Fundamentals of Software Architecture

Software Architecture Introduction

Architecture Styles (cont.)

By: Võ Văn Hải

Email: vovanhai@ueh.edu.vn

1

Chapter 16

Orchestration-Driven Service-Oriented Architecture

2

Orchestration-Driven Service-Oriented Architecture

History and Philosophy

- This service-oriented architecture style appeared just as companies became enterprises in the late 1990s: merging with smaller companies, growing at a breakneck pace, and requiring more sophisticated IT to accommodate this growth. However, computing resources were scarce, precious, and commercial. Distributed computing had just become possible and necessary, and many companies needed variable scalability and other beneficial characteristics.
- Many external drivers forced architects in this era toward distributed architectures with significant constraints. Thus, architects were expected to reuse as much as possible.
- This style of architecture also exemplifies how far architects can push the idea of technical partitioning, which had good motivations but bad consequences.

3

Orchestration-Driven Service-Oriented Architecture

Topology

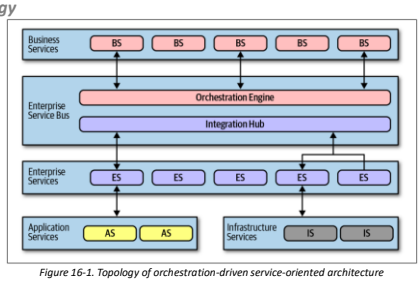


Figure 16-1. Topology of orchestration-driven service-oriented architecture

- Not all examples of this style of architecture had the exact layers illustrated in above figure, but they all followed the same idea of establishing a taxonomy of services within the architecture, each layer with a specific responsibility

4

Orchestration-Driven Service-Oriented Architecture

Taxonomy (1/5)

- The architect's driving philosophy in this architecture centered around enterprise-level reuse. Many large companies were annoyed at how much they had to continue to rewrite software, and they struck on a strategy to solve that problem gradually. Each layer of the taxonomy supported this goal.
- Business Services
  - Business services sit at the top of this architecture and provide the entry point. For example, services like ExecuteTrade or PlaceOrder represent domain behavior. One litmus test common at the time—could an architect answer affirmatively to the question "Are we in the business of…" for each of these services?
  - These service definitions contained no code—just input, output, and sometimes schema information. Business users usually define them, hence the name business services.

Orchestration-Driven Service-Oriented Architecture

Taxonomy (2/5)

- Enterprise Services
  - The enterprise services contain fine-grained, shared implementations. Typically, a team of developers is tasked with building atomic behavior around particular business domains: CreateCustomer, CalculateQuote, etc. These services are the building blocks that make up the coarse-grained business services, tied together via the orchestration engine.
  - This separation of responsibility flows from the reuse goal in this architecture. If developers can build fine-grained enterprise services at just the correct level of granularity, the business won't have to rewrite that part of the business workflow again. Gradually, the business will build up a collection of reusable assets through reusable enterprise services.
  - Unfortunately, the dynamic nature of reality defies these attempts. Business components aren't like construction materials, where solutions last decades. Markets, technology changes, engineering practices, and other factors confound attempts to impose stability on the software world.

Orchestration-Driven Service-Oriented Architecture

Taxonomy (3/5)

- Application Services
  - Not all services in the architecture require the same level of granularity or reuse as the enterprise services. Application services are one-off, single-implementation services. For example, perhaps one application needs geo-location, but the organization doesn't want to take the time or effort to make that a reusable service. An application service, typically owned by a single application team, solves these problems.
- Infrastructure Services
  - Infrastructure services supply the operational concerns, such as monitoring, logging, authentication, and authorization. These services tend to be concrete implementations owned by a shared infrastructure team that works closely with operations.

