ITC INFOTECH

Business-friendly Solutions

# Leveraging Microservices for Unified Commerce


**Authored by ITC Infotech**

**Bharath KS**
Vice President
Product Engineering Services

**Alok Verma**
Senior Vice President
Retail & Global Strategic Partnerships

**Ronojit Mukherjee**
Senior Vice President
Hi Tech


As sales channels proliferated with increased adoption of social, mobile and cloud technologies, enabling omni channel commerce emerged as the single most critical need for brands across business domains.

However, this proliferation in the number of connected devices and interfaces requires a hyper connected, flexible and scalable technology architecture, enabling a seamless customer experiences ranging from browse-any-channel, buy-any-channel, market-any-channel, service-any-channel and so on.

# The Emergence of Unified Commerce

> In omni channel, you have multiple channels, but you don't have one piece of software, one version of the truth: You have many versions of the truth. In the unified commerce world, it's all connected in realtime. I don't just mean the web side, but the mobile side, the web side and the store side-all in real time.

*Ken Morris,*
*Co-Founder Boston Retail Partners*

Retailers, now, believe that in order to truly improve customer engagement and experience, a unified commerce environment is essential.  While some investment initiatives are targeted toward integrating software systems required to sell products across different channels into a unified platform, other initiatives aim at moving the needle on "unified commerce" further by bringing the entire retailing system from the store to website under a single, unified technology system.

# Forces driving the Unified Commerce Model

## Demand-side forces

As digital technologies enable deeper modes of engagement between the brand and the consumer, the nature of consumer demand is changing in fundamental ways.

- Online shopping has extended and enhanced the traditional storefront. Buyers expect a larger product assortment, ratings/reviews, more in-depth product descriptions, additional rich media, related products, tie-ins to social media, and so on.
- Consumers expect a seamless, connected experience across all channels (point of sale, web, mobile, kiosk, etc.). They expect to see the same inventory levels, product assortment, pricing, and other aspects, regardless of how they interact with a brand.
- Every Internet-connected consumer device and interface is a potential channel that consumers can use for shopping. New user interfaces are launching with amazing regularity, and successful brands must be able to extend their experience on every one of these new devices.

## Supply-side forces

We all agree that the nature of consumer demand is undergoing a series of fast-paced changes that require technology solutions that are capable of scale and flexibility,while providing traceability and governance. However, current commerce platforms and related IT organizations are not capable of addressing this transformation -

- Monolithic commerce systems do not scale – (i) complexity - it is simply too large for any developer to fully understand (ii) obstacle to agile development and deployment (iii) scaling the application is challenging – application modules have conflicting resource requirements (iv) reliability – as all modules are running within the same process, a bug in one module sometimes causes the entire application to crash (v) requires long-term commitment to a technology stack
- Strong-coupling across architecture- large, monolithic applications such as ERP, CRM, WMS, OMS, CMS, etc., expose different endpoints, which are not independently consumable, and need to be called in a specific order and fed specific data. That's why these monolithic applications are glued together by the use of enterprise service buses, with a lot of business logic residing in those buses. This tight coupling of large monolithic applications results in testing and releasing all monolithic applications together as an atomic unit.
- Strong-coupling across organization teams –enterprises establish teams with single focus that result in tight coupling between horizontal layers. For example, each user interface (point-of-sale, web, mobile, kiosk)has its own team. Respective UIs are tightly coupled to one or more applications, which are each owned by a separate team. Often, there's an integration team that glues together the different applications. Then, there's a database on which all teams are completely dependent. Infrastructure is managed by yet another team. These barriers cause tight coupling between teams, which introduces communication overhead and causes delays.

# Evolution of the Microservices Architectural Pattern

Microservices patterns have emerged as “Cloud Native” architectures have evolved with the growth of successful Web-Scale business models implemented by the internet giants of today – the likes of Google, Amazon, Facebook, eBay, Netflix, Twitter and a handful of others.

At its core, Microservices are individual pieces of business functionality that are independently developed, deployed, and managed by a small team of people from different disciplines.

Microservices support and drive the following design objectives:

| | | |
|---|---|---|
| Services are modular, small and easily maintainable | Each service owns its data, interaction via APIs | Services are independently deployable |
| Services can scale independently | Service teams function autonomously | Improved fault tolerance |
| Allows for polyglot programming and rapid experimentation | Allows for multiple versions of the service to coexist in the same environment | Choreography is preferred over Orchestration |

However, Microservices adoption introduces added complexity, and this is key to implementing solution roadmaps –

| | |
|---|---|
| Choosing the right set of services is often complicated | The process of decomposing a system into services is most often an iterative program, guided by the nature of domain workflows and business transactions |
| Managing distributed systems is inherently complex | Inter-process communication is pervasive. Managing operational complexity with multiple moving parts renders release and deployment automation is a necessity |
| Deploying features that span multiple services requires careful coordination | Deploying features that span multiple services requires coordination across multiple development teams |
| Shift to microservices can be a difficult timing decision | Adoption of the microservices architecture renders “rapid iteration” difficult – this can be a potential dilemma |

# Design Patterns Key to Implementing Microservices

At an aggregate level, the design of microservices requires a categorical separation of concerns – internal architecture of individual services, and the external architecture that involves infrastructure orchestration, release and deployment automation, service discovery and traceability.

