MICROSERVICES

MICROSERVICES FOR APPLICATION SUPPLEMENTATION

Keep my COTS vanilla please!

Major organizations rely heavily on commercial off-the-shelf (COTS) or packaged applications, which come with pre-configured tested functionality as these are built on leading industry best practices. However, COTS applications do not always meet the full needs of businesses - and in some ways they cannot - because every business has its own unique differentiation. Sometimes the differentiation is a proposition, sometimes a way of selling, sometimes a way of servicing, but it is typically a level of uniqueness that cannot be readily encapsulated in a COTS application.

This aspect is being magnified today, since businesses are being challenged to increase their differentiation in a digitally enabled world where the ‘me too’ proposition can hold less and less weight in the market. In this backdrop, this paper details how microservices might be a better fit and how it can be a valuable supplement for existing COTS applications.

CONTEXT

Ignoring the interesting but rather academic debate of "*off-the-shelf*" vs "*custom-commissioned solution*", organisations tend to get locked into customizing packaged applications from power vendors that are neither one nor the other. The resulting implementations often exhibit significant problems of vendor lock-in, proprietary implementation and perpetuate monolithic architectures. The realization of solutions goes against domain-driven design approach that offers an alternative proposition.

The journey many organisations go on with COTS applications is having the stated objective to keep COTS vanilla, but having no concrete mechanism for realizing this objective. Consequently, the realization involves customization and often also business compromise. The reality of a compromise is that the system does not do exactly what is required and the business as a whole also operates in a way that is not exactly ideal. Another interesting facet is the understanding of the system is also suboptimal, for the reason that through the implementation of a COTS system that is customized it can be difficult for users and the business in general to understand where the COTS ends and where the customization begins. It is also worth noting that customisation often gets incorrectly badged as configuration in an attempt to obscure the real situation and this gets increasingly played out into the implementation on larger transformations where the boundaries become ill understood.

This paper describes the problem and illustrates an example reference architecture that allows for COTS to stay vanilla, and exploiting microservices in an architecture that exhibits:

- Separation in a domain-centric manner
- Ability to build scalable applications while failing fast
- Reducing the high switching cost of vendor lock-in
- Decentralizing data into Context Bound Domains, unlocking the latent value of data
- All while, empowering the topology to depend on the intellectual property invested in a COTS application where there is no need of a technological competitive edge (e.g.: a financial ledger for a retail company)

SOLUTION

THE FIRST STEP: Stick a principle in the IT rule book. *"Never customize a commercial product"*.

Without delving into the "what is a microservice question", there is a need to create a topology to cater to the following features, to enable engineering of functionally composed applications and services.

- Ability to access the data stored in the premium and commercial applications
- Ability to De-couple packaged applications from user journeys, services and interactions (which do not belong to the context of the packaged application). This is beneficial even if the use of COTS application is extensive compared to microservices
- Ability to reduce the IT footprint (reduce the capital cost – servers, core licenses, infrastructure, and the operational cost – smaller focused team, more effective traceability to package features; all while promoting more business value on commodity infrastructure, on-demand, server-less, pay-per-usage, pay-as-you-go paradigms)
- Ability to promote agility, product development without getting sucked into endgame fallacy

Building a reference architecture inclusive of packaged applications with principles of event-driven and microservices .

Unlocking the context agnostic data in COTS application.

One very pertinent characteristic of any microservice implementation is "Decentralized Data Management". To arrive at that, we need a distributed message bin or message store to be able to store a complete log of all the changes of the data from the COTS application.

