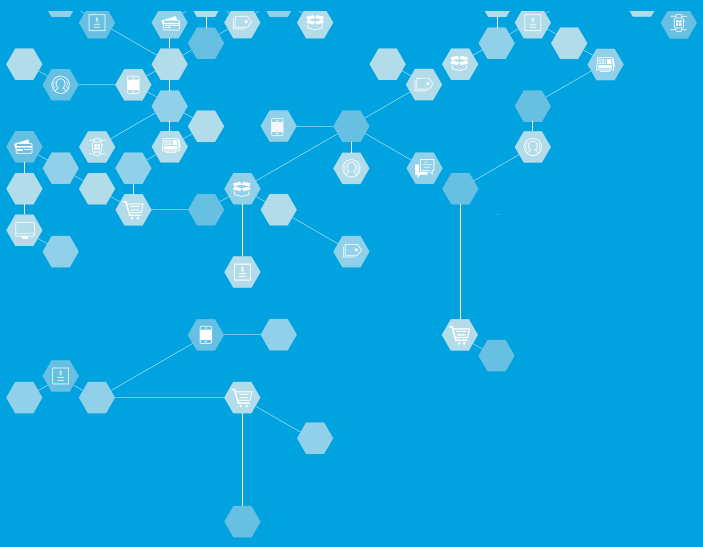
WHITEPAPER

MuleSoft



THE TOP SIX MICROSERVICES PATTERNS

HOW TO CHOOSE THE RIGHT ARCHITECTURE FOR YOUR ORGANIZATION

EXECUTIVE SUMMARY

For the last several years, microservices has been an important trend in IT architecture. Technology consulting firm Thoughtworks has declared that "a microservices architecture as programming model" is one of the four rising trends of 2017[1], whereas ZDNet said it was a "technology to watch" in 2016[2]. Clearly the press is expressing its endorsement of microservices, which might make architects and IT executives feel a fear of missing out on the next exciting trend.

The problem with the supposed imperative to adopt microservices is that there are many people who feel that it is a prescriptive architecture; it must be done one certain way or it simply can't be done. But that point of view is not feasible for many organizations and, worryingly, can lead to failure. For organizations that have particular structures and cultures, this purist view can only go so far because of various legal, technological, and cultural constraints. Organizations will fail if they follow an overly prescriptive point of view in the microservices space if their needs are not compatible with the purist approach.

For example, MuleSoft works with a global food and beverage provider, who, because of the nature of their business, aren't going to do 100 updates a day like a Netflix might. At most they will do an update every 10 days, because of the different nature of their business. So why do they have to deploy what a purist says they have to do when they don't have those needs? Large enterprises with a lot of legacy systems have different needs than a younger company, so their culture might not fit with a purist approach to microservices either. Finally, some enterprises, particularly in highly regulated industries, have different security or compliance needs than a Netflix or a Spotify. In that case, constant software updates might not be feasible or even legal, further necessitating a departure from "pure" microservices.

In reality, trying to adapt to a microservices architecture needs to be done pragmatically, in a way that makes sense for your organization. Not every company has to be a Netflix or a Spotify. You can adopt microservices in a way that meshes with your culture, your goals, and your own organization.

Instead of adopting microservices as a singular approach — which would defy the point of the architecture — we propose considering microservices as a series of overlapping patterns which you can pick and choose to adopt depending on your own situation. There's no need to adopt every aspect of microservices all at once; it can be adopted in a pragmatic way in a manner that makes sense for you.

---

[1]Krill, Paul. "Docker, machine learning are top tech trends for 2017." November 7, 2016.

[2]Hinchcliffe, Dion. "The enterprise technologies to watch in 2016." May 29, 2016

INTRODUCTION

In 1987, chef Ferran Adrià heard the seminal phrase "creativity means not copying[3]"; this simple, self-evident statement drove a major movement within the culinary world. We can state a similarly simple, yet potentially profound statement with microservices: "microservices are not a monolith."

Originally created as an alternative to a monolith, microservice patterns should inherently be non-monolithic: change is easy, units are small, scalability is semi-infinite. This also means that microservices are not just one thing. Rather, we posit that microservices are a category of grouped, related patterns that share the same set of goals. This is analogous with database systems: they all share similar goals, typically with different priorities like scalability or maintainability, yet their specifics differ significantly. RDBMSes, NoSQL Stores, Time Series Databases, Big Data Stores all share a similarity — they offer the ability to store and query data — yet the specifics of their individual trade-offs couldn't be more different.

This whitepaper will highlight a taxonomy of microservices patterns that have been practiced in the wild. Each of these patterns cannot be seen as better or worse than another, but rather as choosing a specific set of trade-offs made that prioritize different things along the way. They all obey the microservices architectural goals (speed, scalability, cohesion) but arrive at them in different ways. It can even be argued that some of the patterns are "steps along the way", where they move an architecture forward towards the microservices goals but in and of themselves tend to produce predictable failure conditions that then encourage a team working in this way to seek alternatives, to make harder trade-offs, and then to evolve to a different, more specialized pattern based on their growing experience. This is certainly how microservices architecture came to be: starting with a conceptual approach and set of goals, that resulted in a first iteration, then iterating relentlessly towards the more specialized (many would view as more "extreme") patterns. This is not a bad thing; organizations that are implementing or living with these early stage patterns have not done something retrograde, but rather have improved their current state to the stage where they now can clearly see the benefits of taking the next step. This is one way that microservices architectures are very "real-world"; they follow the pattern of human learning. At the end of the day, implementers (at least for now) are all humans, and need to learn the value of more specific patterns and trade-offs through personal experience, rather than by slavishly following any particular text or blog post.

Microservice architectures should be evolutionary. Because they are by goal defined by an ability to change rapidly, the approach to building a system develops over many iterations. It is relatively uncommon in most organizations to have the luxury of implementing a microservices architecture from the ground up (and many consider it an anti-pattern to do so), and as such there will usually be a continuum of architectures, patterns, and technologies in play based on maturity, need and timing. Some of this is driven by a team's ability to absorb change, some of it depends on setting up enabling infrastructure, and some of it depends on organizational change and restructuring teams (for example, around DevOps practices). The ability to focus on what can be done right now while enabling teams to keep evolving towards a more productive state is one of the key strengths of microservice architectures.