Orchestration-Driven Service-Oriented Architecture

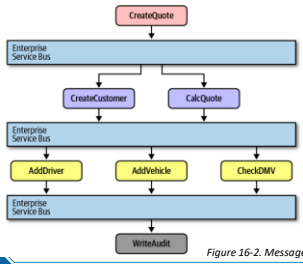
Taxonomy (4/5)

- Orchestration Engine
  - The orchestration engine forms the heart of this distributed architecture, stitching together the business service implementations using orchestration, including features like transactional coordination and message transformation. This architecture is typically tied to a single relational database, or a few, rather than a database per service as in microservices architectures. Thus, transactional behavior is handled declaratively in the orchestration engine rather than in the database.
  - The orchestration engine defines the relationship between the business and enterprise services, how they map together, and where transaction boundaries lie. It also acts as an integration hub, allowing architects to integrate custom code with package and legacy software systems.

Orchestration-Driven Service-Oriented Architecture

Taxonomy (5/5)

- Message Flow
  - All requests go through the orchestration engine—it is the location within this architecture where logic resides. Thus, message flow goes through the engine even for internal calls



The CreateQuote business-level service calls the service bus, which defines the workflow that consists of calls to CreateCustomer and CalculateQuote, each of which also has calls to application services. The service bus acts as the intermediary for all calls within this architecture, serving as both an integration hub and orchestration engine.

Figure 16-2. Message flow with service-oriented architecture

9

Orchestration-Driven Service-Oriented Architecture

Reuse…and Coupling

- A major goal of this architecture is reuse at the service level—the ability to gradually build business behavior that can be incrementally reused over time. Architects in this architecture were instructed to find reuse opportunities as aggressively as possible.

Figure 16-3. Seeking reuse opportunities in service-oriented architecture

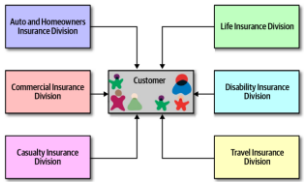


Figure 16-4. Building canonical representations in service-oriented architecture

10

Orchestration-Driven Service-Oriented Architecture

Architecture Characteristics Ratings

- Many of the modern criteria we use to evaluate architecture now were not priorities when this architecture was popular. In fact, the Agile software movement had just started and had not penetrated into the size of organizations likely to use this architecture.

Architecture characteristic	Star rating
Partitioning type	Technical
Number of quanta	1
Deployability	★
Elasticity	★★★
Evolutionary	★
Fault tolerance	★★★
Modularity	★★★
Overall cost	★
Performance	★★
Reliability	★★
Scalability	★★★★
Simplicity	★
Testability	★

11

Chapter 17
Microservices Architecture

12

Microservices Architecture

Introduction

- Most architecture styles are named after the fact by architects who notice a particular pattern that keeps reappearing.
- Microservices differs in this regard—it was named fairly early in its usage and popularized by a famous blog entry by Martin Fowler and James Lewis, "Microservices," published in March 2014.
- They delineated many common characteristics in this relatively new architectural style. Their blog post helped define the architecture for curious architects and helped them understand the underlying philosophy.
- Microservices is heavily inspired by the ideas in domain-driven design (DDD), a logical design process for software projects. One concept in particular from DDD, *bounded context*, decidedly inspired microservices. The concept of bounded context represents a decoupling style. When a developer defines a domain, that domain includes many entities and behaviors, identified in artifacts such as code and database schemas.

13

Microservices Architecture

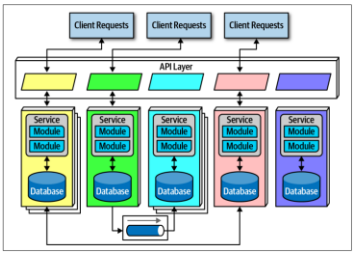
Introduction (cont.)

- In a traditional monolithic architecture, developers would share many of these concepts, building reusable classes and linked databases. Within a bounded context, the internal parts, such as code and data schemas, are coupled together to produce work; but they are never coupled to anything outside the bounded context, such as a database or class definition from another bounded context. This allows each context to define only what it needs rather than accommodating other constituents.
- Trade-off: re-use and coupling
  - When an architect designs a system that favors reuse, they also favor coupling to achieve that reuse, either by inheritance or composition.
  - However, if the architect's goal requires high degrees of decoupling, then they favor duplication over reuse. The primary goal of microservices is high decoupling, physically modeling the logical notion of bounded context.