A distributed message store is required to allow for the following features on the layers stacked on this storage layer

- The stacked layers on this layer of storage should be repeatable and reproducible on-demand
- Fast and ability to add/remove data at pace to cater to changes to business requirements

- Effective Infrastructure management, reproducible caches, failsafe mechanisms and extremely fast recovery times and high percentage SLA's

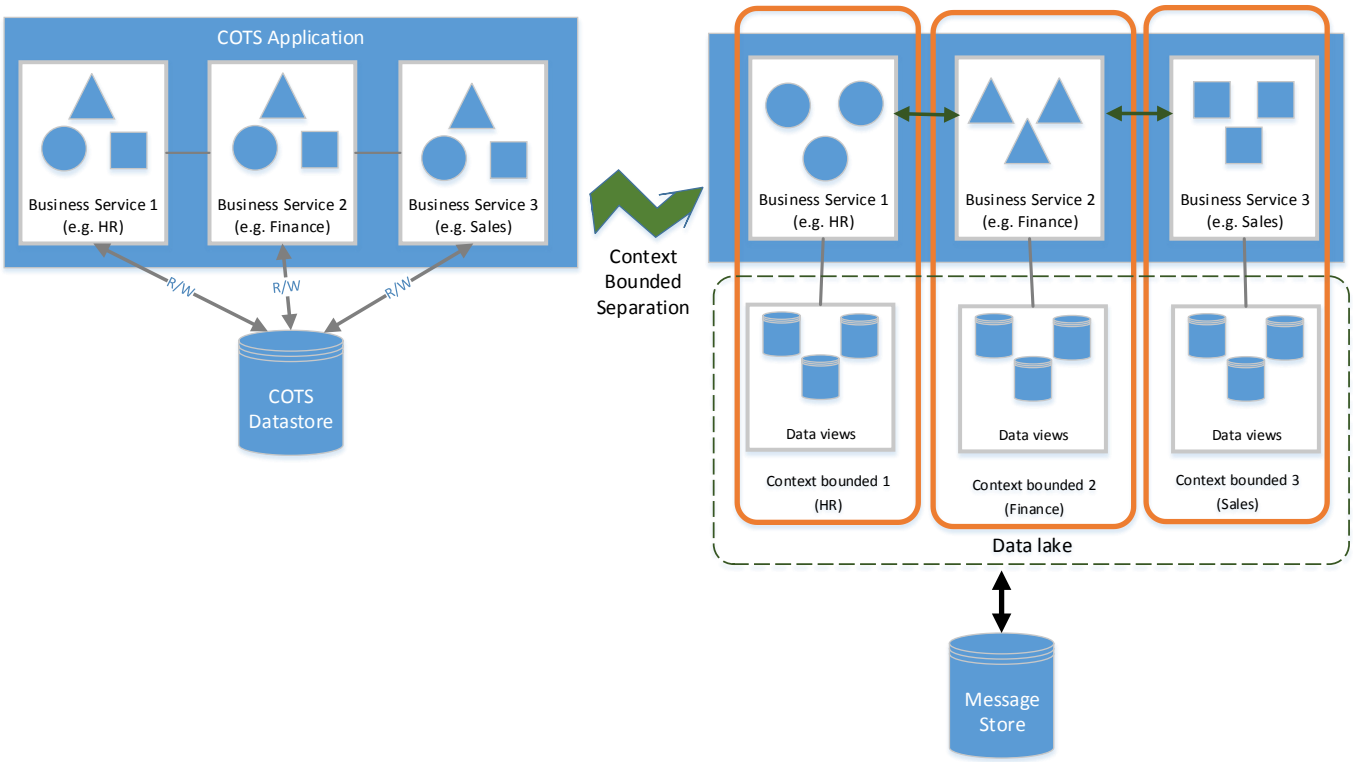
The distributed message stores will contain the same context unaware, COTS application pertinent data objects stored as a sequence of messages. The store can be populated by various means – messaging, CDC, remote procedures, COTS propriety access /protocols, events, API's.

Decomposing the data based on context bound domains.

This is essentially how effectively we fragment this data into functionally composed bounded context views of information that can be interpreted, scaled, modified, and actioned independently across different domains in the organization. These fragmented views of context bound information form the organization's data lake.