---

[3] Adrià, Ferran. The History of El Bulli.

The 6 microservices patterns

Below is a taxonomy of the 6 microservices patterns available to make adopting this architecture an easier task for your organization. Each of these patterns is not a general pattern; each one chooses to make a set of trade-offs that prioritize particular things over others. As such, it is our hope that users who implement a particular pattern will use this as a reference in order to plan their architecture following these patterns, rather than attempting to over-apply any single pattern and therefore break it through compromise. The intended state is to design and implement a microservices architecture using a mixture of these patterns, rather than to choose one and migrate towards it.

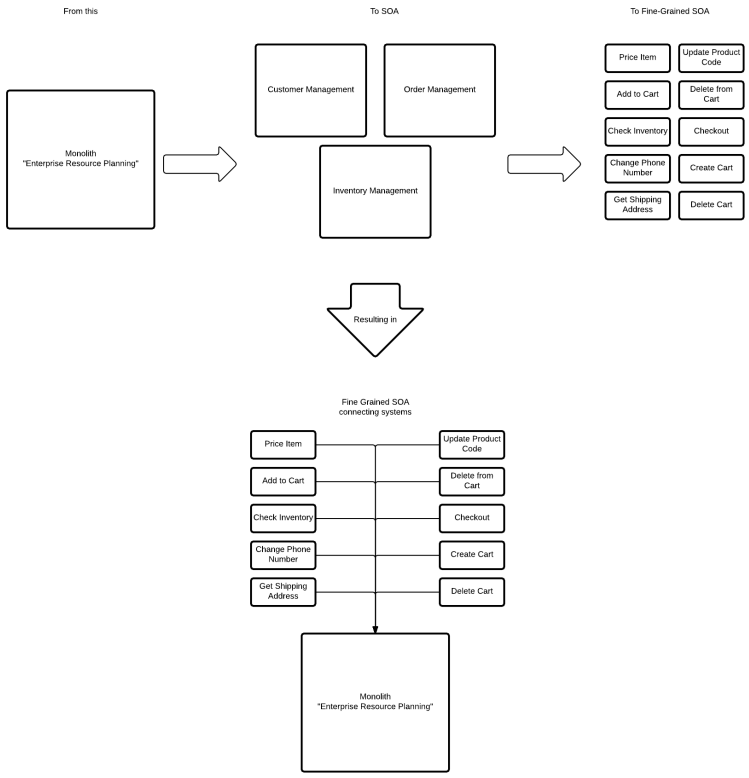
No single one of these patterns is better than the others, but each is uniquely better at a particular task. We attempt to make it clear what that task is when describing each pattern.

As a final note, we should emphasize that microservices architecture is not appropriate for every use case. If you depend heavily on an ERP application, for example, and have no intention to remove it, microservices probably aren't for you. It's unlikely, however, that there aren't parts of your business where speed, scale, and cohesion would be of benefit. Those are the areas where you should look to take advantage of this rapidly growing ecosystem.

These patterns don't apply specifically to any one scale or type of organization. It's true that the patterns that address state management specifically tend to be more useful to large scale, fast moving orgs, but they can be applied to any kind of business or company size that has an appropriate need and is willing to balance the needs of their architecture correctly. In other words, "where you stop" is entirely dictated by the problems you're trying to solve, rather than any linear progression through these patterns.

Section I: Introductory Patterns

Fine-Grained SOA



Fine-grained SOA is arguably the "big bang" of microservices. To many, Netflix is the origin of this style of architecture, and this was their own articulation of the pattern at the beginning. Fine-grained SOA is somewhat self-explanatory (at least, to SOA practitioners): to reduce the issues experienced with SOA and apply the same principles but break down the design into smaller, more fine-grained pieces. This is, in fact, what most people still think of as the only microservices pattern: small services.

However, along the way Netflix (and others following this path) have experienced a number of basic issues with this pattern. When you make things smaller, and attempt to scale them, some difficult properties emerge:

- Where you used to make a single network call, now you must make tens of calls or even hundreds of calls. This is inefficient[4].
- Where you used to manage a small number of things, now you manage hundreds or thousands or more. Your management tools therefore no longer work as well for you.
- When tens, hundreds, or thousands of things all can query or modify the state of your central applications, it becomes almost impossible to trust the consistency of that store as accessed through any one of these microservices.

These are the key overlooked challenges with truly living with microservices: efficient inter-process communications, overarching monitoring, management and governance, and consistent state.

In most cases this pattern is applied as an extension of Service-Oriented Integration, where the point of each service is to provide connectivity to external systems. This forms tight dependencies to those external stores, making speed of change drag, and making the cohesion of the system reflect the internal state of those applications.

When you begin to implement fine-grained SOA, be aware of what you're getting into. These pain points will emerge if you succeed, and you should be ready for them. When the pain becomes significant enough, you will inevitably need to seek out other patterns.

Problem:
Coarse-grained services are too difficult to change without side-effects, and the monolithic nature of traditional architecture is "holding the teams back".

Solution:
Break up services into finer-grained pieces, to reduce the scope of any given change, allowing it to happen more easily.

Application:
Services are broken up into finer-grained microservices, typically each with a single purpose.

Impacts:

- Traffic increases.
- Number of services managed becomes large
- Traditional monitoring solutions become insufficient
- Automation of integration, testing and deployment become critical
- Orchestration of microservices becomes necessary.
- Ability to change rapidly is improved.

Goals:

- Speed of change

---

[4]Note that high-efficiency Inter-Process Communications frameworks are very available now (gRPC, Avro, Hystrix, Finagle) so this isn't a long term concern. Rather, it should be something to flag: if you plan to do microservices, raw HTTP/JSON or XML will eventually become too inefficient for the pattern at high scale.

- Scalability can theoretically be improved but practically tends to decrease unless supporting automation infrastructure is put in place.