14

Microservices Architecture

Topology



- Due to its single-purpose nature, the service size in microservices is much smaller than that of other distributed architectures.
- Architects expect each service to include all necessary parts to operate independently, including databases and other dependent components.

15

Microservices Architecture

Distributed

- Microservices form a distributed architecture:
  - each service runs in its own process, initially implying a physical computer but quickly evolving to virtual machines and containers.
  - Decoupling the services to this degree allows for a simple solution to a common problem in architectures that heavily feature multitenant infrastructure for hosting applications.
    - For example, when using an application server to manage multiple running applications, it allows operational reuse of network bandwidth, memory, disk space, and a host of other benefits. However, if all the supported applications continue to grow, eventually, some resources become constrained on the shared infrastructure. Another problem concerns improper isolation between shared applications.
- Separating each service into its own process solves all the problems brought on by sharing. Before the evolutionary development of freely available open-source operating systems, combined with automated machine provisioning, it was impractical for each domain to have its own infrastructure. However, with cloud resources and container technology, teams can reap the benefits of extreme decoupling, both at the domain and operational level.

16

Microservices Architecture

Bounded Context

- The driving philosophy of microservices is the notion of bounded context: each service models a domain or workflow. Thus, each service includes everything necessary to operate within the application, including classes, other subcomponents, and database schemas. This philosophy drives many of the decisions architects make within this architecture.
- Microservices take the concept of a domain-partitioned architecture to the extreme. Each service is meant to represent a domain or subdomain; in many ways, microservices is the physical embodiment of the logical concepts in domain-driven design.

17

Microservices Architecture

Bounded Context – Granularity (1)

The term "microservice" is a *label*, not a *description*. —Martin Fowler

- Architects struggle to find the correct granularity for services in microservices and often make the mistake of *making their services too small*, which requires them to build communication links back between the services to do useful work.
- The purpose of service boundaries in microservices is to capture a domain or workflow. In some applications, those natural boundaries might be large for some parts of the system—some business processes are more coupled than others.
- Some guidelines architects can use to help find the appropriate boundaries:
  - Purpose
  - Transactions
  - Choreography

18

Microservices Architecture

Bounded Context – Granularity (2)

- **Purpose**
  - The most obvious boundary relies on the inspiration for the architecture style, a domain. Ideally, each microservice should be extremely functionally cohesive, contributing one significant behavior on behalf of the overall application.
- **Transactions**
  - Bounded contexts are business workflows; often, the entities that need to cooperate in a transaction show architects a good service boundary. Transactions cause issues in distributed architectures, and they generate better designs if architects can design their systems to avoid them.
- **Choreography**
  - If an architect builds a set of services that offer excellent domain isolation yet require extensive communication to function, the architect may consider bundling these services back into a larger service to avoid the communication overhead.
- *Iteration* is the only way to ensure good service design. Architects rarely discover the perfect granularity, data dependencies, and communication styles on their first pass. However, after iterating over the options, an architect has a good chance of refining their design.

19

Microservices Architecture

Bounded Context - Data Isolation

- Another requirement of microservices, driven by the bounded context concept, is data isolation. Many architecture styles use a single database for persistence. However, microservices tries to avoid all kinds of coupling, including shared schemas and databases used as integration points.
- Data isolation is another factor an architect must consider when looking at service granularity. Architects must be wary of the *entity trap* and not simply model their services to resemble single entities in a database.
- Architects are accustomed to using relational databases to unify values within a system, creating a single source of truth, which is no longer an option when distributing data across the architecture. Thus, architects must decide how they want to handle this problem: either identifying one domain as the source of truth for some fact and coordinating with it to retrieve values or using database replication or caching to distribute information.

