Best Practices for Microservices

Implementing a foundation for continuous innovation

Executive Summary

Today's business environment is extraordinarily competitive. No company, no matter its size or what industry it is in is safe from disruption. It is easier than ever for new entrants to come into a market, turning entire industries upside down. Unless organizations can nimbly innovate at the speed of their competition, they will be left behind, and large organizations with calcified processes and structures will be hit the hardest.

It is possible for any organization to harness the opportunities that digital transformation provides; it simply requires a more agile enterprise. This type of business employs a framework of smaller, hyper-focused teams rapidly innovating on defined units of business value, working in concert to deliver something much larger. We call this becoming the composable enterprise. Making an organization composable requires changes to how IT supports the business. It means creating scale through reusable services and enabling self-service consumption of those services.

The ability of your business to change quickly, innovate easily, and meet competition wherever it arises is a strategic necessity today. This will allow you to thrive in a market which constantly changes, and create new customer experiences in new contexts using new technologies. Your business can shape innovative customer experiences only if your IT team provides digital assets in the form of core business capabilities that bring real value to your business in multiple contexts. This is how IT can be an enabler of the strategic goals of your business.

Organizations that have successfully laid a foundation for continuous innovation and agility have adopted microservice architectures to respond rapidly to the demands of the business.

Organizations that have successfully laid a foundation for continuous innovation and agility have adopted microservice architectures to respond rapidly to the never-ending demands of the business.

Why Microservices?

Microservices are the evolution of best-practice architectural principles that shape the delivery of solutions to the business in the form of services. All businesses, no matter what industry they are in, must strive to deliver the ideal customer experience, as customers are more demanding than ever and will abandon a business that is too slow to respond. IT must deliver solutions that can be adapted to deliver a holistic and uniform experience to the customer across all the business channels. To achieve this the architecture should identify and define digital assets which align to core business capabilities. They offer the potential to break down the coupling between business channels and the backend systems of record that cater to them. A microservice that encapsulates a core business capability and adheres to the design principles and goals outlined below should be considered a true digital asset. It can bring value to the business because it can be adapted for use in multiple contexts. The contexts for use of the service are the business processes and

transactions, and the channels through which your customer, employee, or partner interacts with your business.

Benefits to the Business

A microservice architecture ***aligns with the business*** in such a way that changes to your business can be dealt with in an agile fashion. Business processes and transactions are automated with the composition of microservices. When processes are changed or when new ones are introduced, IT can respond by re-wiring services into new compositions.

The ease and speed with which your company can change will determine your ability to react to trends in your industry and maintain competitive advantage. With solution logic in the form of composable services, IT can run at the pace of the business and match business changes with an ***agile response*** in the delivery of solution logic. Innovation is the face of this agility. It may take the form of new channels of business (like Google creating new revenue streams by productizing their APIs), new digital engagements with customers (like Spotify engaging with their customers through Uber), entirely new products and services which may demand entirely new business processes, or the simple modification of existing business processes. All of this speaks of change. Companies need to be able to alter direction based on market forces. IT must be able to facilitate change by composing existing digital assets into new business capabilities.

Your enterprise can deliver solution logic in a decentralized manner with the standardization of microservice contracts in the form of APIs. Multiple teams from different domains can implement services with their own choice of technology but yet remain aligned with the business in their purpose. This represents the evolution of IT's role from provider to partner resulting in the ***enablement of teams*** from the lines of business to adapt the core capabilities built by the central team to their own particular needs.

The ease and speed with which your company can change will determine your ability to react to trends in your industry and maintain competitive advantage.

Microservices are natively able to communicate with each other because of industry-wide adoption of standards like HTTP and JSON. In other words, they are ***intrinsically interoperable***. They facilitate an exchange of information independent of how they have been implemented because their interface is defined according to standards which already exist at the level of the industry and which are also defined by the teams in your organization.

Your business can ***pick and choose best-of-breed vendor products*** and platforms because of the interoperability that standardized interfaces bring. For the same reason, your teams can also choose the best technologies to implement the microservices with a polyglot approach to development.

What are microservices?

Traditionally, enterprises have delivered software applications in siloes. These arose from the isolated demands of individual departments. Software was developed or purchased in attention to these limited scopes. Customer facing business processes on the other hand typically span multiple departments. The lack of alignment between these two realities led to duplicate efforts and missing or inaccurate information in each solution. The latter was a major force behind enterprise integration.
The service concept evolved from attempts to respect the business process as focal point of solution requirements. The need to modify existing processes or invent entirely new ones should be met with an agile response from IT to adapt to the change. Rather than build or purchase “applications” in the traditional sense, a service approach stipulates the creation of services as the building block of solution logic which are composable to address the automation requirements of business processes.

Consumption Modes

Microservices are most often directly invoked by HTTP REST API calls. Standards like RAML (Restful API Modelling Language) allow for the formal definition of REST APIs. RAML can define every resource and operation exposed by the microservice. It’s an industry standard which has gained popularity as a lightweight approach to microservice interface definition and publication.

In those cases where microservices collaborate for the realization of a complex business transaction or process, an event-driven approach is also adopted. The idea is for the microservice to subscribe to business domain events which are published to a message broker. Standards like JMS and AMQP are dominant in major broker technologies in the industry. The involvement of a message broker in the microservice invocation adds reliability to the consumption model. Even if the microservice subscribed to a particular queue, topic or exchange were down when the event was published, the messaging paradigm guarantees that the microservice will receive the message once it comes back online again.

The composition of microservices is realized with a mix of direct calls through HTTP and indirect calls through a message broker.

API-led Connectivity

With the conviction that building an adaptable business capability is much better than building a tactical point to point integration, IT should strive to deliver these assets which are accessible through API invocation and domain event subscription and which can deliver value to the business in multiple contexts.

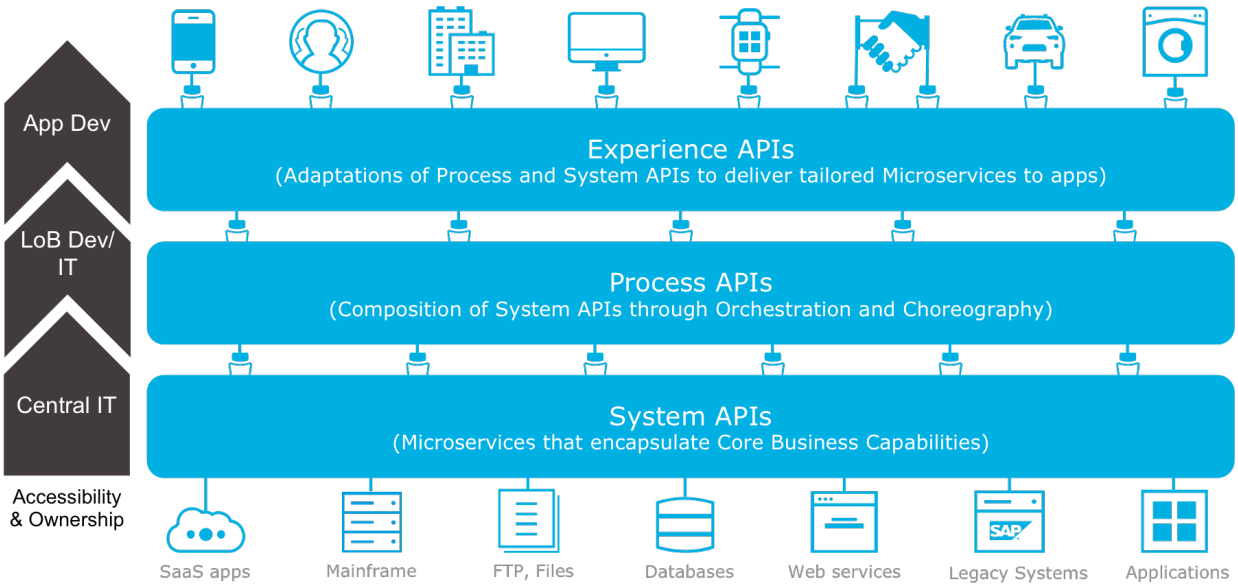


Figure 1. Microservices classification

The asset is primarily an encapsulation of a business entity capability, like Customer, Order, or Invoice. These system APIs or system-level microservices are in line with the concept of an autonomous service which has been designed with enough abstraction to hide the underlying systems of record. None of these system details are leaked through the API. The responsibility of the API is discrete and agnostic to any particular business process.