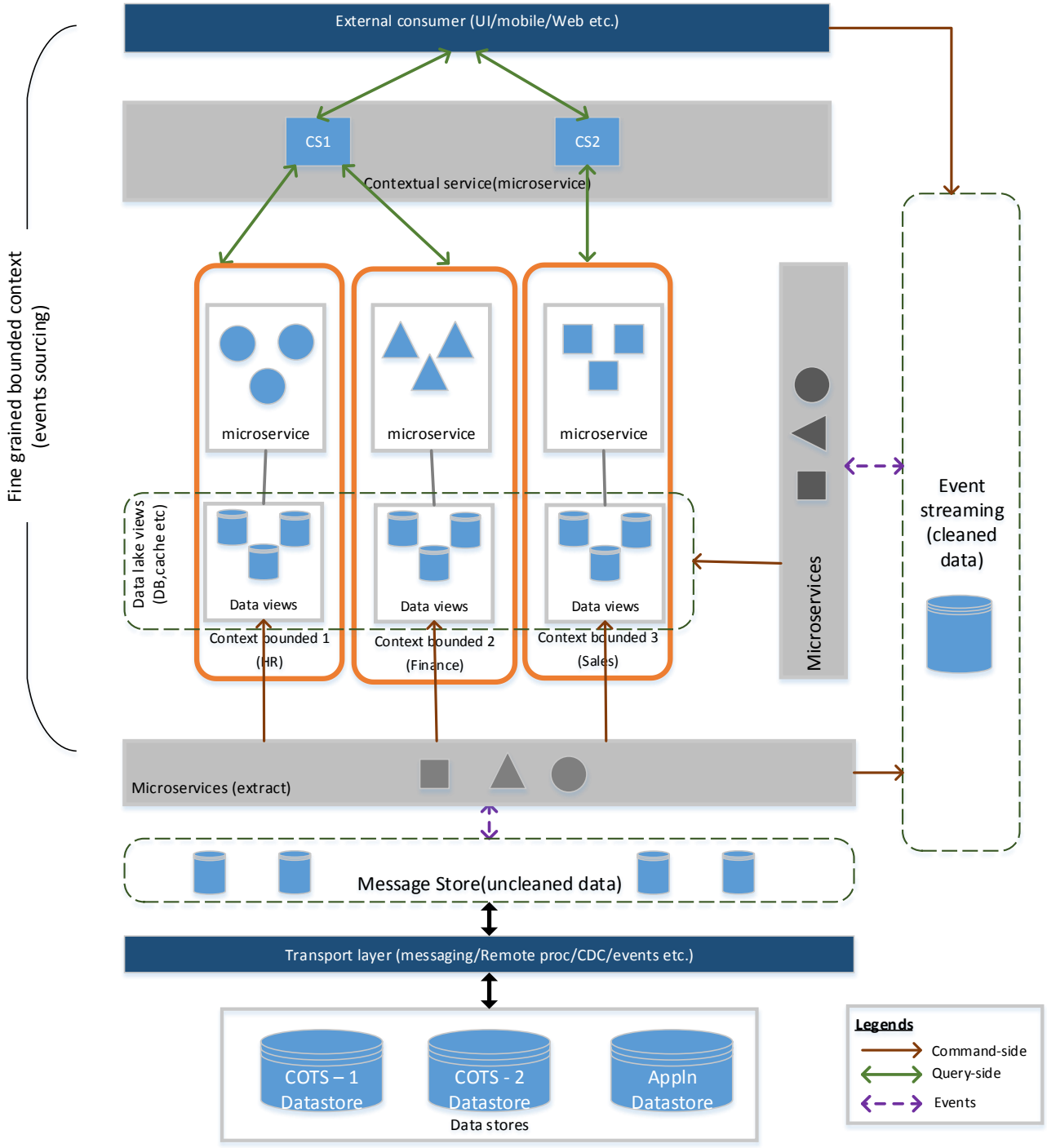
Business context should be identified explicitly when we build our software and model our data. We build and refine a model that represents our domain and that model is contained within a boundary that defines our context.