20

Entity-Trap Anti-Pattern

- Creating entity managers to perform CRUD operations is not architectural thinking, and the result is not architecture.
- These operations can be easily handled by frameworks such as
  - Ruby on Rails,
  - NakedObjects, or
  - Apache Isis (Causeway)
- Mapping from a database to a server doesn't need a full-blown architecture. Incorrectly identifying database relationships as workflows is called an entity trap, and it's an anti-pattern.

https://github.com/NakedObjectsGroup/NakedObjectsFramework

http://isis.apache.org/

21

Microservices Architecture

API layer

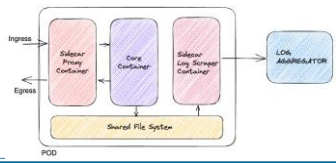
- Most pictures of microservices include an API layer sitting between the consumers of the system (either user interfaces or calls from other systems), but it is optional.
- It is common because it offers a good location within the architecture to perform useful tasks via indirection as a proxy or a tie into operational facilities, such as a naming service.
- While an API layer may be used for a variety of things, it should not be used as a mediator or orchestration tool if the architect wants to stay true to the underlying philosophy of this architecture: all interesting logic in this architecture should occur inside a bounded context, and putting orchestration or other logic in a mediator violates that rule.
- This also illustrates the difference between technical and domain partitioning in architecture: architects typically use mediators in technically partitioned architectures, whereas microservices is firmly domain partitioned.

22

Microservices Architecture

Operational Reuse (1/2)

- The Sidecar pattern is a design pattern that focuses on extending the functionality of a primary service by attaching an auxiliary container, known as a "sidecar" to it.
- This sidecar container runs in the same context as the main service and enhances its capabilities without requiring any modifications to the service code.
- It provides support functionalities, such as logging, monitoring, service discovery, circuit breakers, security, and more.

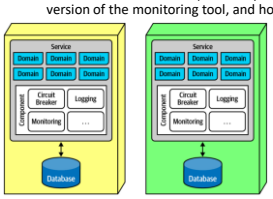


23

Microservices Architecture

Operational Reuse (2/2)

- Once a team has built several microservices, they realize that each has common elements that benefit from similarity.
  - For example, if an organization allows each service team to implement monitoring themselves, how can they ensure that each team does so?
  - And how do they handle concerns like upgrades?
  - Does it become the responsibility of each team to handle upgrading to the new version of the monitoring tool, and how long will that take?



The common operational concerns appear within each service as a separate component, which can be owned by either individual teams or a shared infrastructure team.
The sidecar component handles all the operational concerns that teams benefit from coupling together. Thus, when it comes time to upgrade the monitoring tool, the shared infrastructure team can update the sidecar, and each microservices receives that new functionality