System APIs are composable with other APIs to form aggregate process APIs. The composition of system APIs can take the form of explicit API orchestration (direct calls) or through the more reliable API choreography by which they are driven by business events relevant to the context of the composition (order fulfillment, for example).

The API Gateway

Both process and system APIs should be tailored and exposed to suit the needs of each business channel and digital touchpoint. The adaptation is shaped by the desired digital experience and is what we call the experience API. Sometimes the adaptation of the API is technically motivated: a particular security mechanism might be needed on one channel; the types of channel may differ greatly as do mobile, web and devices; a composition of multiple APIs might be needed according to the backend for frontend pattern. In other contexts business differences must be catered to with adaptations that consider the special requirements of groups of users like employees, customers, and partners.

In all these cases the API Gateway pattern is a good approach because it is where API compositions and proxies are deployed. API Management facilitates the administrative application of recurring logic, like security, rate limiting, auditing and data filtering to the experience APIs on the gateway. The use of API management to apply policies that encapsulate the tailored logic makes the adaptation of system and process APIs relatively quick and easy.

The "Micro" of Microservices

Scope of Responsibility

The most obvious candidate microservice is the business entity easily identifiable with a glance at a business process or transaction. A *customer* is an example of such an entity. It is a business entity that may be relevant in a number of contexts both at the process layer and the experience layer. The responsibility is not limited to mere data carrying. A microservice should not be reduced to a simple CRUD service. Each entity encapsulates within itself all responsibilities relevant to the business domain for which it was designed. All of this goes hand in hand with a deliberate limitation of the scope of responsibility. "Discrete" is the best common sense interpretation of "micro" with respect to scope of responsibility. This helps in guaranteeing that the system level microservice doesn't take on any responsibility that more appropriately pertains to a business process level microservice. The system level microservices are agnostic to any particular business process and hence can be used in more than one composition.

The scope of a business process must also be considered. Some sub-domains have their own local business processes (*shipping*, for example). A system level microservice can be adapted for use in multiple processes within the *shipping* domain. Whenever its capability is needed outside the domain, it can be adapted for this with an experience level microservice. There are of course business processes, typically customer facing, which traverse multiple domains. The rule here is to compose multiple microservices adapted for this use in experience APIs.

Team Organization and Responsibilities

Teams should be small enough to work locally together and focus entirely on a single sub-domain of the business, and include domain experts so that the language of that sub-domain is modelled in the solution.

The ownership of the microservice includes everything from design to deployment and management. The API is considered a business product, an asset to be delivered to the business. There is no handoff to other teams to manage the running instance. The whole lifecycle belongs to the team who develop it.

Scope of Effort

Microservices that address the needs pertaining to a specific sub-domain will recognize that certain business concepts are perceived in a way that is particular to the sub-domain. No attempt is made to model a universal solution unless the entity is naturally shared across the entire organization.

Ease of deployment

The cost of delivery to production is greatly reduced by the combination of small teams with complete ownership, the discrete responsibility of the microservice, and the infrastructure which facilitates continuous delivery.

Fundamental Principles of Microservice Design

Microservice capabilities are expressed formally with ***business-oriented*** APIs. They encapsulate a core business capability and as such are assets to the business. The implementation of the service, which may involve integrations with systems of record, is completely hidden as the interface is defined purely in business terms.

The positioning of services as valuable assets to the business implicitly promotes them as ***adaptable*** for use in multiple contexts. The same service can deliver its capabilities to more than one business process or over different business channels or digital touchpoints.

Dependencies between services and their consumers are minimized with the application of the principle of ***loose coupling***. By standardizing on contracts as expressed through business oriented APIs, consumers are not impacted by changes in the implementation of the service. This allows the service owners to change the implementation and switch out or modify the systems of record or even service compositions which may lie behind the interface and replace them without any downstream impact.

Identification of Candidate Microservices

Identifying the microservices that you need to build and the scope of their responsibility can be helped by considering the types of information that are exchanged in the transactions they cater to. In a healthcare setting, *patient*, *encounter* and *claim* are examples. In an e-commerce setting there are: *order*, *item*, *discount*, or *customer*. In a banking solution, you might have *transfer*, *account*, *payee*. The responsibility ought to be lean and focused. The system level microservices are not an abstraction over an entire back-end system, but only that part of the system or systems responsible for the storage of the business entity. Business transactions and processes are often the focus of IT efforts because it is the business process that effectively defines the business. This may be *order fulfillment* in the e-commerce sector. A visit that a patient makes to a hospital could be driven by the *visit administration* microservice. This microservice is implemented with the composition of microservices in the system layer.

Autonomy is a measure of control that the implementation of the service has over its runtime environment and database schema. This enhances the performance and reliability of the service and gives

consumers more guarantees about the quality of service they can expect from it. Coupled with statelessness, autonomy also contributes to the overall availability and scalability of the service. Each service is necessarily ***fault tolerant*** so that failures on the side of its collaborating services will have minimal impact on its own SLA. Services, by virtue of being independent of each other, have the opportunity to cut off communication to a failed service. This technique, called a "circuit breaker" and inspired by the electrical component of the same name, stops individual service failures from propagating through the larger, distributed system.

All of these design principles contribute to the principle of ***composability*** which allows the service to deliver value to the business in different contexts. Its composition together with other services to form a new service aggregate is effectively the new form of application development.

The aim of ***discoverability*** is to communicate to all interested parties a clear understanding of the business purpose and technical interface of the microservice. Thus, the service must be published in a way that ensures that developers of client software have everything they need to easily consume it.

Reuse

Reuse continues to be a principle of microservice design. However, the scope of reuse has been reduced to specific domains within the business. The effort of designing for this reuse, which in the early days of SOA included wasted efforts in designing enterprise-wide canonical models, was fruitless because it was too ambitious. However, it must be noted that the canonical model in its restricted scope can be of benefit. In line with the reuse it facilitates, its scope has been reduced. With the 'merit based reuse' approach, an emerging model is preferred over a predetermined one. Teams can agree on communication models for deciding how microservices must be adapted for use outside the contexts in which they were designed. A collaboration hub like Anypoint Exchange encourages Merit based reuse with reviews, ratings, etc. If an existing microservice API does not suit your domain or 'business group', you might be better off building another microservice that does it.

Business Domain orientation of Microservice Architecture

It is important to approach service design for a particular domain and NOT insist on doing so for every aspect of the business. The failure of enterprise-wide canonical modeling exercises of the last decade are evidence of this. The reality of the business is that business entities can be perceived in deeply different ways across the business units or sub-domains.