Key characteristics:

- Works well at low scale, pain emerges at high scale.
- Focus is on small size of services, but "ilities" (e.g. scalability, reliability, etc.) typically have not been considered.
- Tends to be integration-centric, with each microservice depending on external systems[5].
- Inefficient Inter-Process Communication in order to achieve high speed of change.
- Data consistency and state management is poor[6] but allows leveraging existing systems.
- Due to similarity to existing patterns, existing problems tend to happen.

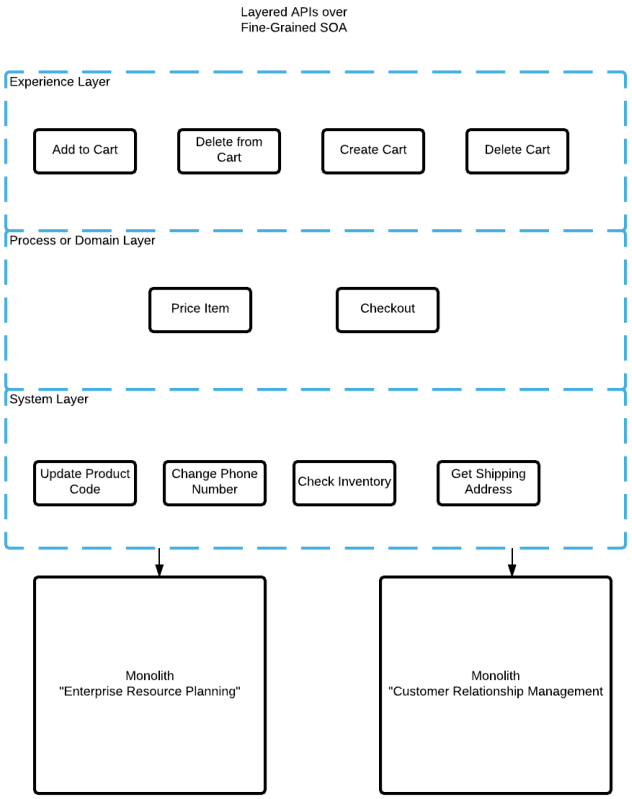
How does this coexist with existing systems, SOA or APIs?

This pattern is very similar to SOA and API-led approaches, and leverages more value out of existing systems. The coexistence model is simple, and as such causes most of the friction with this model since the low speed of change on existing assets can cause a drag on the microservices approach.

Support within Anypoint Platform today or on the roadmap

This pattern is mostly innate to Anypoint Platform, as it is one of the key use cases. However, it can be noted that the Mule runtime consumes ~1GB RAM per runtime at minimum, meaning that isolating a fine-grained design to 1 microservice per container (best practice) can be very resource intensive, although it reduces the negative impact of updating apps.

Layered APIs



[5]Note that generally speaking microservices (and APIs in general) are increasingly treated as "external" assets regardless of if they are or not. This is what we at MuleSoft often refer to as "productised" APIs, where regardless of context every point of interconnectivity is treated as something that is robust and ready to be used across any permissible context.

[6]There are multiple data masters, but the architecture doesn't typically address the existence of a source of truth or use a specific consistency model. There are often multiple ways to make a change to data that can have cascading impact which is difficult to understand or respond to.

MuleSoft advocates an approach termed “API-led connectivity”. Simply speaking, this can be thought of as a layered approach to API design (at least for the purposes of this paper). System APIs expose systems, Process APIs orchestrate them, and Experience APIs provide end-user experiences. This approach is well-aligned with fine-grained SOA, and often either the two can co-exist or else fine-grained, layered APIs become an evolutionary pattern that follows fine-grained SOA.

This approach gives some structure to a fine-grained API approach, allowing some ability to consistently manage and reason with the APIs (or microservices). However, there are some deep issues with this approach, that are basically the same as fine-grained SOA (which is intuitive):

Where fine-grained SOA makes a single network call, now you must make multiple calls through the layers. This can be inefficient from the perspective of “network hops”, however it is key to note that the existence of layers does not mandate them. Calls directly across layers are completely valid, the goal of layering is to increase flexibility while also structuring the architecture in a way that separates concerns well.

Where fine-grained SOA manages a large number of things, now you manage multiple layers of a large number of fine-grained things. Your management tools no longer work as well, as they may not have ways of visualizing complex microservices views.

End-application datastore consistency is actually improved, because the set of things that modify or query them are organised and focused (i.e. System APIs).

Ultimately, this a good pattern for most enterprises, but, like fine-grained SOA, there will be pain along the way. However, this pain exhibits mostly at large scale, so one should only plan for other patterns when high scale is expected or experienced.

Problem:
Microservices architectures without some amount of structure are difficult to rationalize and reason with, as there is no obvious way to categorize and visualize the purpose of each microservice.

Solution:
By creating layers of microservices that are grouped by purpose (systems, processes (or domain models) and experience), you can manage the complexity of the architecture more easily.

Application:
Microservices are categorized into layers. Often, standards can be put in place to make microservices that have similar purposes to behave in similar ways, which further rationalizes complexity.

Impacts:

- Ability to change rapidly is improved through standardization and further decomposition.
- Number of inter-process calls can be increased because of the more specialized microservices structures.
- Monitoring and visualisation tools may need to be reviewed for their ability to work with the layered structure correctly.

Goals:

- Speed of change
- Scalability can theoretically be improved but practically tends to decrease unless supporting automation infrastructure is put in place.

Key Trade-offs:

- Inefficient IPC in order to achieve high speed of change.
- Data consistency and state management is poor but allows high degrees of reuse. Re-use itself trades off against speed of change.

- Due to similarity to existing patterns, existing problems tend to emerge.
- Works well at low scale, pain emerges at high scale.
- High cohesion due to structured architectural approach.
- Focus is on small size of services, but "ilities" typically have not been considered.
- Tends to be integration-centric, with each System microservices depending on external systems. This reduces speed of change.