## Key Elements of Internal Architecture Include:

### APIs

Microservice architecture structures an application as a set of services collaborating in order to handle requests. Since service instances are typically processes running on multiple machines, they must interact using IPC.

Services can use synchronous request/response-based communication mechanisms such as HTTP-based REST or gRPC. Alternatively, they can use asynchronous, message-based communication mechanisms such as AMQP or STOMP.

There are also a variety of different message formats. Services can use a human-readable, text-based formats such as JSON, or XML. Alternatively, they canuse a more efficient, binary format such as Avro, or Protocol Buffers.

### Versioning

As a defining requirement, microservices should be able to support multiple versions in the same environment, at the same time.

The development environment, therefore, requires implementing a feature-rich source control management system (SCM), and the deployment mechanism should be aware of the multiple versions of the code that are running and be able to quickly pull out a version if it's not working well.

Auto scaling and monitoring needs to be version-aware, as well.

### Containers

Containers are fast emerging as the standard way of packaging and running distributed applications. Teams can package their respective microservice modules into one or more containers, which can then be promoted through environments as atomic, immutable, units of code/configuration/runtime/system libraries/operating system/start-and-stop hooks. A container deployed locally will run the exact same way in a production environment.

Managing container environments is an external architecture concern.

### Software-defined infrastructure

Since each team needs to own its entire stack and not be dependent on any other team, operating in a cloud environment becomes a requirement.  A microservice's configuration could be packed into the container, or it can be externalized and pulled by the microservice as required. It's best to place the configuration inside the container so that the container itself runs exactly the same regardless of its environment.

As part of achieving disposable infrastructure, HTTP session state (login status, cart, pages visited, etc.) should be persisted to a third-party system, like a cache grid. None of it should be persisted to a container because of its ephemeral nature. Further, every microservice needs to exclusively own its data.

### Circuit Breakers

Circuit Breakers are designed for isolating failures. Calls from one microservice to another should always be routed through a circuit breaker such as Hystrix from Netflix.

A circuit breaker uses active, passive, or active plus passive monitoring to keep tabs on the health of the microservice being called. Active monitoring can probe the health of a remote microservice on a scheduled basis, whereas passive monitoring can monitor how requests to a particular microservice is performing. If a microservice is not responding, the circuit breaker will stop making calls to it. This is key to limiting the cascading nature of system failures.

### Distinctive Programming Model

Microservices have evolved from SOA with its own distinctive programming model. SOA applications typically use heavyweight technologies such as SOAP and other WS* standards along with a ESB, which is a 'smart pipe' containing business and message-processing logic to integrate the services. Applications built using the microservice architecture tend to use lightweight, open-source technologies, while communicating via 'dumb pipes' such as a message broker or lightweight protocols such as REST or gRPC.

Data is key to the microservices architecture. SOA applications typically have a global data model and share databases. In the microservices universe, each service has its own database. Moreover, each service usually implements its own domain model. Each microservice team should have some freedom in selecting the language/runtime for their respectiveimplementation. A team writing a microservice for inventory might want to use Node.js because of its ability to gracefully increment and decrement a number without locking.

## Key Elements of External Architecture Include

### Container Orchestration

Container orchestration is a PaaS that is increasingly being adopted in conjunction with microservices. The container itself becomes the artifact that the container orchestration system manages, rendering it extremely flexible. Container orchestration systems are less opinionated than traditional PaaS and are more flexible.

### Software-Defined Networking, Autoscaling, Storage, Security

While container orchestration provides direct support for implementing microservices architecture, related capabilities that require to be addressed from a cloud platform include software-defined networking, autoscaling, storage and security (layered above networking, including identification, authentication and authorization)

### Release Management

Every team should release code using the same process. The artifacts should be containers that, like microservices, do only one thing. For example, your application should be in one container and your datastore should be in another. Container orchestration systems are all built around the assumption of a container running just one thing.

Key functions that need to be orchestrated include:

- Build container images, inclusive of code/ configuration/runtime/system libraries/ operating system/start-and-stop hooks
- Define success/failure criteria
- Define rollout strategy
- Following the deployment, the container orchestration system needs to update load balancers with the new routes, cutover traffic, and then run the container's start/stop hooks

### Service Registry

In a distributed environment, with container orchestration in place, service discovery needs to be addressed as a core architectural requirement -

- Containers might live for only a few seconds, minutes, or hours
- Containers often expose nonstandard ports. For example, you might not always be able to hit HTTP over port 80.
- A microservice is likely to have many major and minor versions live at the same time, requiring the client to state a version in the request
- There are dozens, hundreds or even thousands of different microservices

Service discovery can adopt two basic approaches: client-side and server-side.