The following abstracted representation attempts at a comparison of Domain Driven Design vs Context Agnostic Design:



Context bounded separation

Engineering microservices on top of the data lake.



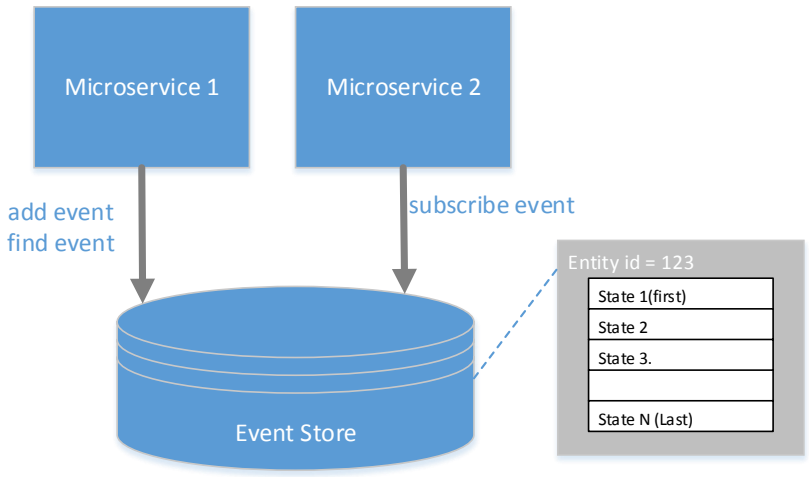
Microservice reference architecture

- The message store in the topology is the state and the point up to which data is “unclean” and “unaware”
- Any microservice above the message store, will need to be built within a bounded context and be context aware (whether it is a microservice serving up an API, or an UI, or a mobile app), it will need to emit fine grained context bound events. Any change that is effected to the data views has to be an event into the event stream. Event sourcing is explained in below section
- A separate microservice (or cluster of microservices) can stream these events and update or create new views in the data lake. A query side microservice will expose information in standard mechanisms to be consumed by contextual microservices or client applications. Some of these standard mechanisms are Rest API, Graph API’s, Atom, thrift
- The events in the event stream can be consumed by command-side microservices that is part of other domains if the information that has changed is pertinent to them. E.g.: if a priority of a supplier is changed in the support domain, it may be relevant in the supplier planning domain

- The events can be used to recreate and create repeatable views, the events can be sourced into a data warehousing and data science platform where statistics and intelligence can drive business decisions (e.g.: a supplier has made too many changes to tray sizes in the last week , so need to change the priority of the supplier)
- Based on the business demands, a new or existing data views can be created or refreshed from message store and event store using command-side microservices

Event Sourcing:

Microservices persist events in an event store, which is a database of events. The store has an API for adding and retrieving an entity's events. The event store also behaves like a message broker. It provides an API that enables services to subscribe to events. When a service saves an event in the event store, it is delivered to all interested subscribers.