Figure 17-2. The sidecar pattern in microservices

24

Microservices Architecture

Frontends

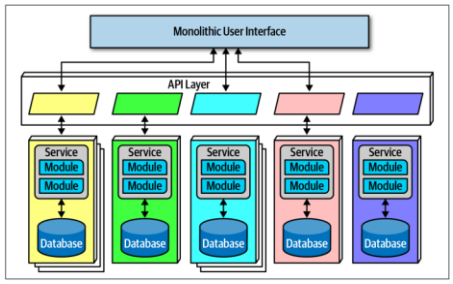


Figure 17-5. Microservices architecture with a monolithic user interface

25

Microservices Architecture

Frontends

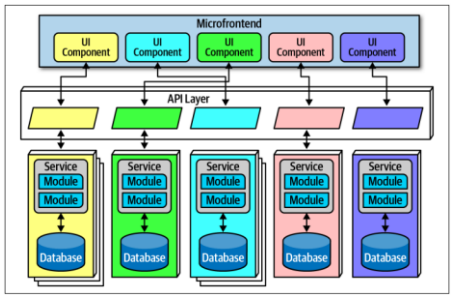


Figure 17-6. Microfrontend pattern in microservices

26

Microservices Architecture

Communication

- In microservices, architects and developers struggle with appropriate granularity, which affects both data isolation and communication. Finding the correct communication style helps teams keep services decoupled yet still coordinated in useful ways.
- Fundamentally, architects must decide on synchronous or asynchronous communication. Synchronous communication requires the caller to wait for a response from the callee. Microservices architectures typically utilize protocol-aware heterogeneous inter-operability.
- Protocol-aware
- Because microservices usually don't include a centralized integration hub to avoid operational coupling, each service should know how to call other services. Thus, architects commonly standardize on how particular services call each other: a certain level of REST, message queues, and so on. That means that services must know (or discover) which protocol to use to call other services.
- Heterogeneous
- Because microservices is a distributed architecture, each service may be written in a different technology stack. Heterogeneous suggests that microservices fully supports polyglot environments, where different services use different platforms.
- Interoperability
- Describes services calling one another. While architects in microservices try to discourage transactional method calls, services commonly call other services via the network to collaborate and send/receive information.

27

Microservices Architecture

Communication - Choreography and Orchestration

- Choreography utilizes the same communication style as a broker event-driven architecture. In other words, no central coordinator exists in this architecture, respecting the bounded context philosophy. Thus, architects find it natural to implement decoupled events between services.
- Domain/architecture isomorphism is one key characteristic that architects should look for when assessing how appropriate an architecture style is for a particular problem.
- This term describes how the shape of an architecture maps to a particular architecture style.
  - For example, in Figure 8-7, the Silicon Sandwiches' technically partitioned architecture structurally supports customizability, and the microkernel architecture style offers the same general structure. Therefore, problems that require a high degree of customization become easier to implement in a microkernel.
- Similarly, because the architect's goal in a microservices architecture favors decoupling, the shape of microservices resembles the broker EDA, making these two patterns symbiotic.
- In choreography, each service calls other services as needed, without a central mediator.

28

Microservices Architecture

Communication - Choreography and Orchestration

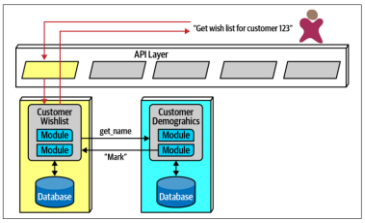


Figure 17-8. Using orchestration in microservices

In Figure 17-7, the user requests details about a user's wish list. Because the Customer *WishList* service doesn't contain all the necessary information, it makes a call to *CustomerDemographics* to retrieve the missing information, returning the result to the user.

29

Microservices Architecture

Communication - Choreography and Orchestration

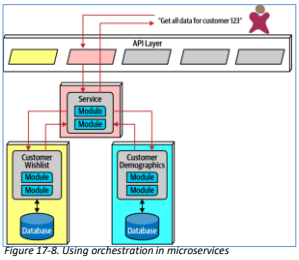


Figure 17-8. Using orchestration in microservices

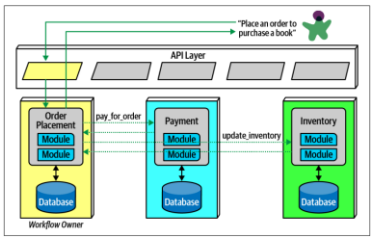
Because microservices architectures don't include a global mediator like other service-oriented architectures, if an architect needs to coordinate across several services, they can create their own localized mediator, as shown in Figure 17-8.

In Figure 17-8, the developers create a service whose sole responsibility is coordinating the call to get all information for a particular customer. The user calls the Report *CustomerInformation* mediator, which calls the necessary other services.

30

Microservices Architecture

Communication - Choreography and Orchestration



In Figure 17-9, the first service called must coordinate across a wide variety of other services, basically acting as a mediator in addition to its other domain responsibilities. This pattern is called the front controller pattern, where a nominally choreographed service becomes a more complex mediator for some problem. The downside to this pattern is added complexity in the service.

- Trade-offs: In choreography, the architect preserves the highly decoupled philosophy of the architecture style, thus reaping maximum benefits touted by the style. However, common problems like error handling and coordination become more complex in choreographed environments.

31

Microservices Architecture

Communication - Choreography and Orchestration

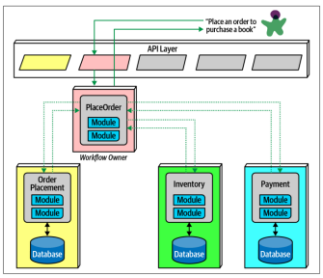


Figure 17-10. Using orchestration for a complex business process

Alternatively, an architect may choose to use orchestration for complex business processes.