How does this pattern coexist with existing systems like SOA or APIs?
This approach tends to be the best way to coexist with existing assets. Because the layering reduces the scope of each microservice as well as focuses its purpose, it is able to connect and leverage existing systems best without causing significant slowdown.

However, coordinating changes that ripple through a fine-grained and also layered design can be challenging to work with, and you may need to employ technologies to manage the contracts between all the different pieces or employ thorough automated testing technology to ensure that changes don't break things.

Support within the Anypoint Platform today or on roadmap:
This is essentially the API-led connectivity pattern, but fine-grained and with some new issues around IPC efficiency. This is supported by the Anypoint Platform today, with the same friction point related to resource usage as fine-grained SOA.

A large number of microservices in layers may cause a lot of cross-talk, requiring either more micro-experience APIs to provide facades (which create dependencies that can be difficult to manage) or else a higher performance IPC mechanism. Using third-party tools like gRPC, Avro, Thrift, Hystrix, etc can mitigate some of this, either used within the Mule runtime or working in a proxy configuration.

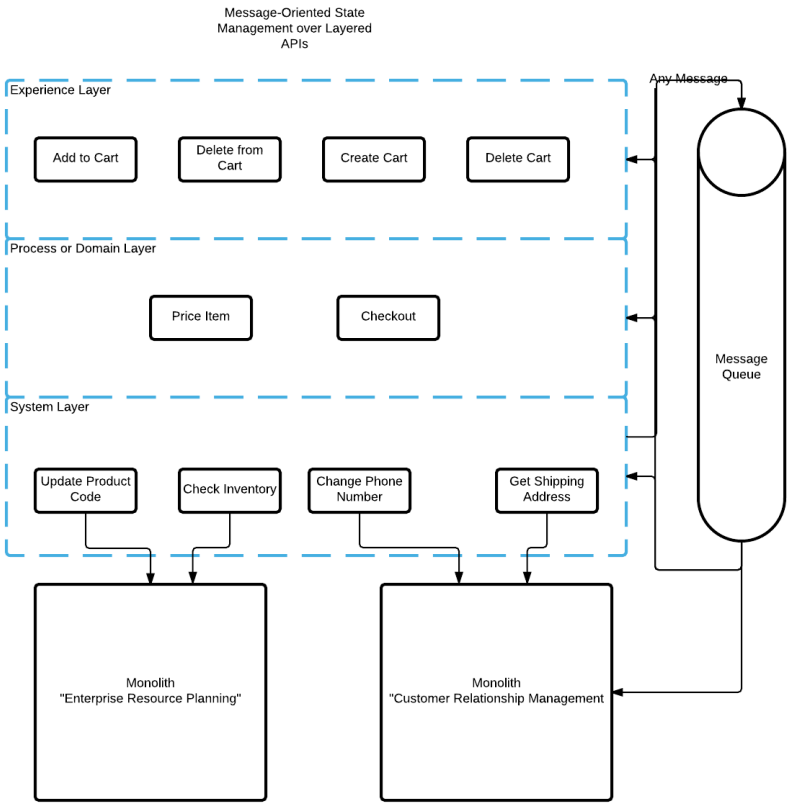
Section II: Managed State Patterns

The following patterns are all focused on managing state. State is ultimately one of the most challenging aspects to a distributed architecture, because traditional system design favors consistent data queries and mutations even though consistency is difficult (if not arguably impossible) in a sufficiently distributed architecture.

Because many of our customers' microservice designs are focused on integration use cases, the issue of managing state comes front and center after even a small amount of success. This is because providing connectivity or integration means that the microservices system is inherently either querying or mutating state (or both), so they may not be the only way to do so for a given entity or piece of information. This means that in order to avoid data corruption or unexpected results, you need to consider one of two strategies:

- Explicitly declare state and use a strategy to deal with the side-effects of mutating and querying it.
- By doing so, achieve a greater level of physical autonomy for each component, allowing a faster rate of change.

Message-Oriented



This is usually the first pattern implemented as a way to avoid the side-effects of accessing and mutating state. By providing an asynchronous queue as the primary mechanism to communicate state changes (by command or event) or to query other microservices, we allow each microservice the time necessary to converge events to provide a consistent external view. This pattern is commonly used by teams experienced with SOA and ESBs, as it is an intuitive path to follow given their prior experience.

We recommend that this pattern be used as a transitory state; this should be used by an organization beginning the microservices journey, but may not suit your needs as your approach becomes more mature. By decoupling components temporally using a queue, the implementation and behavior of each microservice becomes obscured, meaning that often the side-effects of the design become even more prominent than when simply exposing existing systems to a microservices design. It is not uncommon, for example, to see messages used to propagate events, commands, batches and even to just stream data, all appearing at a high level to be the same thing. This can make systems unpredictable, which in more traditional environments (e.g. ESBs) is manageable, but when moving to microservices scale (thousands of services, hundreds of thousands of message types), it becomes rapidly impractical to reason with. This pattern is therefore is often a transitory step, where success translates into teams realizing that at a basic level queueing is useful but in order to really succeed they need to establish some standards around what is passed over the queues.

Problem:
In order to ensure data integrity, there is a need to replicate the state of key business data between microservices or data

stores.

Solution:
Using a message queue allows state to be asynchronously and reliably sent to different locations.

Application:
When a change in data occurs, it is sent as a message over a queue or ESB to any other microservice or store that needs to be notified of the change.

Impacts:

- Can increase complexity as it provides a new way for state to change and move.
- Does not offer any standard patterns, so implementation can be inconsistent unless standards are agreed and applied.
- Does not offer any specific opinions as to how to deal with data conflicts or to rebuild state in the case of failures or outages.

Goals:
Scalability: using message queues provides the means to scale task processors independently from task producers, with a reliability layer between them.

Key Trade-offs:

- Efficient IPC due to asynchronicity.
- Data consistency and state management is made worse by the unpredictability of behaviour and the potential side-effects of a given message. Re-use is actually hampered by the inability to predict the use patterns.
- Due to similarity to existing patterns, existing problems tend to emerge.
- Works well at high scale technically, but tends to become operationally unpredictable.
- Poor cohesion due to lack of design standards.

