MACH Alliance

2024

Interoperability

PART 3

Understanding composable architectures

Key concepts, patterns, non-functionals and assumptions

Overview

Whether you've read the whitepaper **“How to evaluate and integrate composable solutions: MACH Interoperability #1”** or are starting here, the concepts from both these whitepapers are important foundational starting points for your composable journey.

Here we delve into why it’s important to identify the core architectural and design principles that will steer a MACH ecosystem. These principles serve as the bedrock upon which a composable architecture is constructed, and they guide in making consistent and effective decisions throughout the development and operational lifecycle.

These principles are crafted to ensure seamless interaction, scalability, and resilience in an ever-evolving business landscape.

Architectural and Design Principles

The following section will describe these principles in detail, providing the rationale and implications for each.

Simple components

Simple components consisting of few subdomains are easier to understand and maintain than complex components.

Team autonomy

A team needs to be able to develop, test and deploy their software independently of other teams.

Fast deployment pipeline

Fast feedback and high deployment frequency are essential and are enabled by a fast deployment pipeline, which in turn requires components that are fast to build and test.

Support multiple technology stacks

Subdomains are sometimes implemented using a variety of technologies; and developers need to evolve the application's technology stack, e.g. use current versions of languages and frameworks.

Ideals pattern:

- **Interface segregation**
  Instead of an interface with all possible methods and data clients might need, there should be separate interfaces catering to the specific needs of each type of client. I.e. the Backend for Frontend (BFF) described below. This is true for all touchpoints to all components, not just experiences and this is where the ports and adapters come in. Also described below.

- **Deployability**
  Ensure that the teams take responsibility for the services they provide and that their consumers are happy with the service levels and the availability.

- **Event-driven**
  An event driven architecture is more able to meet the scalability and performance requirements of a composable business.

- **Availability over consistency**
  The CAP (Consistency, Availability, Partition tolerance - https://en.wikipedia.org/wiki/CAP_theorem) theorem states that as you partition a network you can choose either availability or consistency of data ("CAP theorem"). As a composable or MACH architecture is by definition partitioning the network through the use of microservices, or modules, an enterprise must make a decision on whether to strive for consistency or availability of data in the system. Availability means a more performant application. A faster website means potentially lower bounce rate of your customers and a higher Search Engine ranking.

- **Loose coupling**
  When you think of a well run organization, you think of departments that are collaborating effectively when needed, where communication is efficient. The loose coupling in a MACH architecture that makes use of concepts such as microservices, domain-driven design, events and hexagonal ports and adapter patterns can be thought of in the same terms as that well run organization. This flexible setup allows for easy adaptability, resilience, and scalability, much like agile departments within an organization. Within this architecture, each "department" or microservice can work in isolation, yet efficiently integrate for mutual goals through event-based communication and a boundary-protecting hexagonal structure. The result is an IT infrastructure that enhances business agility, reduces costs, and mitigates risk by allowing various components to evolve independently without causing disruptions across the system.

- **Single responsibility**
  As the name suggests, each domain will have a single responsibility, and this ensures that we can understand and manage a complex architecture. This is akin to having specialists in a business setting, where each expert focuses on one specific area of competence. This specialized focus makes it easier to manage, update, and scale individual components without impacting the rest of the system.

Key Architectural Concepts

Event Driven Architecture

Being able to respond swiftly and efficiently to evolving demands is vital. An event-driven architecture (EDA) is designed with exactly this kind of responsiveness in mind. In simple terms, this kind of architecture is built around the detection and reaction to events or changes that occur in a system or an external environment. An event refers to a significant change in state: for instance, the placing of a customer order, or the updating of a product's inventory status.

In an event-driven setup, different components or services in the system communicate with each other through the emission and reception of events. Instead of being continuously active, services are lying in wait, springing into action only when they receive notifications of events that are relevant to them. This ensures an efficient use of system resources, and allows for a more flexible, decoupled architecture where components can operate independently of each other while still collaborating seamlessly. The benefits of this approach are manifold. It fosters a dynamic environment where the system can swiftly respond to changes, paving the way for real-time processing and analytical capabilities. Moreover, it enhances scalability, allowing our enterprise to grow and adapt without being held back by rigid structural limitations.