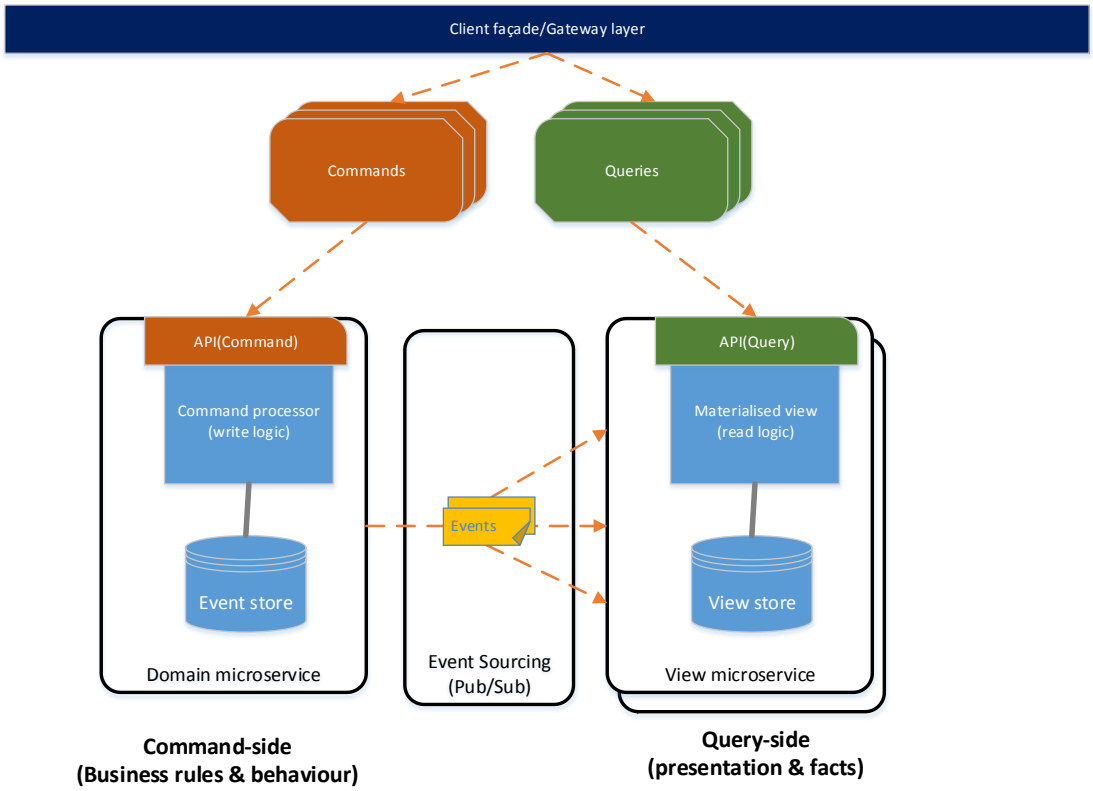
Event sourcing/streaming

Some entities, can have a large number of events. In order to optimize loading, an application can periodically save a snapshot of an entity's current state. To reconstruct the current state, the application finds the most recent snapshot and the events that have occurred since that snapshot. As a result, there are fewer events to replay.

The event store is difficult to query since it requires typical queries to reconstruct the state of the business entities. That is likely to be complex and inefficient. As a result, the service must use Command Query Responsibility Segregation (CQRS) to implement queries. This in turn means that services must handle eventually consistent data. The Command Query Responsibility Segregation (CQRS) is often used with event sourcing.

Command Query Responsibility Segregation (CQRS)

CQRS is an architectural pattern for developing software whereby the system that are responsible for changing the system's state (write-parts) are physically separated from the system that manage view (read-part).



Command Query Responsibility Segregation (CQRS) using event sourcing pattern

All the communication between command-side and query-side microservices is purely event-driven based on “event sourcing” architectural style.

Applications business logic is implemented by aggregates. An aggregate does two things:

1. Processes commands and returns events,
2. Consumes events, which updates its state.

Command-side

A command-side microservice module processes update requests such as HTTP POSTs, PUTs, and DELETEs. It consists of the following components:

- Aggregates and their associated commands
- Services, which create and update aggregates
- Event handlers, which subscribe to events and respond by creating and updating aggregates
- Snapshot strategies that create snapshots, which optimize the loading of aggregates

Query-side microservice:-

A query-side microservice module maintains a materialized view of command-side aggregates. It has one or more event handlers that subscribe to events published by command-side aggregates. The event handlers process each event by updating the view.

- Defining query-side event handlers

CHALLENGES

This approach helps a lot, but comes with its own challenges.