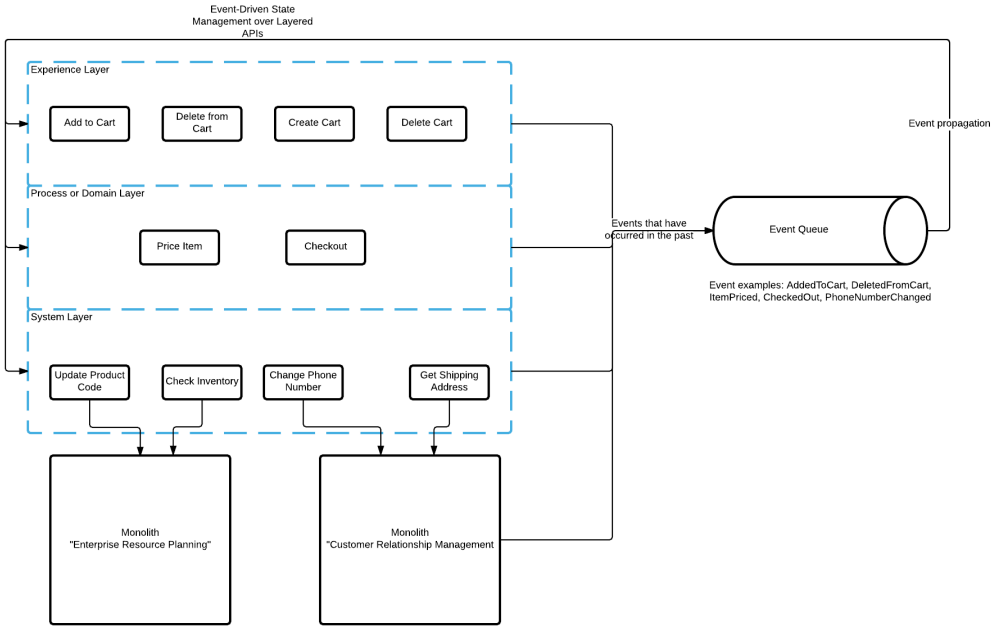
How does this co-exist with existing systems, SOA or APIs?
This approach may be the best way to co-exist existing assets and connectivity with minimal change. By decoupling at the interface level using messages, you gain a great deal of flexibility and reliability. The ability to transform and route messages in the message layer means that changes can be kept isolated to their source, rather than happening at the interface level.

This increased flexibility comes with a lack of visibility, and may require new management tools and troubleshooting techniques (for example, a common step is to add tracking data into message headers to understand the path a given message has taken, for troubleshooting and debugging purposes).

Support within the Anypoint Platform today or on the roadmap:
Currently supported by either the use of Anypoint MQ or a third-party messaging system (e.g. on-prem) such as RabbitMQ, ActiveMQ, NATS and more.

RAML does not currently support message-oriented designs, but stay tuned for updates.

Event-driven



Event-driven architectures are nothing new (Mule ESB was originally designed as an event-driven system, for example), but when overlaid on microservice patterns they provide some powerful abstractions. Event-driven systems usually use a queue of some kind (like message-oriented systems) but enforce a standard around the design and behavior of the what is passed over the queue: specifically, the concept of an event.

People confuse this pattern with other patterns, and as a result it covers a wide array of designs. Strictly, an event is something that occurred in the past with an associated representative state and timestamp. This event allows any service receiving it to reconstruct a materialized view of the state by replaying the events in order. However, in many implementations, the concept is muddied, where events (e.g. something happened) are mixed with commands (e.g. make something happen) and without the distinction, the predictability of the design is flawed. That said, this approach is undeniably better than message-orientation (due to its more specific design), but tends to have problems in implementation due to a lack of consistency. Teams that articulate and enforce a consistent standard will find this pattern tends to work very well in microservice architectures.

Problem:
In order to ensure data integrity, there is a need to replicate key business events to synchronize between microservices or data stores.

Solution:
Use a common event abstraction to represent the unit of change in the architecture.

Application:
When something changes in the business, an event encapsulating it in the past tense is sent to interested parties.

Changes in the business are the product of these events being sent and processed.

Impacts:

- Can increase complexity as it provides a new way for state to change and move.
- Does not offer any standard patterns, so implementation can be inconsistent unless standards are agreed and applied.
- Does not offer any specific opinions as to how to deal with data conflicts or to rebuild state in the case of failures or outages.

Goals:

- Cohesion: this architecture is very easy to work with and understand due to its standardised nature.
- Scalability requires deeper technical decisions (how to send/process/store events? What about retransmission?) but is achievable.
- Speed of change is good due to the more cohesive architecture, but without dependency analysis tooling relies a little too much on tribal knowledge and luck.

Key Trade-offs:

- Efficient IPC due to asynchronicity.
- Flexibility of design is lost in favor of predictable behavior.
- Data consistency and state management are improved through a specific model of consistency: any given state is merely a reconstruction of events.
- Due to similarity to message-orientation, confusion can occur where events are mixed up with commands.
- Effective scaling model with reasonable operational oversight.
- Strong cohesion when applied consistently, but cohesion tends to drift over time.

How does this co-exist with existing systems, SOA or APIs?

Event-driven systems can co-exist with existing systems, but they tend to require a "translation layer" across the boundary of the event-driven parts of the archtiecture with those that aren't. In essence, the event-driven system talks a consistent language internally, and anything on the outside needs to be converted (in or out) to participate.