In Figure 17-10, the architect builds a mediator to handle the complexity and coordination required for the business workflow. While this creates coupling between these services, it allows the architect to focus coordination into a single service, leaving the others less affected. Often, domain workflows are inherently coupled—the architect's job entails finding the best way to represent that coupling in ways that support both the domain and architectural goals.

32

Microservices Architecture

Communication - Transactions and Sagas

- Architects aspire to extreme decoupling in microservices, but then often encounter the problem of how to do transactional coordination across services. Because the decoupling in the architecture encourages the same level for the databases, atomicity that was trivial in monolithic applications becomes a problem in distributed ones.
- Building transactions across service boundaries violates the core decoupling principle of the microservices architecture (and also creates the worst kind of dynamic connascence, connascence of value). The best advice for architects who want to do transactions across services is: don't! Fix the granularity components instead. Often, architects who build microservices architectures who then find a need to wire them together with transactions have gone too granular in their design. Transaction boundaries is one of the common indicators of service granularity.

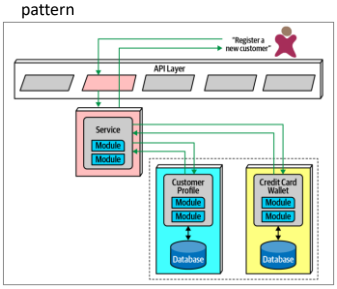
Don't do transactions in microservices—fix granularity instead!

33

Microservices Architecture

Communication - Transactions and Sagas

- A popular distributed transactional pattern in microservices is the *saga* pattern



A service acts a mediator across multiple service calls and coordinates the transaction.
The mediator calls each part of the transaction, records success or failure, and coordinates results.
If everything goes as planned, all the values in the services and their contained databases update synchronously.

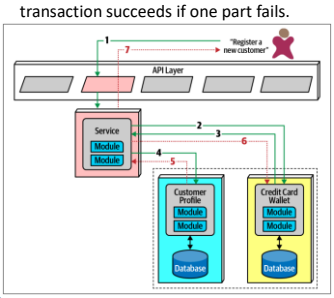
Figure 17-11. The saga pattern in microservices architecture

34

Microservices Architecture

Communication - Transactions and Sagas

- In an error condition, the mediator must ensure that no part of the transaction succeeds if one part fails.



If the first part of the transaction succeeds, yet the second part fails, the mediator must send a request to all the parts of the transaction that were successful and tell them to undo the previous request.
This style of transactional coordination is called a *compensating transaction framework*.

Figure 17-12. Saga pattern compensating transactions for error conditions

35

Microservices Architecture

Communication - Transactions and Sagas

- Developers implement this pattern by usually having each request from the mediator enter a pending state until the mediator indicates overall success. However, this design becomes complex if asynchronous requests must be juggled, especially if new requests appear that are contingent on pending transactional state. This also creates a lot of coordination traffic at the network level.
- Another implementation of a compensating transaction framework has developers build do and undo for each potentially transactional operation. This allows less coordination during transactions, but the undo operations tend to be significantly more complex than the do operations, more than doubling the design, implementation, and debugging work.
- While it is possible for architects to build transactional behavior across services, it goes against the reason for choosing the microservices pattern. Exceptions always exist, so the best advice for architects is to use the saga pattern sparingly.

A few transactions across services are sometimes necessary; if it's the dominant feature of the architecture, mistakes are made!

36

Microservices Architecture

Architecture Characteristics Ratings

- With microservice, the notable is the high support for modern engineering practices such as automated deployment, testability, and others not listed. Microservices couldn't exist without the ***DevOps*** revolution and the relentless march toward automating operational concerns.