The client queries a standalone service registry to ask for the path to a fully qualified endpoint. The query could be a formal JSON document stating version and other quality-of-service preferences, depending on the sophistication of the service registry. The major drawback of this approach is that the client must "learn" how to query each microservice, which is a form of coupling. Another issue is that the client will need to re-query for an endpoint if the one it's communicating with directly fails.

The server-side method is often preferable due to its simplicity and extensive use today. This approach uses a load balancer. When the container orchestration places a container, it registers the endpoint with the load balancer. The client can make some requests about the endpoint by specifying HTTP headers or similar. Unlike client-side load balancing, the client doesn't need to know how to query for an endpoint. It is simpler as the load balancer just picks the best endpoint.

### Load Balancing

With server-side service registry, load balancing becomes critical. Every time a container is placed, the load balancer needs to be updated with the IP, port and other metadata of the newlycreated endpoint.

There are two levels of load balancing within a container orchestration system: local and remote.

Local load balancing is load balancing within a single host. By intelligently aggregating containers on the same host, we can minimize network traffic. Networking can also be simplified because it's over localhost. Latency is zero, which helps improve performance.

In addition to local load balancing, remote load balancing would require to be provisioned. It's a standalone load balancer that is used to route traffic across multiple hosts. API load balancers are more purpose built, supporting identification, authentication, and authorization-related security concerns. They can cache entire responses where appropriate and better support versioning.

### API Gateway

When a webpage or a screen on a mobile device requires to retrieve data from multiple microservices, an API gateway (intermediary) makes concurrent requests to each service required to build a single response. The client gets back one tailored representation of the data. As microservices are meant to be omnichannel, the API gateway typically uses intelligence to optimize the queries it makes to each service.

The API gateway plays the role of a façade, provides the REST APIs that are used by the web and mobile applications. The gateway may play the role of an API composer. This option makes sense if the query operation is part of the application's external API. Instead of simply routing a request to another service, the API gateway implements the API composition logic. This approach enables a client, such as a mobile device, that is running outside of the firewall to efficiently retrieve data from numerous services with a single API call.

This potentially creates coupling because the layer above now needs to know more details about your service. So, there is a trade-off that requires to be evaluated.

### Eventing

Event sourcing represents an effective way to implement business logic in an event-driven microservices environment. We capture domain events, which communicate changes to data between services. Event sourcing is a different way of structuring the business logic and persisting aggregates. It persists an aggregate as a sequence of events. Each event represents a state change of the aggregate. An application recreates the current state of an aggregate by replaying the events.

Clients, API gateways, and other microservices might synchronously call into a microservice and ask for the current inventory level for a product, or for a customer's order history, for example.

But behind the synchronous API calls, there's an entire ecosystem of data that's being passed around asynchronously. Every time a customer's order is updated in the order microservice, a copy should go out as an event. Refunds should be thrown up as events. Eventing is far better than synchronous API calls because it can buffer messages until the microservice is able to process them. It prevents outages by reducing tight coupling.

In addition to actual data belonging to microservices, system events are also represented as microservices. Log messages are streamed out as events—the container orchestration system should send out an event every time a container is launched; every time a health- check fails, an event should go out. Everything is an event in a microservices ecosystem.

# Toward an API-driven Implementation Framework

In this paper, we have highlighted the key architectural tenets that underlie the move to cloud-native applications and platforms leveraging emerging microservices architectural patterns.

We will follow up with implementation frameworks and best practices that have emerged in our experiences transitioning legacy commerce stacks to microservices-driven distributed cloud architectures.

As is evident from our earlier discussions, APIs quickly emerge as the key functional component for scaling and sharing service interfaces across business and customer domains. However, Business APIs require a management framework for design, documentation, versioning, workflow, analytics, security and connector integration.

We, at ITC Infotech,have created, JANA, an API lifecycle management platform for accelerating the transition to unified commerce. JANA comes pre-packaged with key microservices in the areas of payment, order management, search as well as a growing list of connectors with existing commerce platforms and other third-partyservices, capabilities or products that can be quickly pluggedin, providing extensibilityand enabling quick, iterative experimentation.

We will discuss JANA's architectural framework in an upcoming paper.

**About ITC Infotech**

ITC Infotech is a specialized global full service technology solutions provider, led by Business and Technology Consulting. ITC Infotech's Digitaligence@work infuses technology with domain, data, design, and differentiated delivery to significantly enhance experience and efficiency, enabling its clients differentiate and disrupt the business.

The company caters to enterprises in Supply Chain based industries (CPG, Retail, Manufacturing, Hi-Tech) and Services (Banking, Financial Services and Insurance, Airline, Hospitality) through a combination of traditional and newer business models, as a long-term sustainable partner.

ITC Infotech is a fully-owned subsidiary of ITC Ltd, one of India's foremost private sector companies and a diversified conglomerate.

www.itcinfotech.com | Email: contact.us@itcinfotech.com