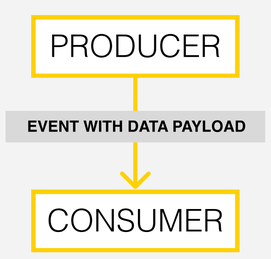
As we navigate further, we will delve deeper into the strategic advantages this approach and these principles brings to an enterprise: the nimbleness to adapt to market shifts dynamically while maintaining a high degree of reliability and robustness in operations. Implementing an event-driven architecture is, therefore, not just a technological enhancement, but a strategic imperative, putting us firmly on the path to operational excellence and heightened responsiveness to business triggers and customer demands.

There are many architectural patterns that enable an event driven architecture. Here are some of the prominent patterns that we have found to facilitate a MACH architecture.

Poor performance and latency

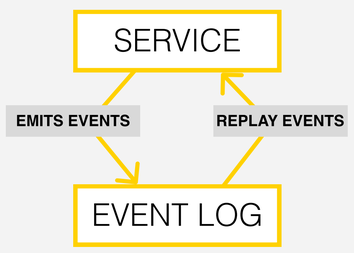
A source service sends a notification to interested consumers that an event has occurred, without expecting a response.

Event-Carried State Transfer



Instead of merely notifying about an event, the event carries a data payload, allowing consumers to act without needing to query the producer.

Event sourcing



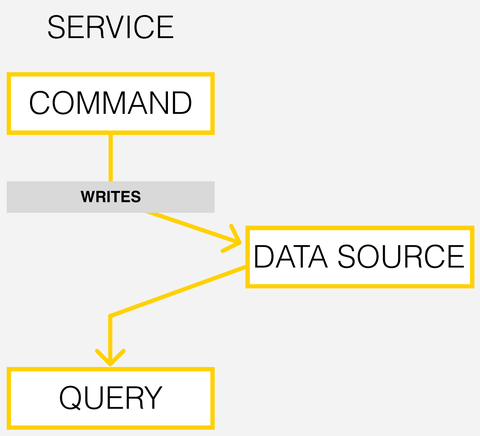
Changes to the application state are stored as a sequence of events. These events can then be replayed to recreate the system state, which is particularly useful for system restores, auditing, and debugging.

Examples

The event driven architecture excels in business verticals that require high levels of scalability, real-time data processing and systems integration. Good targets for MACH and composable architectures.

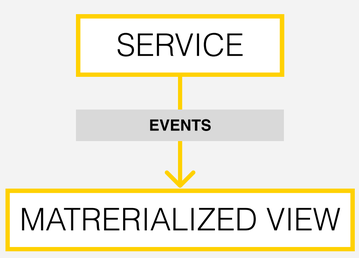
We can see that e-commerce, with its inventory management requirements, personalized customer experience and multiple collaborating systems is a good candidate for EDA. Financial services such as banking or insurance that deal with lots of data movement, stock trading and real-time transaction processing coupled with fraud detection that is often delegated to a third party system can benefit from EDA. Internet of Things, Healthcare and Supply chain and Logistics all have real-time elements or are integrated with multiple systems and need to be performant and reactive.

Command Query Responsibility Segregation (CQRS)



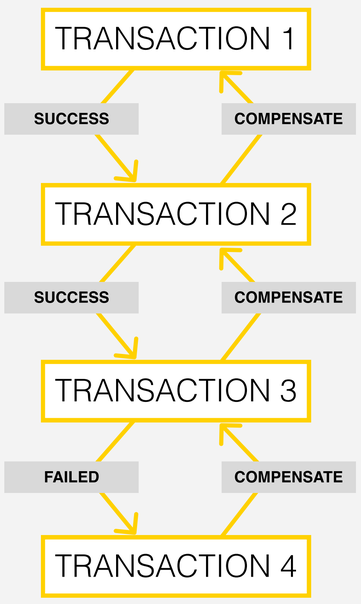
This pattern separates read and write operations. It pairs well with event sourcing, where the write model generates events, and the read model consumes those events to provide a real-time view of the data.

