleable
3. Integration code is abstracted away

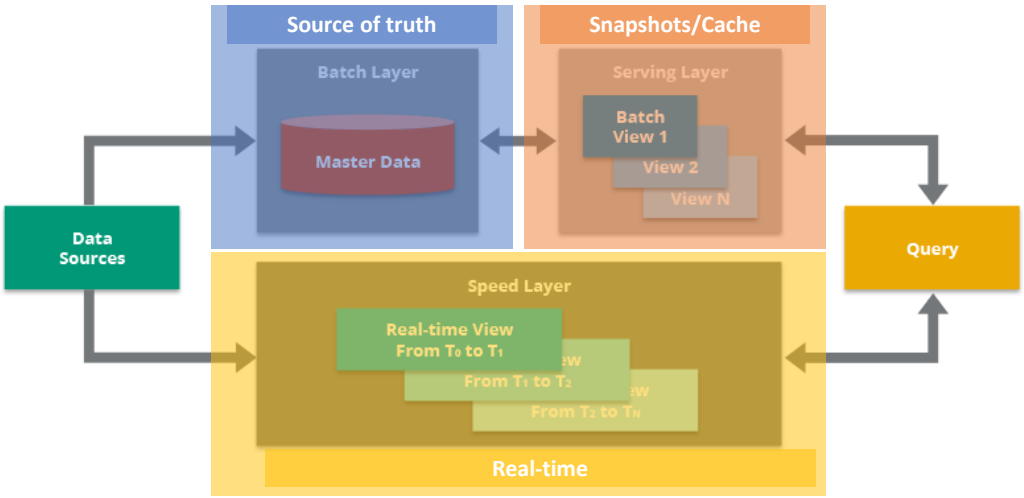
You're getting some state values, you run your calculation and provide results

Prevent dual writes with change data capture



Once business logic is complete then you have to write
Write-once (source of truth)
Don’t write to a database and then notify downstream consumers in the same process about changes – don’t
Dual write exposes us to failure and inconsistent state
Change data capture is the safe way to do this – only after commit event is published
You will never lose an event
Cosmos db change feed

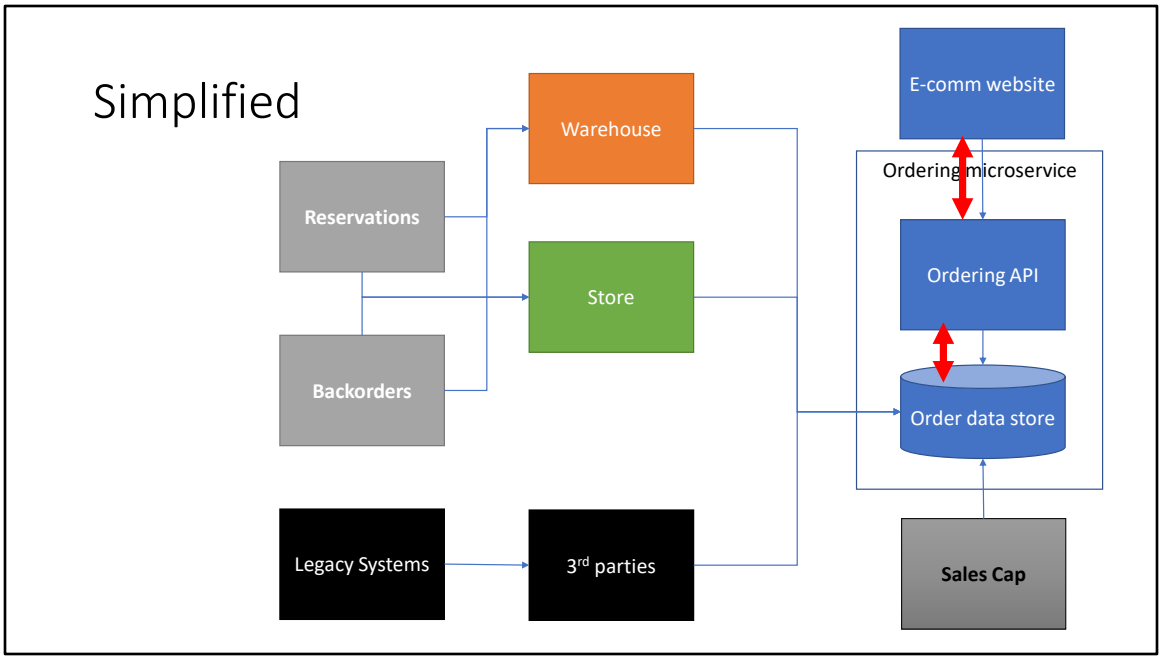
Replace real-time computations with data lookup