No attempt should be made to design enterprise-wide services if each domain has a very different perception of the same concept. An example business entity with very different perceptions is the *Customer* in the Customer Care, Orders, Invoicing and Shipping domains. In figure 2 you can see how each domain has its own microservices which encapsulate core business capabilities for that particular domain. This can result in apparent duplication of microservices as is the case for the Customer API.

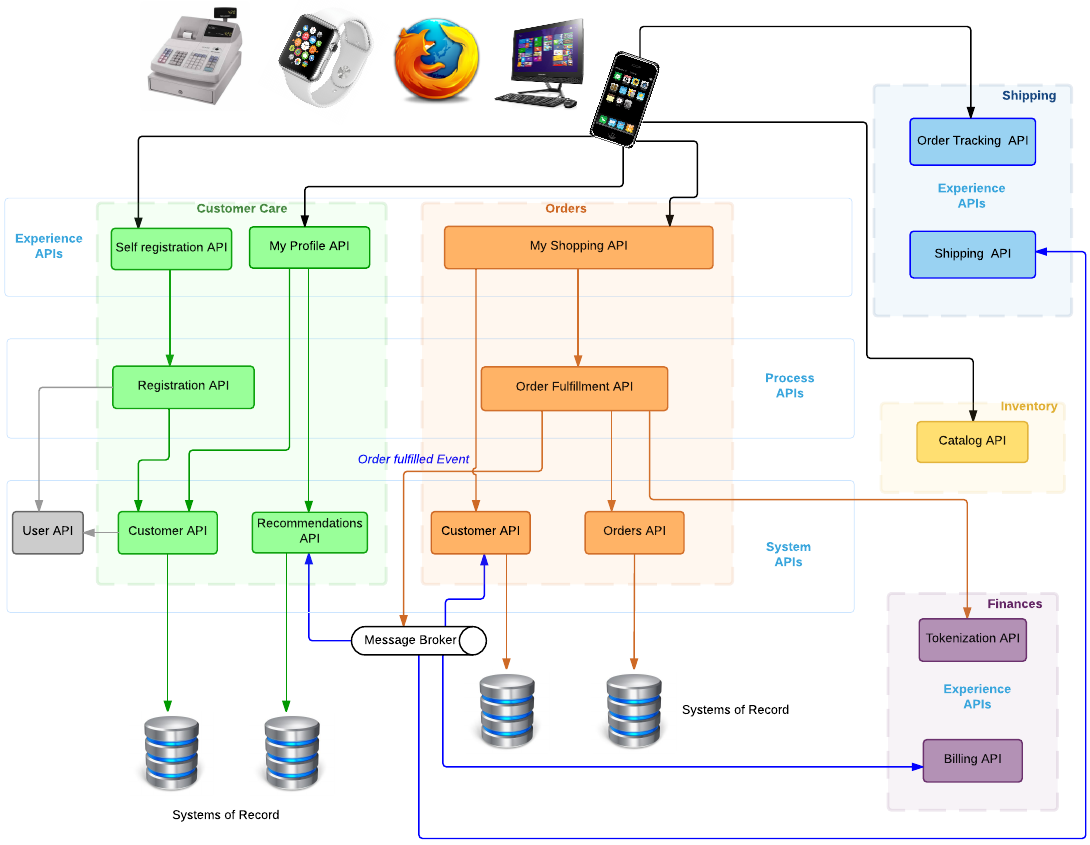


Figure 2. Microservices designed for particular Business Domains

As microservices communicate with each other, especially those designed for different domains, there may be a need to negotiate a contract that suits the needs of the consuming software. This particular adaptation will be manifest as an experience API tailored specifically to that client. It could also take the form of a domain event published to a queue.

Microservices and The Monolith

Microservices are a fundamental shift in how IT approaches software development. Traditional software development processes (waterfall, agile, etc) usually result in relatively large teams working on a single, monolithic deployment artifact. Project managers, developers and operational staff can reach varying degrees of success with these models, releasing application candidates that can be verified by the business, particularly as they gain experience using a particular software and deployment stack. There are, however, some lurking issues with the traditional approaches:

- Monolithic applications can evolve into a "big ball of mud"; a situation where no single developer (or group of developers) understands the entirety of the application. This problem is exacerbated as senior developers roll off a project and are replaced by junior or offshore resources for maintenance.
- Limited re-use is realized across monolithic applications. Monolithic applications, by definition, hide their internals. While some re-usability might be realized by API's at the "edge" of the monolith, chances for re-use of internal components is limited. When reuse is achieved it's usually through shared libraries which foster tight coupling and are limited to the development platform used to implement the monolith.
- Scaling monolithic applications can be challenging. Identifying and tuning specific aspects of application functionality for performance in isolation of other aspects is usually impossible since all functionality is bundled in a single deployment artifact. Scaling monolithic applications is usually not in "real time."
- Operational agility is rarely achieved in the repeated deployment of monolithic application artifacts. While operational automation can alleviate most of the manual pain in deploying these applications, for instance by automating the provisioning of VM's or automatically configuring network devices, there's still a point where developers are blocked by operations staff waiting for these activities to occur. In the worst cases, where automation is not in place, developers and operators are under great stress every time a deployment fails or a production issue is hit.
- By definition monolithic applications are implemented using a single development stack (ie, JEE or .NET.) In addition to limiting reuse for application not implemented on the stack it also robs the opportunity to use "the right tool for the job." For instance, implementing a small part of a JEE application with Golang would be difficult in a monolithic application.

Avoiding the Distributed Monolith

The danger that new architectural trends pose is that they are perceived as a silver bullet for IT's problems and should be used as the "latest best thing" without due consideration for all pre-requisites regarding IT operating model, infrastructure and developer skillsets.

A microservices strategy should take a careful, measured approach as follows to reap maximum benefit: we strongly recommend designing and building microservices that encapsulate capabilities for particular business domains. The risk of not doing this is that you will end up building a monolithic suite of microservices. In other words: a distributed monolith with all the downfalls of the monolith, the added complexity of distribution, and a reduction in the overall return on your investment.

We also recommend that you establish the strict discipline of continuous delivery and have the necessary tooling for the automation of the release pipeline. A lack of Devops-style team coordination and automation will mean that your microservices initiative will bring more pain than benefits.

A microservice architecture, in concert with cloud deployment technologies, API management, and integration technologies, provide an alternate approach to software development which avoids the delivery

of monolithic applications. The monolith is instead "broken up" into a set of independent services that are developed, deployed and maintained separately. This has the following advantages:

- Services are encouraged to be small, ideally built by a handful of developers that can be fed by "2 pizza boxes." This means that a small group, or perhaps a single developer, can understand the entirety of a single microservice.
- If microservices expose their interfaces with a standard protocol, such as a REST-ful API or an AMQP exchange, they can be consumed and re-used by other services and applications without direct coupling through language bindings or shared libraries. Service registries can facilitate the discovery of these services by other groups.
- Services exist as independent deployment artifacts and can be scaled independently of other services.
- Developing services discretely allows developers to use the appropriate development framework for the task at hand. Services that compose other services into a composite API, for example, might be quicker to implement with MuleSoft than .NET or JEE.

The tradeoff of this flexibility is complexity. Managing a multitude of distributed services at scale is difficult:

- Project teams need to easily discover services as potential reuse candidates. These services should provide documentation, test consoles, etc so re-using is significantly easier than building from scratch.
- Interdependencies between services need to be closely monitored. Downtime of services, service outages, service upgrades, etc can all have cascading downstream effects and such impact should be proactively analyzed.

The SDLC lifecycle of services should be highly automated. This requires technology, such as deployment automation technologies and CI frameworks, but more importantly discipline from developer and operations teams.