Materialized View



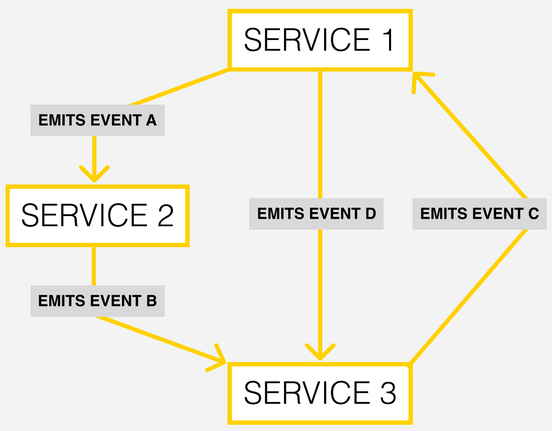
This pattern maintains a view of data shaped specifically for querying by listening to the event stream and updating itself. This is sometimes used in conjunction with CQRS.

Saga Pattern



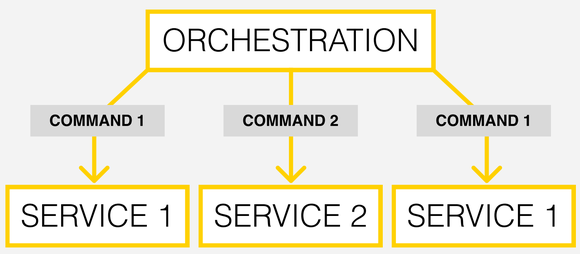
In distributed systems, long-running transactions are broken into smaller, isolated transactions (or sagas). If a saga fails, compensating sagas can reverse the changes.

Event Choreography



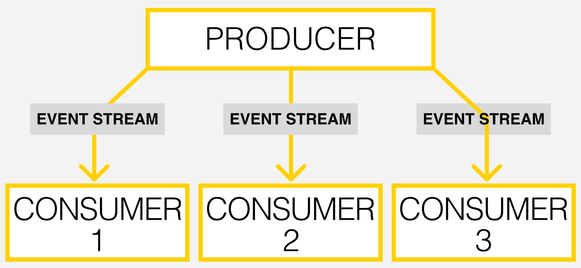
A source service sends a notification to interested consumers that an event has occurred, without expecting a response.

Event Orchestration



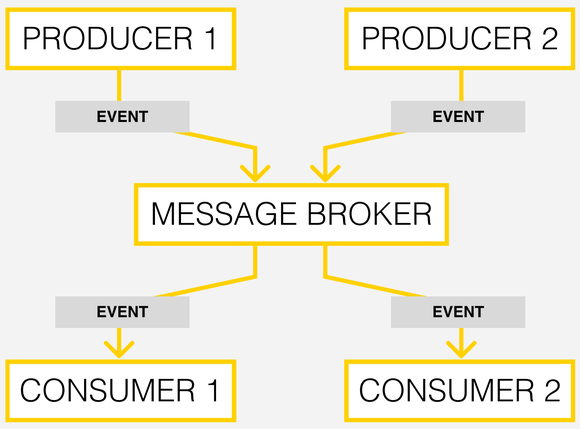
A central service or orchestrator takes responsibility for sequencing business processes by directing other services to execute using commands.

Event Stream



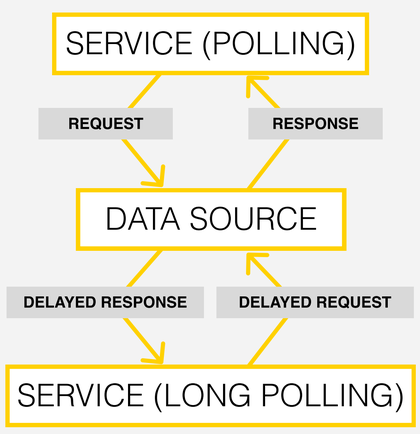
A continuous flow of events is provided, often with the help of platforms like Apache Kafka. Consumers can subscribe to, process, and potentially store these events.

Publish-Subscribe (Pub-Sub)



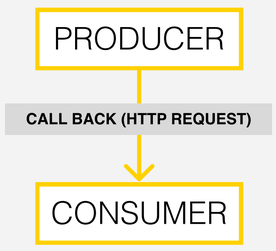
Producers (publishers) create messages without knowledge of subscribers. Subscribers express interest in certain events and process the events they receive.