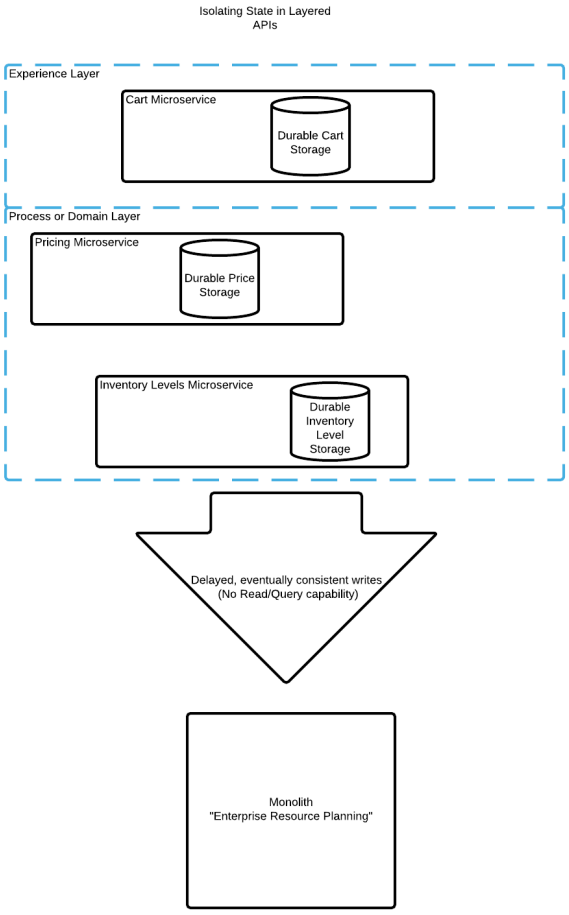
This makes for a clean way to separate the event-driven parts of the architecture from traditional integration and enterprise systems, but it does mean that you tend to create "new" functionality with event-driven microservices that out-of-band update or sync with the systems and APIs outside of the boundary.

Support within Anypoint Platform today or on the roadmap:

Support is limited for event-driven designs. They can be applied through governance and design pattern using Anypoint MQ or a third-party event processing system, but while Mule itself is Event-Driven, there isn't first-class support for event design.

RAML does not currently support event-driven designs.

Isolating State



An alternative to coalescing the exchange pattern of a microservices architecture (for example, into events) is to coalesce the internal consistency of each microservice. Rather than expect consistency in the interchange, expect consistency at the time of query. This is done most commonly by isolating state, or in other words, "each microservice contains its own state." In this pattern, each microservice contains an internal data store that it constantly reconciles with external stores (be they an event log or an enterprise asset) so that it becomes the "single source of truth". This is a challenging thing, since single source of truth patterns tend to echo the complexity of master data management and its associated challenges. However, using an external store as a single source of truth with microservices is much more practical, because of the typically single-purpose nature of a given microservice. It's difficult to isolate the state of a customer, for example, but it's not so difficult to isolate the state of the customer's email address. This pattern, therefore, must by design favor extremely granular microservices in order to succeed, and typically also requires asynchronous event propagation as a means to pass state change from one point to another. This pattern can also be thought of as something of a "distributed database", with each microservice almost representing a column in a traditional RDBMS design.

Microservices contain a data store that is the source of truth for the entity they represent. For example, a "product" microservice could contain a MySQL database that contains all information about product, and is the only way to query or update that concept in the organization.

Unlike more SOA-oriented patterns, reuse doesn't tend to be a priority in these designs. Each microservice has a "use", certainly, often accessed from different contexts, but each microservice isn't designed with reuse in mind. If reuse occurs, it's accidental, not with the intention that comes with SOA.[7]

---

[7]SOA emphasizes designing any one thing to have many uses, to be "reusable". By contrast, more and more specific microservice patterns emphasise "do one thing and one thing well": don't cater to every possible use case, but be focused and simple. Ironically, "reuse" is more common in these designs, because a simple thing that is reasonable tends to be useful in many contexts. Reuse is more successful when driven from simplicity than by intention.

Problem:
It is difficult to achieve data integrity when there are multiple sources of truth.

Solution:
Nominate a microservice that represents the single source of truth for each given business entity, and encapsulate the state inside the microservice.

Application:
Microservices contain a data store that is the source of truth for the entity they represent. For example, a "product" microservice could contain a MySQL database that contains all information about product, and is the only way to query or update that concept in the organisation.

Impacts:

- Requires some governance to ensure that data is not copied or has other modes of access.
- Can be difficult to enforce when working with existing assets such as ERPs, where a "strangler" pattern must be employed to replace the existing system's data stores with the new microservices architecture piece by piece.
- Side-steps the issue of data synchronization, so if data entities get out of sync there is no easy fall-back position.

Goals:

- Cohesion: this architecture is very easy to work with and understand due to its standardized nature.
- Very scalable (each small component can implement its own scaling model).
- Speed of change is good due to the more cohesive architecture, but requires governance to ensure the architecture is not breached.

Key Trade-offs:

- Efficient IPC due to asynchronicity.
- Very flexible design, speed of change is high.
- Data consistency is good: there is a single source of truth.
- Scalability may pose challenges, as scaling a process requires also scaling a data store with it.
- It's difficult at scale to divide a data model into completely independent pieces: at some stage, consistency between the views becomes important.

How does this coexist with existing systems, SOA or APIs?
It doesn't; if you build a microservice with isolated state, it's important that it is the "source of truth and function" for its stated purpose. If you have an existing system that also deals with the same data or function, you will need to synchronize it out of band, and it's typically a bad pattern to implement two-way sync here.

This approach typically pairs well with the "strangler" pattern, where you seek to reduce the use of a given enterprise application or other system that does not give you the time-to-value you need. Over time, you replace its functions with these isolated microservices and you deactivate those particular functions in the original system.