Microservice Implementation Patterns

Command Query Responsibility Segregation

The classification of microservices into system, process, and experience types can be further subdivided into consideration of scalability requirements. Most requests to microservices are to retrieve information for presentation purposes. A lesser number of requests are to realize a state changing business function, like a customer modifying their personal profile, or submitting an order. We distinguish these types of requests as queries for the retrieval of information and commands for the state changing business function.
Some high traffic requirements may require a deliberate separation of the deployment so that the one business capability is split into two microservices: one for commands and the other for queries in line with the emerging pattern Command Query Responsibility Segregation (CQRS).

Event Sourcing

The autonomy principle of microservice design stipulates each microservice having its own data store. Database sharing is avoided. This creates a problem when we consider our approach to the automation of a business transaction. We recommended business process type microservices whose responsibility it is to compose system type microservices through orchestration and choreography. Naturally, the information exchange for any business transaction is related, but each system microservice executes its part in the collaboration independent from the rest. The end-state of the whole composition must leave all data in a consistent state.

The industry is moving away from distributed transactions to solve this problem. Hence, one can see that each microservice having its own data store to represent what ultimately is the same information at a higher level can result in an inconsistency between them at any moment in time. This reality is especially prevalent with a domain event driven approach in which the microservices collaborating in a choreography work asynchronously driven by domain events published to a message broker. Eventual consistency is the key here. When every microservice has completed its work, then the whole system is in a consistent state.

This leads to an emerging pattern in which changes to state are stored as journaled business events. The current state is known not by retrieving data from a store but by navigating the history of business events and calculating it on the fly.

Continuous Delivery of Composed Applications

Getting teams to continuously release software aligns with agile development principles. Continuous delivery of software lets business stakeholders verify, in real time, that an application is meeting the ultimate business objective. Continuous delivery also means, in terms of composed microservice applications, continuous integration. In applications that are composed of many services it is critical to ensure that the composition actually works when the software is built.
Having short release cycles, fast feedback on build failures and automated deployment facilities are critical in implementing continuous delivery.

Self-service Consumption of Microservices

Every corner of an enterprise needs technology to build new applications for their specific function or customer. IT needs to transform from its traditional function as the sole technology provider to become an adaptive, responsive and nimble organization that can keep up with the pace of the digital era as well as embrace the opportunities provided by a change driven environment. This transformation can occur only if IT transforms itself into an strategic business enabler rather than a centralized technology function.
Being an enabler means that IT has to decentralize and democratize application development and data access to the different Lines of Business (LoBs) and functional business partners. This way, IT can concentrate on a partnership with the business - i.e. providing a set of strategic and consistent assets and technology.

Service proliferation, however, is a trade off incurred by such an approach. Managing these services at scale raises a number of challenges:

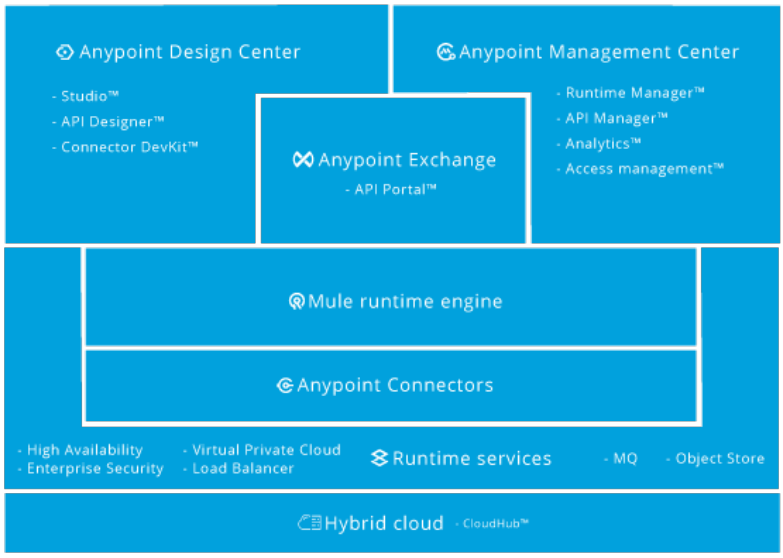
- Service Discovery and Documentation
- Fault tolerance
- Quality of Service
- Security
- Request traceability
- Failure triage

It is imperative that you can easily manage your microservices in a way that facilitates self-service access to them across all the lines of business in your enterprise. API Management represents the evolution of service governance that allows you to do the following:

- Publish your APIs so that developers of consuming software have everything they need to self-serve their needs and understand clearly the purpose, scope and interface of your microservice.
- Adapt your APIs through injectable Policies of logic covering security, quality-of-service, auditing, dynamic data filtering, etc.
- Watch your APIs so that you can strategize scalability according to traffic levels and take a temperature gauge on the impact of your assets.
- Tailor your APIs to the specific needs of different lines of business so that API management becomes a decentralized or federated exercise in collaboration between LOBs and central IT.

Microservices on Anypoint Platform

Anypoint Platform solves the most challenging connectivity problems. It's a unified, highly productive, hybrid integration platform that creates a seamless application network of apps, data and devices with API-led connectivity.
Unlike alternatives, Anypoint Platform can be accessed as a cloud solution or deployed on-premises allowing developers to rapidly connect, orchestrate and enable any internal or external application. Anypoint Platform lifts the weight of custom code and delivers the speed and agility to unlock the potential of this connected era.



End-to-End Microservice Lifecycle

MuleSoft takes a holistic view of microservices. Unlike traditional apps, the ideal microservice development starts with a top-down API-first approach. Which means there are additional steps compared to traditional Software Development Lifecycles (SDLC).
For instance, the design aspect is usually an iterative process that includes:

1. Modelling your API using standard specs like RAML.
2. You typically want to simulate your spec with a mock API endpoint with which you can Solicit feedback from your API consumers.
3. You also want to validate with real code so you can touch and feel the API and provide feedback.
4. Once the API design is ratified, you would build it using your favorite language, with code that includes business logic and connectivity to appropriate back-end systems.
5. Then create test scripts if following the recommended Test Driven Design approach.
6. Once the microservice is deployed, you typically want to publish the documentation for your API using a portal that becomes your engagement tier with users of your microservice.
7. Finally, when operationalized, you want to manage the microservice runtime, as well as its APIs (for instance, apply security and throttling policies) and get usage analytics for your microservice. Analytics could be used for management/monitoring purposes or for metering and chargeback.