Polling or Long Polling



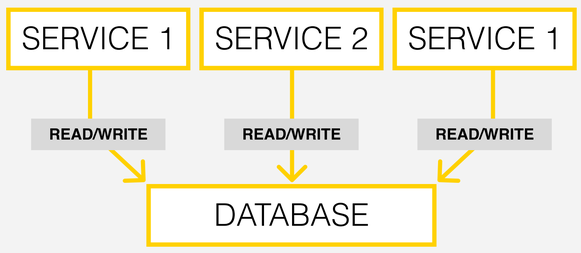
Services or clients periodically check for new events. With long polling, the server holds the request until new data is available.

Webhooks



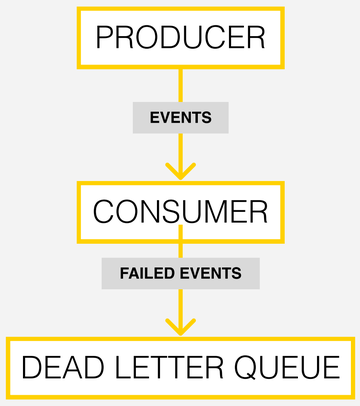
Producers call consumer-specified HTTP endpoints (callbacks) to notify about event occurrences.

Shared Data/Database



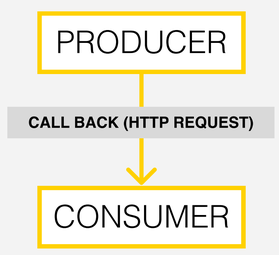
Although not considered a "pure" event-driven pattern, it involves multiple services reacting to changes in a shared data source.