- Setting up microservice architecture requires strong engineering practice, DevOps, infrastructure as code etc
- Licensing cost associated on accessing the COTS data (directly or indirectly)
- There is stronger need arises for centralized monitoring and logging, to have an understanding what's going on in enterprise as the services are fragmented

OTHER CONSIDERATIONS

The reference architecture can be implied to exhibit some other characteristics that can potentially be of interest to a business aside from keeping COTS vanilla. These are implicit already, but it is worth discussing the most significant characteristics in turn.

In the technical topology of a COTS application, this architecture can enable the complete separation of domains even within the COTS application. This approach means that the modules in a COTS application can be integrated together in an externalised rather than internalised way – offering the complexity of the integration activity but the benefit of configurability that otherwise wouldn't be present. This is achieved through the use of master data and event data as an externalised service.

One of the major challenges COTS application implementations have relates to data understanding, data quality and general data management. The approach forces an information-centric approach as well as a domain-centric approach. This means that the data and the meaning of the data are intrinsic to the design, and that understanding must be sourced early. That is the very same understanding that often causes significant complexity in migration planning.

The proposed reference architecture can also support a staged migration between one system to another if a COTS system is being replaced. The externalisation of integration allows for a master/slave relationship where the mastering can be 'flipped' between the original system and the replacement system.

The reference architecture can act as an enabler for SaaS services where they can be integrated into the lifecycle; SaaS can also be utilised in a judicious manner by utilising slices of domain-centric functionality.

The architecture also allows for the separation of the actor and process, this is an important opportunity as it is an enabler for deeper process automation and Robotic Process Automation (RPA).

BRINGING IT ALL TOGETHER

Microservices can be the 'silver bullet' to keep COTS applications vanilla through the reference architecture described, and this can fulfil the benefits that brings. Judicious exploitation of a microservices architecture can be utilised to provide all the features that the business requires for its differentiation unencumbered by the compromises of customising a COTS application. The realisation of this approach is that the end-to-end value chains of the business intersect both the COTS application(s) and microservices to deliver the outcome. The roles of the COTS and microservices are also clearly segmented from a business perspective as follows:

- The COTS applications deliver business commodity (or potentially 'hygiene') capabilities or those with low business differentiation
- The microservices deliver business differentiation in capabilities that the business differentiates in
- The business potentially can now also have more control over the orchestration of these capabilities to respond in a more agile manner to the changing needs of the market. This is the ability to define and change the wiring that delivers the user experience

Businesses can also pursue different sourcing and management strategies around commodity and differentiating capability, and can also better control their own intellectual property related to business differentiation.

In addition to this there is an important side benefit – the Intellectual Property of a businesses' differentiation can be managed by the business with appropriate controls. Data becomes more readily available and also readily available in a bounded context, which arguably has a higher value enabling more intelligent decision making.

As a final note, this paper has primarily considered the problem in the context of keeping a COTS system vanilla; if a COTS system is not vanilla, but is integral to a business, the steps can still be followed to return it to a more vanilla state to enable/unlock flexibility that otherwise would not be easily afforded.

REFERENCES

1. Saw Newman (2015). Building Microservices – Define fine-grained systems, Sebastopol: O'Reilly [Book]
2. Irakli Nadareishvili, Ronnie Mitra, Matt McLarty & Mike Amundsen (2005) Microservice architecture, Sebastopol: O'Reilly [Book]
3. Wikipedia. Vendor lock-in [online] https://en.wikipedia.org/wiki/Vendor_lock-in
4. Martin Fowler. Domain Driven design. [online] https://martinfowler.com/tags/domain%20driven%20design.html
5. Chris Richardson. Microservice Architecture [online] http://microservices.io/
6. Wikipedia. Change data capture [online] https://en.wikipedia.org/wiki/Change_data_capture
7. Martin Fowler. Bounded Context [online] https://martinfowler.com/bliki/BoundedContext.html
8. Wikipedia. Data lake [online] https://en.wikipedia.org/wiki/Data_lake
9. Martin Fowler. Command Query Responsibility Segregation [online] https://martinfowler.com/bliki/CQRS.html