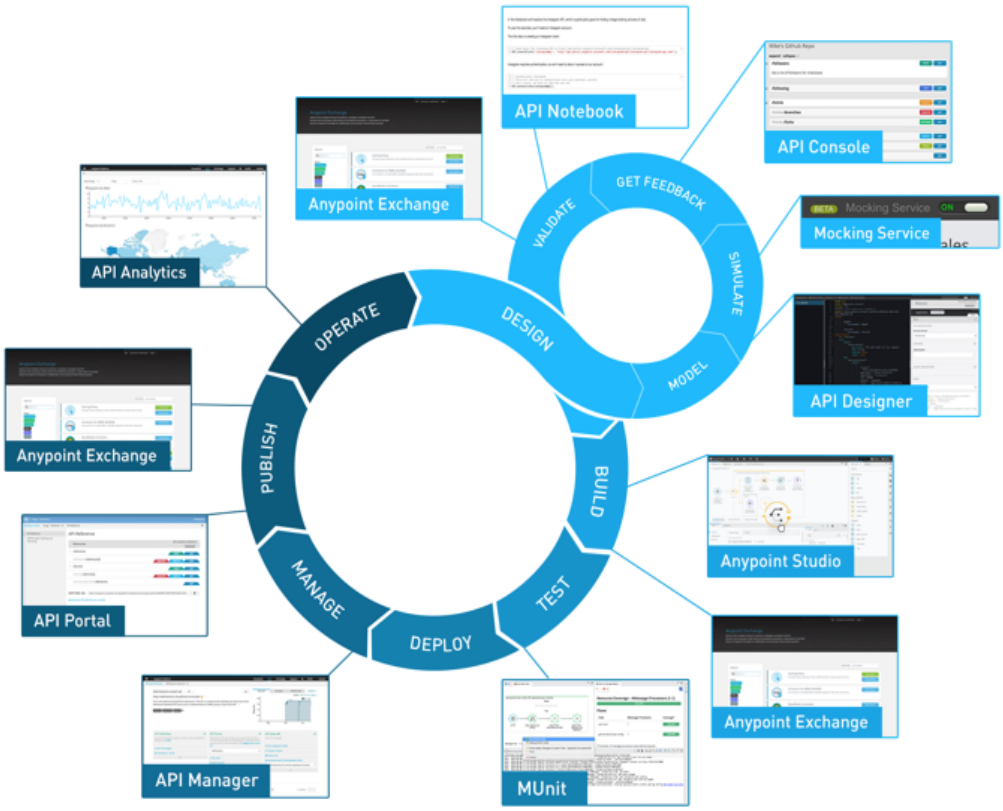
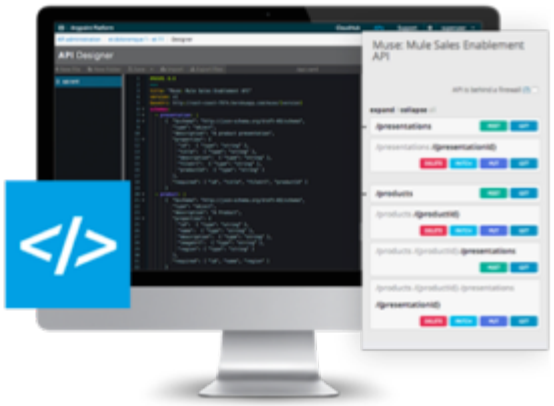


Figure 3. End to end lifecycle of microservices

Now let's go into each phase of the ideal microservice lifecycle in detail, and see how it can be accomplished with Anypoint Platform.

Design and Implementation with Anypoint Design Center

The design of a microservice should begin with its API definition. The API may be REST based or event-driven in line with the two modes of consumption (see Consumption Modes).



API Designer allows you to define a REST API using RAML. RAML is a standard which has achieved rapid adoption as the light-weight language of choice to define APIs. As you define the resources and operations for the API in RAML, API Designer auto-generates a console which centralizes documentation and testability. It also auto-generates a mocking service, deployed to CloudHub. This affords you the luxury of allowing the team responsible for the development of consuming software to write their code against the mocked implementation in parallel with the team who must implement the actual microservice. The team responsible for building the consuming software can showcase their work before the microservice is even developed.

If your team chooses to implement the microservice as a Mule application, then Anypoint Studio will allow them to do so rapidly with its scaffolding of RAML based APIs as a set of flows which implement all the operations. This is done with a graphical drag and drop of message processors. Anypoint Studio also allows the developer to build a true unit test of the application using MUnit, the unit testing framework for Anypoint Platform, with the same graphical approach. Anypoint Connectors encapsulate connectivity to many public APIs, like SalesForce and SAP. The suite of connectors is extensible with Anypoint DevKit, which allows you to turn a simple Java POJO into a reusable connector available for use within Anypoint Studio's palette.

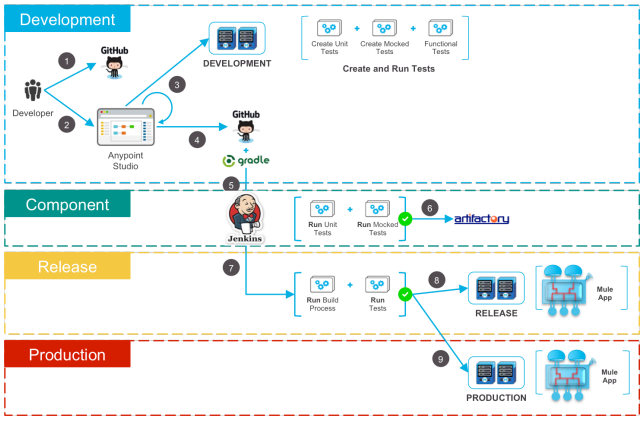
Continuous Delivery with Anypoint Studio, Maven and Docker

A Mule application can be built in Anypoint Studio with Maven or Gradle and committed to an SCM like Github. From there a CI/CD framework like Jenkins can pull down the latest version, build it, execute all relevant tests and deploy it to the next environment in the build pipeline.

Differences in physical endpoints like databases that correspond to each environment are addressed by exploiting external properties files. Thus, the same Mule application progresses without change from dev through test and into production.

Traditional, monolithic applications are typically deployed using the operation convention for the given development platform. Monolithic Java applications, for instance, are usually deployed to multi-tenant application servers like JBoss AS or Tomcat.

IaaS frameworks, such as Chef and Puppet and virtualization technology such as VMWare or Zen can greatly accelerate the deployment of monolithic application stacks. Recent advances in container technology, particularly frameworks such as Docker and Rocket, provide the foundation for almost immediate deployment and undeployment of applications without necessarily provisioning new physical or virtual hardware. Numerous benefits arise from this deployment paradigm, including increased compute density within a single operating system, trivial horizontal and, in some cases vertical autoscaling as well as container packaging and distribution.



These advantages make containers the ideal choice for microservice distribution and deployment. Container technologies follow the microservice philosophy of encapsulating any single piece of functionality and doing it well. They also give microservices the ability to expand elastically to respond to dynamic request demand.

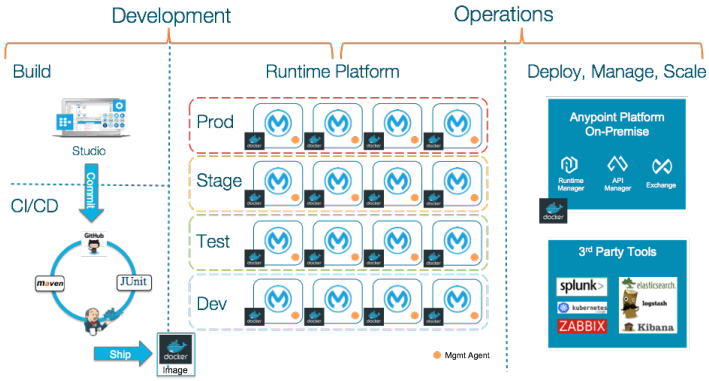
There is no such thing as a free lunch, however, and managing a container ecosystem comes with its own challenges. For instance, containers running on the same host will be bound to ephemeral ports to avoid conflicts, container failure must be addressed separately, containers need to integrate with front end networking infrastructure such as load-balancers, firewalls or perhaps software-defined networking stacks, etc. Platform-as-a-service technologies, such as Pivotal Cloud Foundry and Mesosphere DCOS, are emerging to address these needs.