Dead Letter Queue

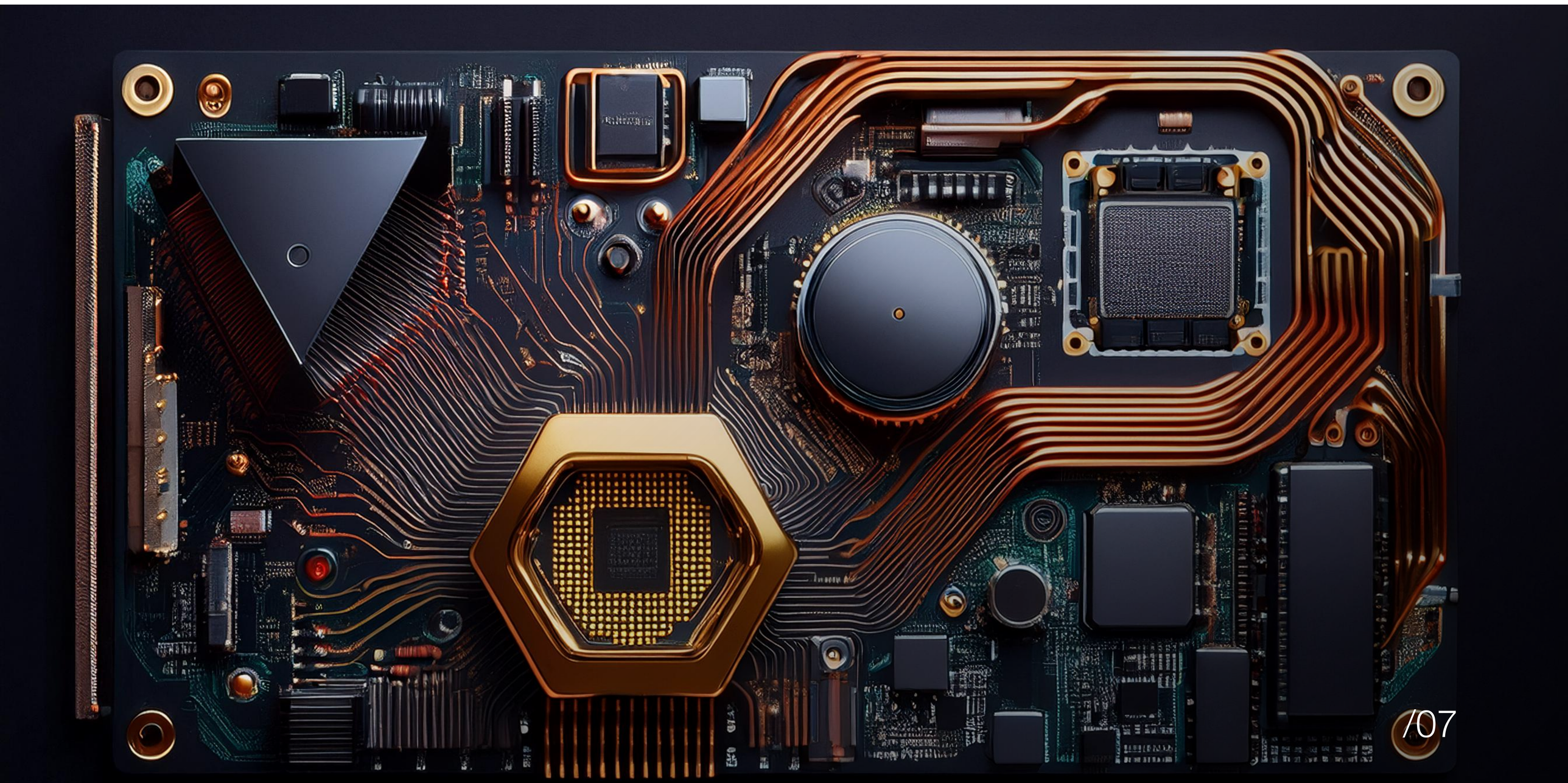


This pattern manages events that could not be processed, ensuring system resilience and providing a way to diagnose processing issues.

Webhooks



Producers call consumer-specified HTTP endpoints (callbacks) to notify about
event occurrences.



Domain Services

Domain services are a well understand aspect of Domain Driven , business functionality and data are encapsulated in standalone units, each focusing on a single responsibility that corresponds to a specific concept within the enterprise. Such a unit is often termed as a 'domain service' and can be composed of one or multiple microservices. Let's break down this idea further for better understanding:

Encapsulation of Business Functionality and Data

By encapsulating both business functionality and data into a distinct unit (e.g. an order, or the concept of a customer), the pattern aims to create a self-contained environment that handles a specific role. This offers several benefits including ease of maintenance, better testability, and improved data integrity.

Standalone Unit with a Single Responsibility

The concept of single responsibility in this context means that each unit or domain service has one, and only one, reason to change. For example, if a unit is designed to handle search functionality, all logic and data related to searching would be contained within that unit. This promotes a separation of concerns and makes it easier to manage and extend functionalities.

Centered Around a Single Concept in the Enterprise

Each unit is aligned with a distinct business concept or domain within the enterprise, such as customer management, vaccine management, search, or order processing. By organizing services around business capabilities, this approach ensures better alignment between the technical architecture and business goals, making it easier to adapt to changes in business requirements.

Formed by One or Multiple Microservices

Although a domain service can be a single microservice, it's often beneficial to use multiple interacting microservices to fulfill complex responsibilities. For instance, an "Order" domain service could involve microservices for payment processing, inventory checks, and shipping logistics. These can work together to provide a holistic solution for order handling.

Real-world Examples: Search or eCommerce order

- For example, a "Search" domain service may encapsulate all functionalities related to keyword matching, filters, and sorting. It could be made up of multiple microservices, each responsible for one part of the search experience, such as text analysis, query optimization, or results ranking.