When you know possible set of inputs in advance
Instead of real-time computation, pre-computed cache of results
Real-time is expensive

This is an example of lambda architecture

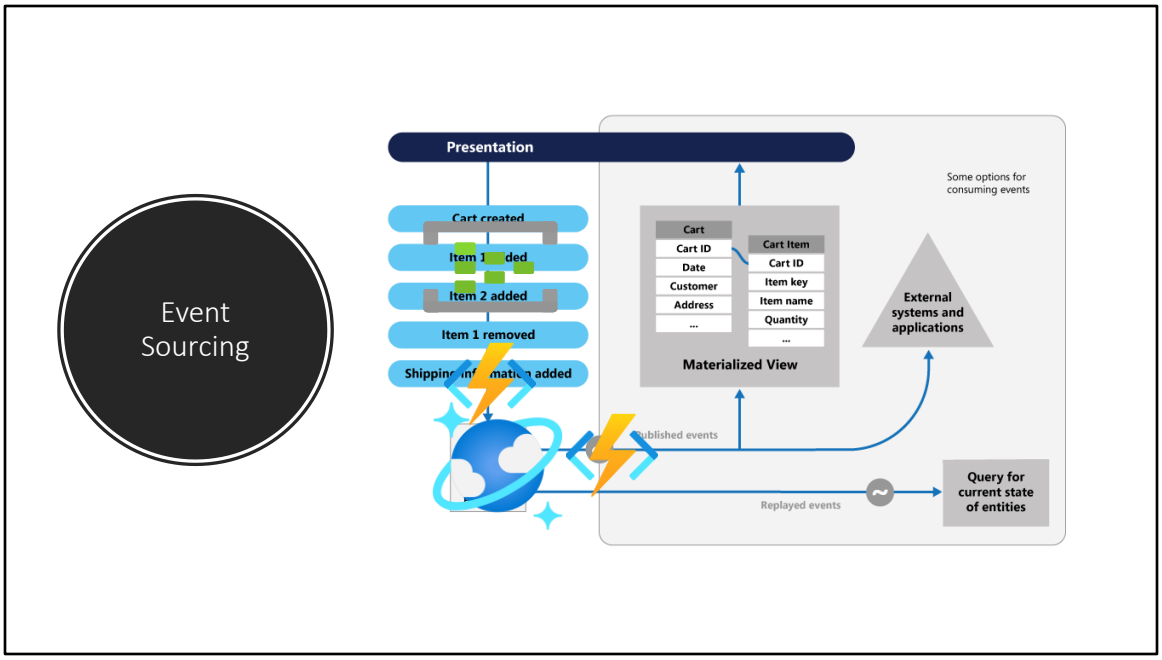
Save snapshots, builds updated stream and updates cache
pUblish snapshots as they occur
Downstream can consume real time or snapshots
Embrace eventual consistency