Hexagonal Architecture of a Mule Application

A mule application is shaped by the ports and adapters (hexagonal) architecture. Thus a microservice implemented in a Mule application can be invoked over multiple transports (HTTP, JMS, AMQP, etc.), transform the various payloads that arrive over those transports and process the data in a way that is agnostic to any of the transports. Likewise, its outbound communication to datastores, other microservices, and systems of record can be over any standard protocol. In contrast to legacy heavyweight ESBs, which advocated centralization of all the "smarts", all logic related to connectivity, routing decisions, transformation, error handling and security, is contained within the Mule application. Thus, a microservices oriented runtime like Mule represents a lightweight option to host a microservice.

Deployment as a Microservice Container

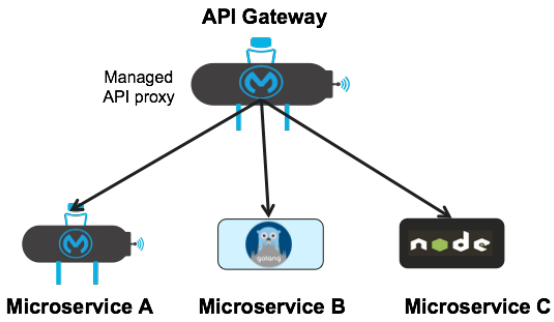
Anypoint Platform's runtime components support a variety of deployment mechanisms, ranging from traditional multi-tenant cluster based deployment to a Mule worker packaged and deployed as a container in CloudHub, MuleSoft's fully hosted and fully managed PaaS.



The Mule runtime can support a microservice architecture in an orthogonal manner using two complementary deployment approaches:

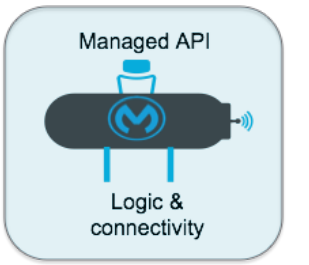
1. Mule as an API Gateway

When deployed in this manner the Mule runtime acts as an API Gateway to proxy HTTP traffic back and forth between microservices. This allows Mule to transparently apply cross-cutting policies, to enforce concerns like governance and security, to all API calls traversing a microservice architecture. It additionally allows Mule to asynchronously collect analytics about microservice traffic and consumption patterns, providing valuable insight back to the business.



2. Mule as a Microservice Container

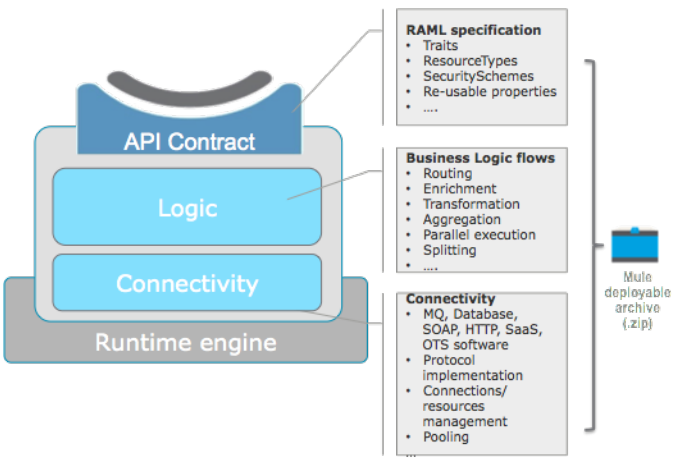
One of the benefits of microservice architecture is the freedom to "choose the right tool for the job" to implement a given service. For certain services, a development language like NodeJS or Java might be appropriate. Microservices that are focused on connectivity, orchestration or transformation however are usually easier to implement with an integration framework like Mule. Microservices implemented in Mule can present a managed API since it runs in the same runtime as the API management layer.



Microservice Container

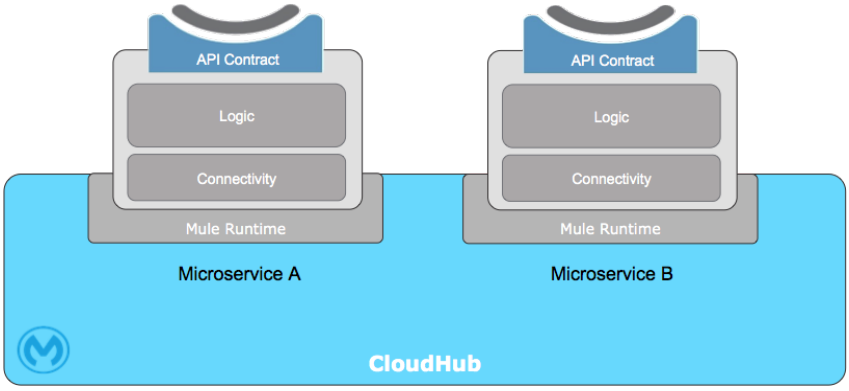
This provides a unified runtime for API management and integration unique to MuleSoft. The Anypoint Platform can be used for the composition and connectivity logic for your microservice, exposed as a managed API endpoint. It eliminates the need for a separate API gateway process by providing both capabilities on the same runtime. This simplifies the container creation scripts and the number of moving parts to maintain, which benefits large scale microservice deployments.
Both options can be configured on-premise or in MuleSoft's hosted CloudHub. Hosting your microservices on CloudHub simplifies the microservices DevOps complexity since MuleSoft automates most of the operations aspects and allows deployments in a self-serve, PaaS model.
However, if you choose to set up your own PaaS for microservices the Mule runtime has the following attributes that make it a perfect fit for on-premise containerization:

- The Mule runtime is distributed as a standalone zip file - a JVM is the only dependency
- It runs as a single low-resource process. For instance, it can even be embedded in a resource constrained device like a Raspberry Pi
- The Mule runtime doesn't require any external, persistent storage to share state. This means external resources like databases or messaging systems aren't required by default.



- The Mule application can be layered on top of a Mule runtime container (e.g. Docker) image

Deployment on CloudHub



We saw how a containerized on-premise deployment can be achieved with MuleSoft. One disadvantage of this approach is that it puts the onus of maintaining the containerization and PaaS on the customer.

If you chose to host it on MuleSoft's PaaS (CloudHub) instead, the self serve aspect is handled by Anypoint Platform and the runtime is a fully hosted and managed service. Each microservice has it's own autonomous runtime. CloudHub is designed to be secure and fully scalable with built-in High Availability and Disaster Recovery. CloudHub also provides a single-click Global deployment of your microservice. Which guarantees your microservice runtime is compliant to local regulatory requirements, keeping traffic within geographical boundaries.
CloudHub drastically reduces barriers to adoption of a microservices architecture by avoiding the so called Microservices Premium required to set up and maintain your own containerization and PaaS framework. Which means, you can get started with microservices in minutes without having to worry about the infrastructure to support it.

Event-driven microservices with Anypoint MQ

Anypoint MQ is an enterprise-class cloud messaging service, fully integrated with Anypoint Platform and which can act as the means to invoke event-driven microservices. Process level microservices can publish domain events to queues. Those system level microservices which subscribe to the same queues will be invoked as soon as the event is published. In line with reliability requirements, even if they are down in the moment of publication, they will receive the message as soon as they come back up.
For scenarios where multiple microservices are interested in a particular business event, the process level microservice can publish the event to an exchange. Queues bound to the exchange will each receive a copy of the event and as described the microservices subscribed to them will be invoked.