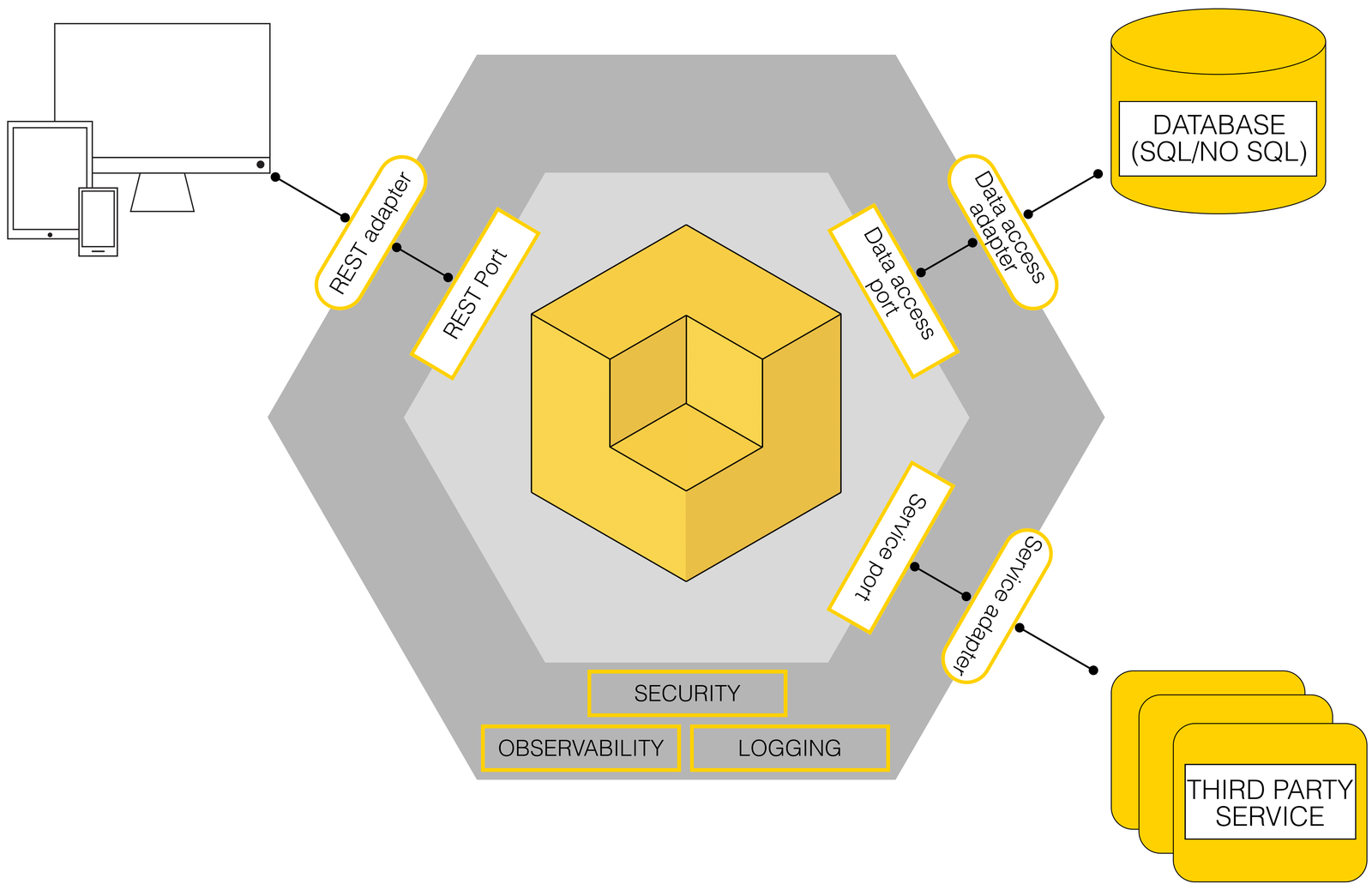
- Similarly, an "Order" domain service could encapsulate functionalities like payment validation, inventory checks, and order dispatch. This could involve multiple microservices that handle these responsibilities individually but collaborate to provide a unified order processing mechanism.

- By adhering to a domain driven design principle, we can create a flexible, scalable, and robust system architecture that aligns well with our complex business requirements while enabling efficient development, deployment, and maintenance.

Microservices Patterns

Hexagonal Architecture

A common design pattern for microservices architectures is the hexagonal architecture that is often deployed for the domain services. A Hexagonal Architecture, also known as the Ports and Adapters pattern, is a modern architectural approach designed to create flexible, maintainable, and easily testable software systems. The idea is to isolate the core business logic of an application from external concerns like user interfaces, databases, and other systems. This way, changes in one area have minimal impact on others, making the system more resilient, scalable and adaptable.



Examples of Hexagonal architectures tend to include more complex applications where the added layers of abstraction allow making significant changes in underlying components while leaving other dependent applications and services untouched since the service adapters remain the same. These would include applications such as commerce or finance.

Ports & Adapters

Imagine "ports" as entry and exit points for information in the application. These are like the doors and windows of a building, controlling what comes in and what goes out. They define the interactions that the application needs to have with the outside world, be it receiving data from a user interface or sending it to a database.

Adapters are like translators that convert external requests into a format that the application can understand, and vice versa. They link the ports to the actual technologies being used, such as a specific type of database or user interface. This means that if you decide to change from one database to another, only the adapter needs to be replaced or updated, not the entire system.