Real time calls in red

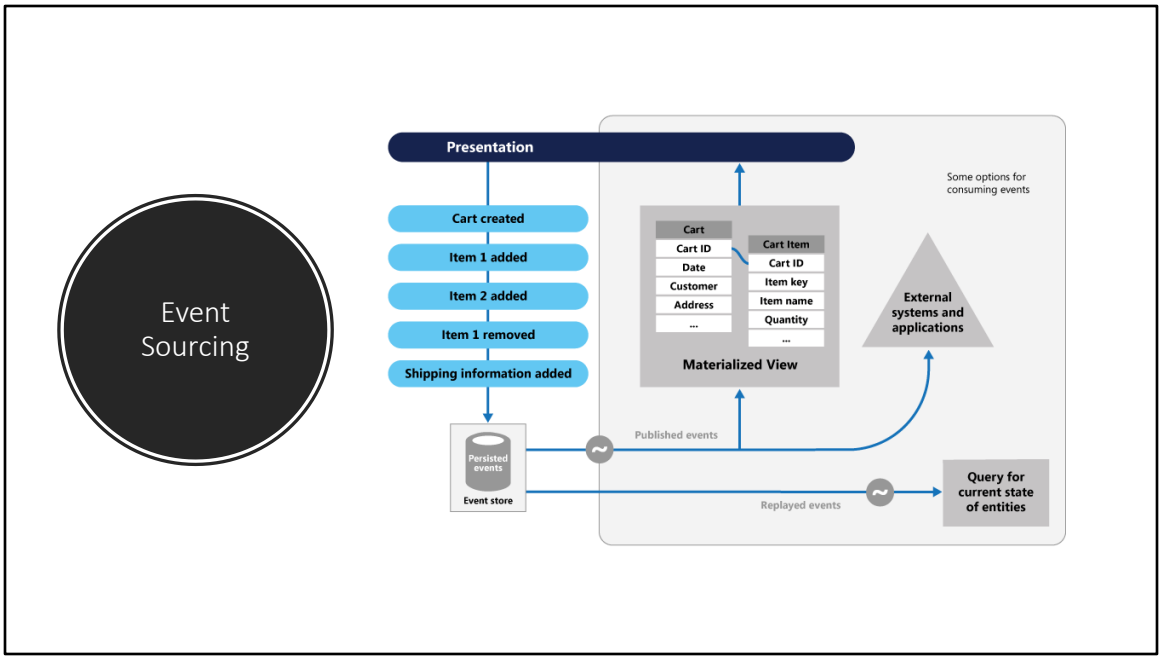
Using duality of code and data,
Stream events and messages over Event Hub
Instead of dependent service pulling the info,
Source system pushes data changes
From service arch to event driven arch
All components are async
Messages flowing from left to right
Trading real time comp to pre-computed data for supply chain
Hot path to calculate
2 service calls with 3 9s SLA with 150 ms

Event sourcing implementation in Azure with Serverless



Instead of storing just the current state of the data in a domain,
use an append-only store to record the full series of actions taken on that data.
The store acts as the system of record and can be used to materialize the domain objects.
This can simplify tasks in complex domains,
by avoiding the need to synchronize the data model and the business domain,
while improving performance, scalability, and responsiveness.
It can also provide consistency for transactional data,
maintain full audit trails and history that can enable compensating actions.

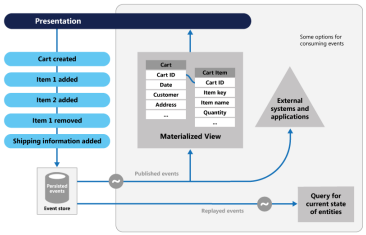
Event sourcing



Instead of storing just the current state of the data in a domain,
use an append-only store to record the full series of actions taken on that data.
The store acts as the system of record and can be used to materialize the domain objects.
This can simplify tasks in complex domains,
by avoiding the need to synchronize the data model and the business domain,
while improving performance, scalability, and responsiveness.
It can also provide consistency for transactional data,
maintain full audit trails and history that can enable compensating actions.

What problems does Event Sourcing solve?

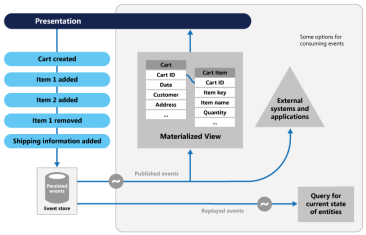
- Current state only
- Locking
- Data loss
- Audit??