Operational management with Anypoint Runtime Manager

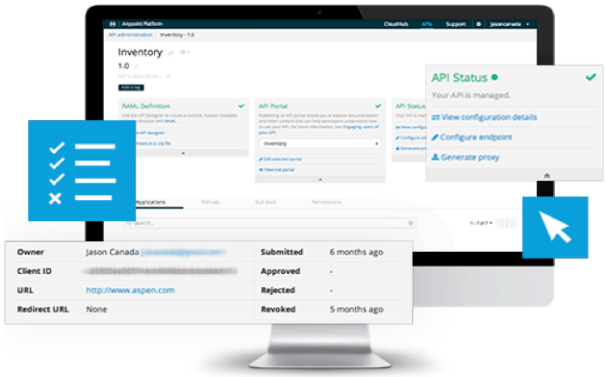
Anypoint Runtime Manager provides a single operational plane to manage both microservice applications built with Anypoint Platform as well as the Mule runtimes that host them. Anypoint Runtime Manager can manage your Mule runtimes on-premises and/or hosted on CloudHub. Anypoint Runtime Manager allows

you to start and stop these and cluster them. You can monitor memory and CPU usage and set alerts for when thresholds of their usage are exceeded.
Logs and business data can be viewed and analyzed from the console and can also be pushed out to Splunk, ELK or any DB (for analysis with tools like Tableau).

API management with Anypoint Platform's API solution

Anypoint Platform provides a number of features to manage microservices at scale in a large enterprise:

- API portals provide self-service documentation, test consoles, SDK client generation and programmable notebooks to allow developers to discover and learn how to consume the API for a microservice.
- Mule runtimes, deployed as API gateways, can proxy communication between microservices. This ensures policies, like security and throttling, are correctly applied across all microservices.
- For microservices built and deployed to CloudHub, you can leverage CloudHub Insight to get in-depth visibility into business transactions and events on your Mule applications deployed to CloudHub. Insight makes information searchable and helps you find and recover from any errors that occurred during processing and replay your transactions instantly if necessary.
  CloudHub Insight helps you answer questions about your integrated apps, such as:
    - What happened with a particular request or synchronization?
    - When did the request occur? How long did it take?
    - What was the result of a request?
    - If something went wrong during processing, at what point did the failure occur?



As the dependency graph between services grows, issues that were previously isolated, such as transient performance problems, can cascade across multiple services. API gateways can potentially act as a "circuit breaker" to quickly detect and isolate services from such failure.

Repeatedly implementing security and other cross-cutting concerns in microservices represents duplicated, potentially difficult, effort for developers. There is also the risk that developers will forget or incorrectly implement each concern. Consider the example of implementing security with OAuth 2.0. The API Management module of the Anypoint Platform includes an OAuth 2.0 Policy out-of-the-box. All you need to do is apply this policy to the microservice and you have OAuth 2.0 security. Leveraging shared policies ensures the correct security policy implementation is applied to all API's. If you have a cross-cutting policy (for instance, to selectively mask specified business data in a request), you can quickly and easily build a custom policy to reuse across other microservices, and non-microservices for that matter.

Publication and engagement with Anypoint Exchange

As digital assets with the potential to bring business capabilities to multiple contexts within or outside of your business domain, microservices ought to have their APIs published. This should be done in a way that minimizes the friction of informing and enabling developers of consuming software to understand everything they need to adapt and / or consume the microservice.

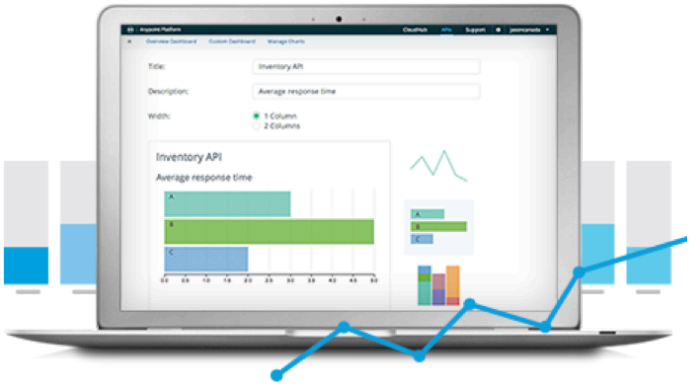
Anypoint Exchange allows you to publish and catalog RAML definitions as well as human-readable documentation for your microservice, which includes rich text, images, videos and attachments. Each API that is cataloged in Anypoint Exchange can have it's own private or public API Portal that is the landing page to learn everything about the microservice. The Anypoint API Portal contains all of the documentation as well as the access control mechanism to request/grant key-based access to the consuming app, this includes requesting tiered access via service level agreements (SLA).
Anypoint Exchange acts as a public / private library of API portals whose scope can embark any one or all of your business domains. An engagement model of this sort acts as the enabling mechanism to realize the ideal of having IT partner with LOBs in your business resulting in the democratization of application development and data access.

On the consuming side, Anypoint Exchange can be leveraged during multiple points in the Microservice lifecycle. It helps with discovery of existing assets that can include best practice templates, RAML snippets, API's (for instance, system API's from the same team/domain). The ability to provide user ratings, user forums and other collaboration tools encourage "Merit based reuse" in the context of microservices.

Contributing and cataloging all assets in one central repository takes the friction out of discovery, drives adoption and accelerates innovation. Note that Exchange can be a central microservice catalog used outside of the context of MuleSoft. Any IT developer, app developer, UI developer can login to Exchange with their corporate SSO, browse and discover microservices, play around with it in the sandbox, request access and start writing code against it in their language of choice. For instance, a Microsoft Visual Studio plugin is available for Anypoint Exchange that allows a .NET developer to browse microservice API's directly from Visual Studio IDE, pull it down and embed in .NET code.

Analysis with Anypoint Analytics

Real-time visibility of the consumption of your microservices is provided with Anypoint Analytics. You can see the frequency of calls, time to completion, policy violations, origin of calls across apps and geographies and much more. The data is visualized in customizable dashboards, available as reports for metering and chargeback as well as ingestible into external analytical tools like Splunk, ELK, tableau, Qlikview, and more.



Decentralized Platform for Microservices

In traditional enterprises, the reality is that there will be a mixed mode for architecture. IT and lines of business could be operating in their own domains. There will be traditional global shared backend systems like ERP's, CRM's, FTP servers, mainframes, etc that are shared by multiple domains. And with microservices architecture on the other hand, certain lines of business may have their own isolated back-end systems with domain specific system/process/experience APIs. The key is for IT to enable the business that owns each domain, irrespective of which architecture they choose.

The Business Groups capability in Anypoint Platform allows IT to decentralize the platform and empower Lines of Businesses. Business Groups can be setup to map to Organization structure. Each Business Group can have its own administrator and RBAC with full autonomy (manage own microservice runtimes, MQ's, APIs).