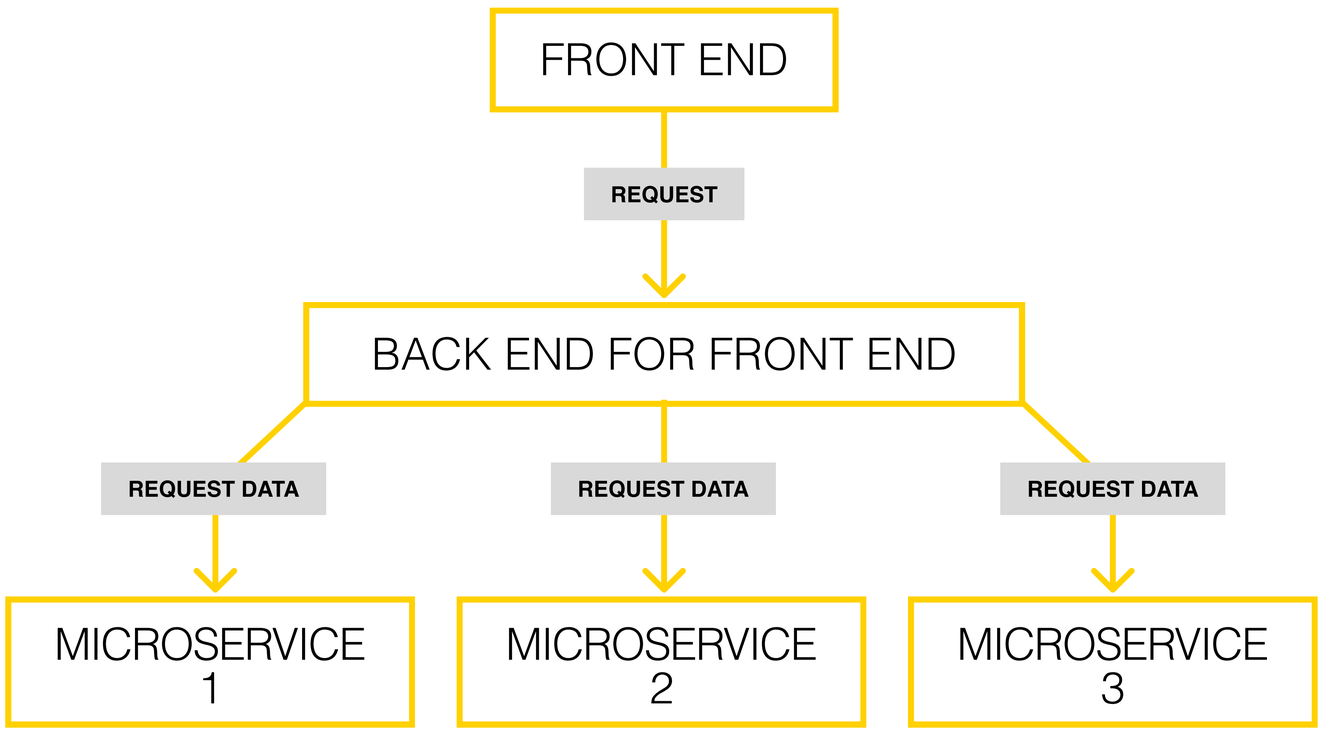
Data Access and Service Access

Data Access refers to how the application retrieves and stores information in databases. Service Access pertains to how the application communicates with external services like payment gateways or email services. In a hexagonal architecture, both are abstracted away from the core business logic, meaning they can be changed or updated independently without affecting the main functionalities of the system.

Backend for Frontend and the Experience Layer

The concept of an Experience API or Backend For Frontend (BFF) represents a pivotal architectural pattern aimed at tailoring data and services to specific user experiences or clients, like mobile apps, web apps, or other consumer endpoints.

This approach addresses several challenges in modern application development, including data transformation, client-specific logic, and the removal of point-to-point integration.



Data Transformation

One of the primary roles of an Experience API/BFF is to transform domain data into a format that is readily consumable by its intended audience. Imagine a scenario where a common data source provides comprehensive information, but different client apps only require specific subsets or forms of this data. The BFF acts as an intermediary layer that takes the domain data and transforms it into the specific shape and format needed by each client. This streamlines data processing on the client side, making the application more efficient and user-friendly.