Most apps work with only current state of data. For instance, customer status
Transaction often lock the data leading to data storage during transaction lock
Data conflicts can cause data state loss during conflict resolution
Needs separate audit mechanism

How Event Sourcing solves these problems

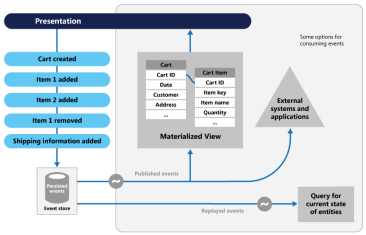
- Data-as-events-sequence
- Append-only store
- Store is authoritative
- Store publishes events
- Materialized View
- Replay events



Approach centers around handling operations on data that's driven by a sequence of events,
each of which is recorded in an append-only store.
Stores acts as authoritative about current state also
Stored publishes events for consumers – publishing code decoupled from subscribing code
Events published can be used to maintain Mat Views which are updated with new data
Apps also read history of events to materialize current state

Event Sourcing advantages

- Immutable events
- Simple objects
- Domain experts friendly
- No direct DB updates
- Audit trail
- Decouple events from tasks



1 Events are immutable changed using append-only ops.
Updating UI/Workflow can continue and event handlers backgrounded. Superior perf & scalability

2 Simple objects describing action with associated data.
Data store not directly updated. Simplifies implementation and management

3 Events have meaning for domain experts
Whereas OR impedance mismatch in databases are complex.
Also tables are often current state, not all events

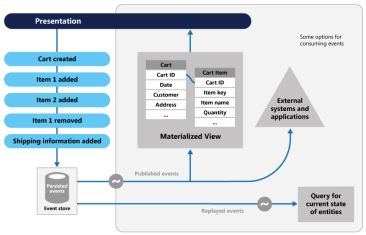
4 No direct datastore updates, prevent concurrency

5 Append only allows audit trail – allows monitoring updates, replaying events, regenerating current state

6 Events decoupled from tasks that handle them provide scalability and flexibility
Tasks know about type of event and data but not what op triggered event
One event can have multiple tasks

Event Sourcing considerations

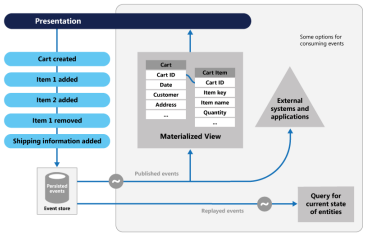
- **Eventual Consistency**
- No out-of-band updates to event store
- Event format vital
- Order of events vital
- Current State = sum all
- Use snapshots
- Reduces data conflicts
- Consumers idempotent



When materializing views, system will be eventually consistent
Event store should not be updated with anything else since it's source of truth
If format of event changes, difficult to combine events before and after
Consistency and order of events in store is vital
Timestamping events is a must
Current state can only be determined by reading all events in history
If event stream is large, consider creating snapshots at intervals – current state from snapshot + replaying events
Data conflicts reduced by eventual consistency and lack of transactions need to be handled by app
Consumers idempotent, so event updates should not be reapplied

Event Sourcing use cases - good

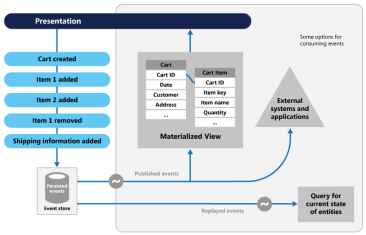
- Capture intent, purpose, reason
- Avoiding conflicts
- Restore / rollback
- Decouple I/P O/P
- Great with CQRS



capture intent, purpose, or reason in the data
When avoiding conflicts is vital
Record and replay event to restore state of a system and rollback or history/audit
Need to decouple input/updates from tasks that respond to these
In conjunction with CQRS, for systems that are ok with eventual consistency

Event Sourcing use cases - bad

- Simple domain
- CRUD
- Consistent systems
- Real-time systems
- Non-audit non-historical systems
- Low conflict systems



Small or simple domains, systems that have little or no business logic nondomain systems that naturally work well with traditional CRUD data management mechanisms.
Systems where consistency and real-time updates to the views of the data are required.
Systems where audit trails, history, and capabilities to roll back and replay actions are not required.
Systems where there's only a very low occurrence of conflicting updates to the underlying data. For example, systems that predominantly add data rather than updating it.

Santosh Hari

Sr. Consultant @ Microsoft

https://linkedin.com/in/santoshhari

Twitter: @_s_hari