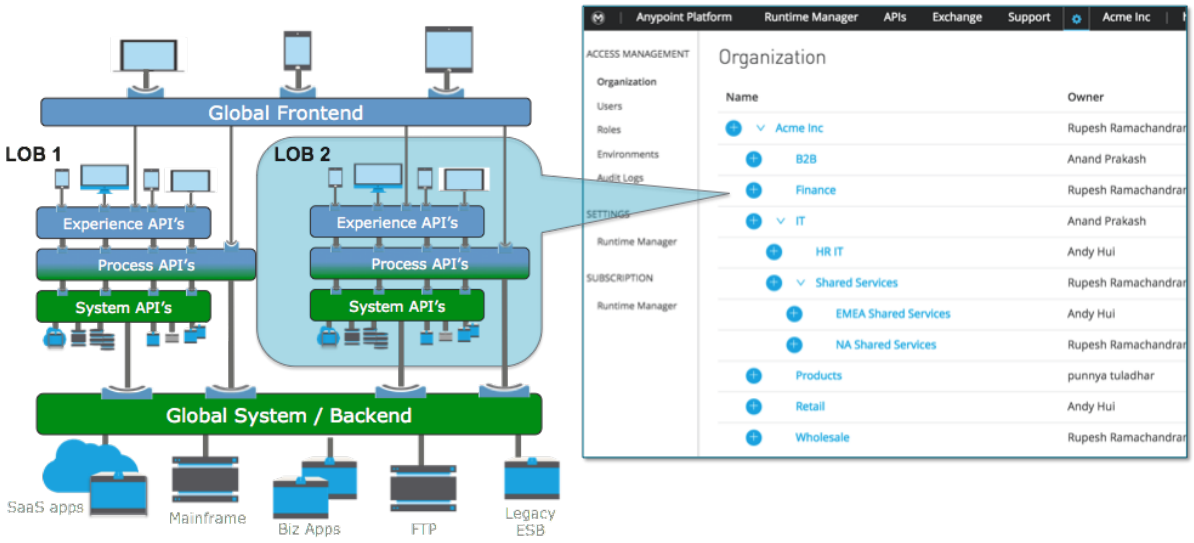


Figure 4: Business Groups feature in Anypoint Platform allows decentralization by domain

Business groups allow intra-domain collaboration and management of API's and MQ's within the domain. Whereas the top-level master group allows cross-domain collaboration and management. For instance, to leverage best practices with Mule, share RAML snippets, Mule application templates, custom connectors, custom API policies, etc. across the entire company, you could contribute these assets to the master group. Democratizing the platform so each business has autonomy for management, is crucial to enterprise wide microservices adoption. At the same time, IT can still take ownership of overarching capabilities like Single Sign On, shared services, overlay security policies configuration (e.g. OAuth server, IP blacklists, threat protection), etc.

In this model, IT truly becomes an enabler for the business's successful microservice rollout. Hence adopting a C4E or Center For Enablement model versus the legacy Center of Excellence model where IT traditionally became the bottleneck for delivery.

Unified Platform for Microservices

Anypoint Platform ties together the entire lifecycle of your microservices deployed across hybrid infrastructure with a single, unified platform. This includes the design, discover, deploy, run, document, manage, contribute and other aspects of the microservice. It also provides a '*single pane of glass*' management UI from which you can manage the microservice runtime, it's API's and its messaging endpoints.

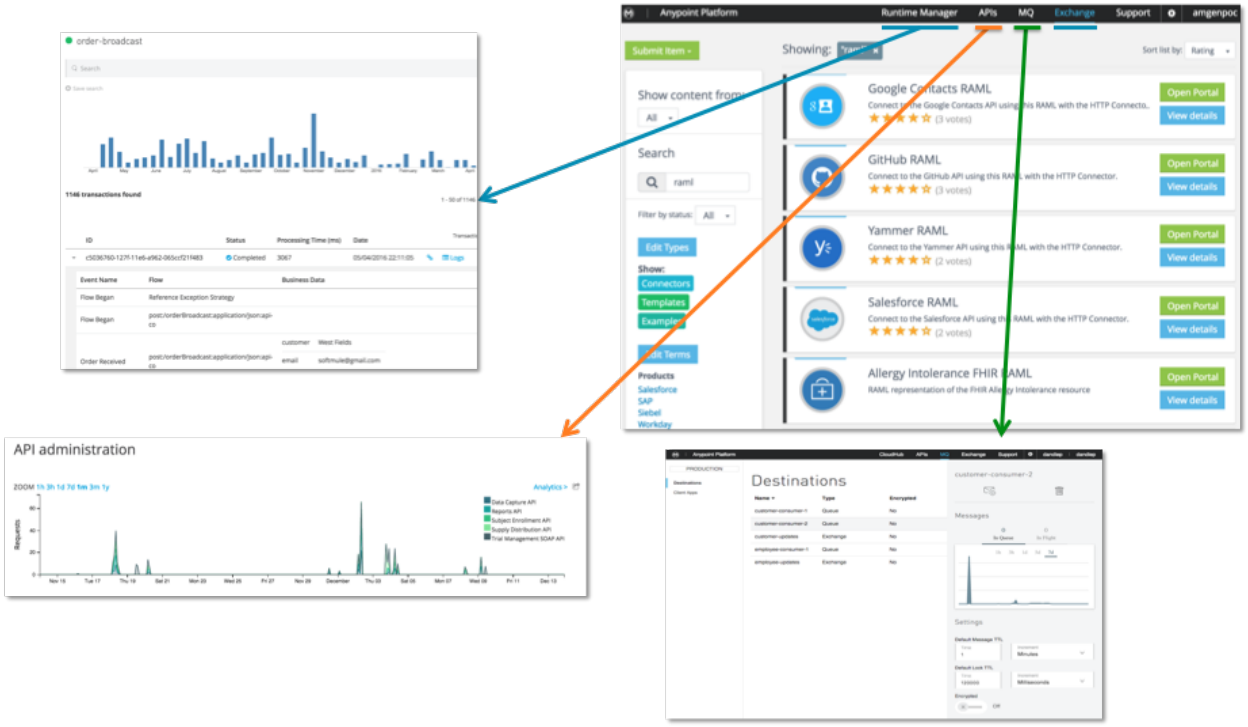


Figure 5: Unified platform with a single pane of glass

The benefits of having a unified platform for your microservice architecture include:

- End-to-end operational visibility. For instance, a slowdown in response times observed in the API dashboard can be correlated to individual transactions that may be erroring out, with the ability to drill down into corresponding logs for further troubleshooting. This reduces the Mean Time To Resolution by providing a single, unified console for operational visibility.
- Tooling leverage. The same skillsets, SDLC tools, single sign on, operation runbooks, engagement/collaboration portal, etc can be used for API management, on-prem microservice runtimes, cloud runtimes, MQ, etc.
- TCO optimization. Integrating the integration stack improves efficiency compared to swivel chair integration when working with disparate tools in your microservice ecosystem. With Anypoint Platform, not having to build your own additional layer to tie all your tools together drastically reduces the complexity, maintenance costs and overall Total Cost of Ownership.

Summary

Microservices is clearly an important and welcome trend in the software development industry, and has many advantages over previous architectural approaches. However, there are various concerns to be aware of when instituting a microservices architecture in your organization. Businesses need to implement microservices because of its ease of deployment and agile nature, but if not managed properly, this architecture can create disorganization and lack of governance. Products developed with a microservices architecture will also need to be integrated with legacy technology stacks, and if this is done poorly, it can create technical debt and more operational costs for the IT team. Therefore, instituting microservices in a way that will create competitive advantage and help your company innovate faster goes beyond a mere selection of products and software. You must also consider the people, process, and culture within the organization.

This is why we recommend a holistic, platform approach to microservices, centered around API-led connectivity. Not only does API-led connectivity create the integration component so crucial to the proper function of your technology stack, it will allow developers inside and outside the central IT team to create new solutions in a manageable, reusable, and governed way, eliminating concerns of too many applications that the business cannot control. In addition, MuleSoft's platform approach provides a unique operating model to allow both LoB and IT to build, innovate, and deliver new solutions wherever needed throughout the organization. Take a look at more resources on API-led connectivity and our vision for changing the organization's culture and process to enable IT to deliver faster at a lower cost.
In today's hyper-competitive business environment, it's important to stand out and provide a delightful experience for customers, employees, and partners. Microservices are a key way for a business to do that. Done in a holistic, manageable fashion, microservices will become a technological standard for the enterprise.