Common Data Source or Set of Data Sources

The BFF architecture is designed to work with a common data source or a set of data sources. This centralized approach simplifies data management by reducing redundancy and ensuring data consistency. Whether the underlying data is sourced from databases, third-party APIs, or other services, the BFF brings it all together, offering a unified layer for data access and transformation.

Removing Point-to-Point Integration

Traditional point-to-point integration makes individual services tightly coupled with each other, complicating scalability and maintainability. Any change in one service can lead to a cascade of required changes in others. By introducing an Experience API/BFF layer, this issue is mitigated. The BFF serves as a single point of interaction for client applications, translating their various needs into appropriate calls to backend services. This eliminates the necessity for each client to integrate directly with multiple backend services, thereby reducing complexity and coupling.

Decomposition Patterns

One of the primary goals of composable architectures when moving away from a monolithic architecture is decomposition - namely the task of breaking down applications into smaller chunks in order to enable domain-driven design and the patterns we have discussed above.

The two primary approaches are:

Decompose by Business Capability

This approach requires looking at various business capabilities and grouping them along functional lines based on what a company does - usually breaking down into tasks and functions.

Decompose by Subdomain

The approach breaks down the system into microservices that correspond to specific subdomains and is more suited for complex environments where specific functions may be reused in multiple contexts.

Which is better?

The “right” approach often depends on the corporate culture and how closely the engineering teams understand the core business requirements (or vice-versa - how the business understands the technology enablement within their organization).

Regardless of the approach taken, one of the key tasks is translation - if you choose to take a subdomain-based approach, you may need to help the business understand why this pattern was chosen - and vice-versa.

It may be difficult to understand the differences in the two approaches, so let us consider a retail B2C ecommerce system. A business capability-based approach might segment an application into various business capabilities:

- Product catalog
- Pricing module (with the ability to create discounts by product type, brand, date/time, etc.)
- Inventory management
- Shipping
- Customer information
- Order management
- Etc.

If the requirement is now to add B2B capabilities you might need additional functionality such as:

- Account management and permissions (including the ability to define who can administer the ability to make orders on behalf of their organization)
- B2B Pricing (you may have different tiers of distributors or shipping rates and custom pricing for each)
- Organization information (rolling up individual business units and individuals ordering)

In all those cases around B2B user management, pricing and order management these domains would interact and behave vastly differently than the similar function for a B2C customer and these existing business capability functions would need to be duplicated or heavily modified to accommodate those changes.

In this case, if you were implementing a B2C ecommerce system but wish to have the option of adding B2B functionality at a later date, a subdomain-based approach might be a better option to start with, since elements around customers and pricing such as roles could be broken down into smaller, reusable components and repurposed cleanly for the separate B2C and B2B use cases. It is added work up front, but would pay dividends down the road as the use case expands.

A subdomain-based approach tends to look across business capabilities in order to identify common patterns and services, but requires a skilled evaluation in order to determine if these services may be re-used in multiple contexts in order to justify that effort and approach. For smaller applications where the problem set is defined (and will not expand), a business capability-based approach may be sufficient.

Decompose by Business Capability

As the MACH architectural paradigm evolves, it brings forth more intricate yet robust patterns, emphasizing the need for a community-driven approach to navigate its complexities. The MACH Alliance is blazing the trail with this series of whitepapers defining concepts, principles and architectural blueprints and is a critical resource for professionals seeking to leverage Microservices, API-first, Cloud-native, and Headless technologies.

We encourage readers to dive deep into the evolving landscape of composable architectures, inviting contributions of thoughts and opinions to enrich the collective understanding and advancement within the MACH Alliance. This is a community and this is a clarion call to engage, learn, and contribute to a rapidly maturing field.

Credits

This whitepaper was developed by the MACH Alliance Interoperability Task Force:

Adam Peter Nielsen
Chris Bach
Daniele Stroppa
Dom Selvon
Filip Rakowski
Mark Demeny
Melanie Richards
Roberto Carrera
Subhasri Vadyar

MACH ALLIANCE

Stay updated with the latest news and content from the MACH Alliance.
Sign up for our newsletter here

e. info@machalliance.org
w. www.machalliance.org