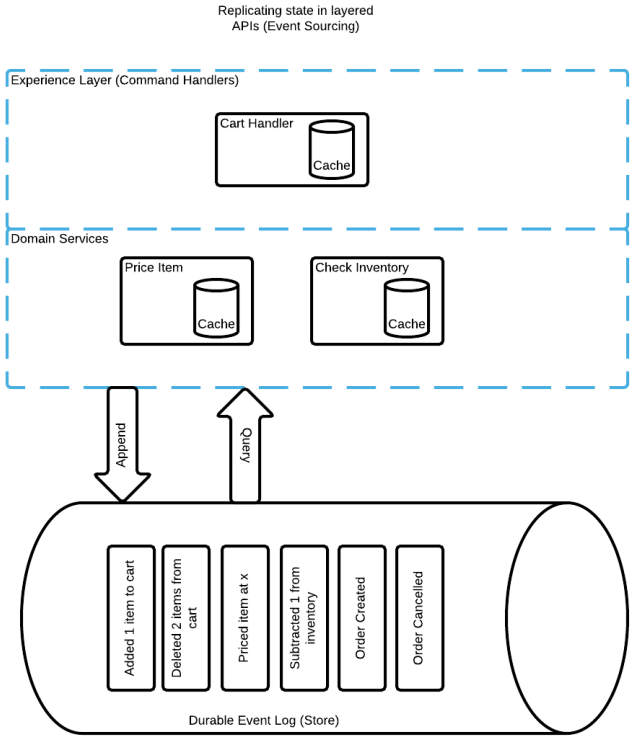
To put it another way: this is not an integration pattern. The goal here is to create a new, fast-moving partial implementation using microservices to gain speed and scale advantages that existing technology has not been able to provide you.

Support within Anypoint Platform today or on the roadmap:

- It is possible to build this sort of system with Anypoint Platform today, but further automation tooling is needed to fully support this pattern.

- Some additional tooling is required to manage starting and stopping instances, which are in future Anypoint Platform plans.
- High-scale data replication technologies aren't native to Anypoint Platform other than used as a backbone for Analytics.
- RAML has no support for data replication but can be used as the contract for a given microservice.

Replicating State



Replicating state is essentially the antidote to the problems that emerge from isolating state: specifically, that consistency is required. A simple example is if we imagine a Catalog, Pricing and Currency microservice. If each of those contains an isolated state of each thing, they become inter-dependent: failure or change in one can cause the function of the other(s) to fail.

This problem is addressed by replicating state: providing a single place to store all state mutations that each isolated microservice can rebuild its internal state from. Often, this is coexistent with event sourcing, where event-driven microservices communicate exclusively via an immutable event log, providing a separate single source of truth that is consistent but difficult to query. The microservices that provide the ability to query state, therefore, have done the work of "materializing a view" of the event log.

This design is, by nature, eventually consistent. While this may seem like a problem in traditional transactional design, it is ameliorated by insight to the nature of the design. For example, one might think of a debit to a bank account as inherently transactional, but most modern banks have realized that it's easier to create an eventually consistent debit (debit if the account exists, and then insure against the possibility the funds are not available) than to expend effort on ensuring every single transaction is consistent. This represents a newer way of thinking about IT systems, but enables greater freedom and speed of change - and therefore, faster time to value.

Replicating state, of course, is challenging. It requires a deep understanding of the state being managed, and the behaviors

of each microservice in order to be predictable. However, it also directly addresses the problems that emerge from other patterns, and as such can be seen as a very specific trade-off: eventual rather than direct consistency, cohesion over top-down design, and speed of change over predictability.

Problem:
It is difficult to achieve data integrity when there are multiple sources of truth.

Solution:
Keep a single source of truth of all changes to data, and replicate the data as needed.

Application:
Send all changes as events to a permanent Event Log. When needing to query data, build a "materialized view" by computing all the changes from the event log.

This is often streamlined by creating snapshots along the way of the views so that full recomputation is not required every time.

Impacts:

- Creates a very cohesive architecture.
- Extremely scalable, due to the inherent Command-Query Request Separation in the design.
- It can be difficult to visualize and understand logical dependencies (physical dependencies have been explicitly reduced).

Goals:

- Cohesion: this architecture is very easy to work with and understand due to its standardized nature.
- Very scalable.
- Speed of change is excellent.

Key Trade-offs:

- Efficient IPC due to asynchronicity.
- Very flexible design, speed of change is high.
- Data consistency is good with caveats, but there is a single source of truth (typically the event log).
- Scalability is effective, this design prioritises the ability to scale each piece independently.
- Autonomy is very high, at the expense of a complex model.

How does this coexist with existing systems, SOA or APIs?
Ironically, this approach can co-exist very well with existing systems, with only one key change: the event log must become the "source of truth" for anything it contains. This means that existing systems and APIs can continue to be used, as long as they update the event log and are updated from it.

This approach can also be used in a strangler pattern way — by migrating event by event to this approach for the services you need to display speed of change and high scale, replacing the existing implementation but with a stable pattern to keep the existing system in sync if you wish.

Support within Anypoint Platform today or on the roadmap:

- In addition to Anypoint Platform, this pattern tends to require automation tooling to manage starting and stopping instances, which is part of future plans.

- High-scale data replication technologies and event logs aren't native to Anypoint Platform other than used as a backbone for Analytics.
- RAML doesn't currently support event or command sourcing, but stay tuned for updates.

Section III: Foundational best practices for establishing the microservices patterns

After reviewing the microservices patterns, and selecting which ones make the most sense to adopt for the rest of your organization, you might be tempted to stop there. However, in order to make the architecture work, there are a number of foundational best practices your organization needs to adopt in order to get microservices functional in your organization. These best practices are outlined below.

Anti-fragile Software[8]

Key to operating a complex infrastructure at high scale is creating a sense of predictability. It becomes important to ensure that your software is designed to be robust in the face of failure on all fronts: you can't rely on your infrastructure to be resilient.