Architecture characteristic	Star rating
Partitioning type	Domain
Number of quanta	1 to many
Deployability	★★★★
Elasticity	★★★★★
Evolutionary	★★★★★
Fault tolerance	★★★★
Modularity	★★★★★
Overall cost	★
Performance	★★
Reliability	★★★★
Scalability	★★★★★
Simplicity	★
Testability	★★★★

37

Microservices Architecture

Building a Microservice Architecture steps (1/2)

- Step 1: Design the Microservices
  - Identify the functionalities of your application that can be broken down into separate, independent services. Each microservice should have a single responsibility and be loosely coupled from other microservices.
  - Choose the appropriate frameworks for building microservices in Java, such as Spring Boot, Micronaut, or Vert.x. These frameworks provide tools and features for building scalable and resilient microservices.
  - Define the APIs for each microservice, including the input parameters, output responses, and error handling mechanisms. Use RESTful or event-driven approaches for communication between microservices.
- Step 2: Set up Communication between Microservices
  - Decide on the communication protocols and patterns between microservices. Common options include HTTP/REST, gRPC, and message brokers like Apache Kafka or RabbitMQ.
  - Implement communication mechanisms using the chosen frameworks and libraries. For example, you can use Spring Cloud, Micronaut Discovery and/or Eureka for service discovery, and Spring Cloud Gateway or Netflix Zuul for API gateway functionality.
  - Implement error handling and fault tolerance mechanisms, such as circuit breakers, retries, and fallbacks, to ensure resilience and reliability in microservice communication.

38

Microservices Architecture

Building a Microservice Architecture steps (2/2)

- Step 3: Handle Data Storage
  - Decide on the data storage strategy for each microservice. Options include using separate databases for each microservice or a shared database with appropriate access controls.
  - Implement data storage using appropriate technologies, such as relational databases like MySQL or PostgreSQL, NoSQL databases like MongoDB or Cassandra, or in-memory data stores like Redis or Hazelcast.
  - Use appropriate frameworks and libraries for data access, such as JPA or Spring Data, to interact with databases from Java microservices.
- Step 4: Manage Deployment
  - Choose a deployment strategy for your microservices, such as containerization with Docker or virtualization with virtual machines. Containerization is commonly used in microservice architectures due to its flexibility and scalability.
  - Use container orchestration tools like Kubernetes or Docker Swarm to manage the deployment, scaling, and monitoring of microservices.
  - Implement logging, monitoring, and tracing mechanisms to ensure observability and troubleshoot microservices in production.
- Step 5: Test and Deploy Microservices
  - Write unit tests, integration tests, and end-to-end tests for each microservice to ensure its functionality and reliability.
  - Use continuous integration and continuous deployment (CI/CD) practices to automatically build, test, and deploy microservices to production environments.
  - Monitor and optimize the performance and reliability of microservices in production to ensure smooth operation.

39

Microservices Architecture

Microservices Frameworks for Java

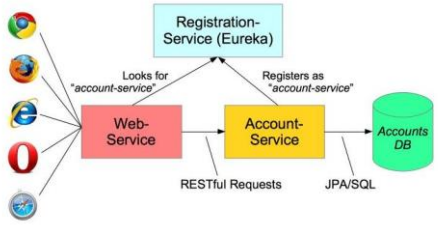
- **Spring Boot**. This is probably the best Java microservices framework that works on top of languages for Inversion of Control, Aspect-Oriented Programming, and others.
- **Quarkus**. This is a lightweight Java framework optimized for GraalVM and HotSpot, and it's tightly integrated with Kubernetes.
- **Micronaut**. It's based on the Netty framework, which makes it well-suited for microservices that need to handle a lot of traffic.
- **Helidon**. A microservices framework developed by Oracle. It is based on the MicroProfile specification, making it interoperable with other MicroProfile-based frameworks.
- **AxonIQ**. A microservices framework designed for event-driven architectures. It provides a comprehensive toolkit for building event-driven microservices, including a message bus, event sourcing, and CQRS (Command Query Responsibility Segregation).

40

Microservices Architecture

Example

- https://spring.io/blog/2015/07/14/microservices-with-spring
  - Git: https://github.com/paulc4/microservices-demo



41

42