This leads microservices developers towards practices that encourage uninterrupted operation in the face of broad systems failure. While the concept of "Chaos Engineering" (also pioneered by Netflix) has existed for some time, the key lessons weren't that broadly adopted until the advent of microservices. "Anti-fragility" is a combination of mindset and specific practice, and some of the key practices include:

- 12-factor app principles are useful, but shouldn't be over-applied. Pivotal Cloud Foundry was originally based on these principles, but experience with the myriad of possibilities in real world scale has recently led to relaxing some of the more extreme aspects. Nonetheless, most of these basic principles apply in any situation (e.g. logging as event streams).
- Use intelligent defaults: if configuration files or environment variables are unavailable for some reason, software should initialize into an operational state by using reasonable default values.
- Manage working directories and temp files: how many times have you had to fix an error caused when an app tries to write to a directory that doesn't exist or the app doesn't have privileges for? If your app needs access to a given file or directory, make sure that the code that accesses it checks for access and creates the file if it needs to.
- Avoid race conditions and orchestrated startup: in most cases, software is simple enough that it should be launchable in isolation, and seek to converge with other components. Many apps today require a specific start order (especially: first the database, then the app server), and this isn't necessary. Instead of erroring if a connection is unavailable, start up anyway and try to reconnect on a backing-off timeout.
- Make bootstrap bulletproof: you can probably sense a theme from these items: the goal is to make sure that software components can start without error (regardless of the runtime environment) and over time seek to reach an ideal state. The benefits of this approach are broad and obvious, but the basic guideline should be that operators never need to understand the internals of the software.
- Circuit Breakers: are a pattern (in microservices deployments, often implemented as part of a high-performance inter-process communications framework like Hystrix or Finagle) that detects failures and provides logic to prevent them from reoccurring. In other words, it detects an error condition and prevents components from attempting to retry the "doomed" action until the error condition resolves.
- Use Timeouts - similarly to using intelligent defaults, any external communication should include a timeout. Further, if many services are involved in and end-to-end scenario, it is important to think of timeout "budgets" - for example, the

[8]In other words, software that is stable and resilient.

timeout value for the third call in a chain should not be the same or longer as the timeout for the first call, or else you can have components still processing "down the chain" for calls that have already terminated at the edge.

HealthZ

A common pattern with microservices is "healthZ", or, in other words, apps exposing a known endpoint (/health in Spring Boot, /healthz as a common pattern) that returns a simple health check. This check should indicate two things:

- Does this app think that it is healthy? (Yes: 200, No: 5xx).
- What does it believe its current state to entail: return a basic set of information (e.g. JSON) that describes the current state of its internal dependences. These should be readable to an operator (not a dev) and be self-explanatory - for example "database: connecting".

Note that health and correct operating state are different: it is common in container frameworks that a component can be running and healthy but not yet "ready" for serving traffic (e.g. the database isn't connected yet). HealthZ endpoints should indicate when something is really wrong, not that there is a temporary operational blip.

"Infinite" (linear) scale

Microservices patterns focus a lot on scalability, often with the term "infinite" scale being bandied about. Of course, this doesn't mean truly infinite scale, but is more a shorthand for the idea of having a clear understanding of how it would be possible with a given piece of software to achieve a linear scaling model. Frequently (as you will see in the patterns below), this focuses on the hard limits to scale: storage of data and state management. With microservices, this encourages developers to think about the patterns of data and storage they are using, and seek a way to perform the task that can support high scale: for example, by using eventually consistent clustered storage instead of relying on transactional boundaries provided by RDBMSes. Generally, this is a good practice. It should be stressed that this point shouldn't be overthought, but there is indeed value in identifying what your storage layers are being used for and making sure you use the "best tool for the job".

Minimize dependencies

When you're deploying many changes frequently, it becomes important to ensure that your component has minimal dependencies on external systems (for example, through using queues for communication rather than synchronous request/reply patterns. While microservice patterns make this almost mandatory, it can be noted that it's useful in general to make each component as self-sufficient as possible. Again, this shouldn't be overthought, but a good rule of thumb is to try to make every unit of deployment as self-contained as you can.

Monitor everything

A sufficiently complex infrastructure requires visibility. Microservices practitioners learn the hard way that it's essential to have an effective monitoring solution baked into their software, and it's hard to argue that this shouldn't be the default state for any well-made app. In an ideal state, it is possible to instrument software without requiring changes in the application code, but following basic practices (e.g. logging as event streams to STDOUT/STDERR) make it easier to put monitoring infrastructure in place. This is only half the battle, of course: more monitoring data creates a need to comprehend it better. The topic of how to effectively design monitors and manage monitoring infrastructure is out of the scope of this paper, but there are many specific texts on this topic, such as James Turnbull's "The Art of Monitoring".

Reduce Batch Size

As has been noted above, in Lean Manufacturing we have learned that there is significant value in reducing the batch size. Microservices as a set of patterns explicitly leverage this approach: the smaller the unit you work with, the simpler each unit is to operate. Of course, conversely, the larger the number of units you have, the more complex the management gets.

However, even without adopting microservice patterns, you can seek to reduce the size of your unit of deployment: by making changes more frequently and in smaller amounts. Even when deploying a change to a monolith, the disciplines that come with implementing smaller, more frequent changes are valuable.

Containerization

This is for a good reason: the use of containers to structure, isolate and manage units of deployment is generally useful, even when applied to monolithic software. With recent data showing that the use of containers brings no performance or security overhead, it is easy to recommend the use of docker as a basic unit of deployment. Better yet, microservices patterns generally require containerization techniques, so adopting containers is a good step towards being ready for a microservices journey.

Focused (Domain-Driven) Design

Domain-Driven Design has risen to a new popularity due to its high applicability to microservices. While in our experience overuse of DDD can be a distraction in a high-scale microservices environment, it's inarguable that the practices proposed by DDD are of high value and make a lot of sense to apply in general. Further, we might even argue that DDD is more appropriate to a monolithic design than it is to microservices patterns, and as such adopting domain thinking within your software will both be helpful as well as paving an easier transition towards microservices when (or if) you need to make the change.

Conclusion: Establish a microservices pattern that's right for you

Now that you have read through the different microservices patterns, it's important for you to assess which ones work best for your organization. It could be one, it could be a set, or you might decide to stay with a monolithic architecture. The important thing is to remember that microservices are not a cure-all that will solve all of your problems. It is an architecture designed to overcome obstacles that, when deployed correctly, will produce certain desired results.

It has yet to be seen how the patterns discussed here will evolve, and how software may rise up to simplify the challenges of this highly complex approach. It is clear, however, that many enterprises will move to adopt microservice patterns, and we hope that by providing some definition of these patterns we may help many of these organizations to avoid the most obvious and painful mistakes along the way.

For more information on establishing a microservices architecture in your organization, take a look at our whitepaper, Microservices Best Practices, or take a look at the demos in our microservices webinar.