6 More Microservices Interview Questions

BYTEBYTEGO
DEC 21, 2023 · PAID

In this issue, we continue our exploration of microservices interview questions. We will cover the following topics:

1. What is an API Gateway?
2. What are the differences between REST and RPC?
3. What is a configuration manager?
4. What are common microservices fault tolerance approaches?
5. How do we manage distributed transactions?
6. How do we choose between monolithic and microservices architectures?

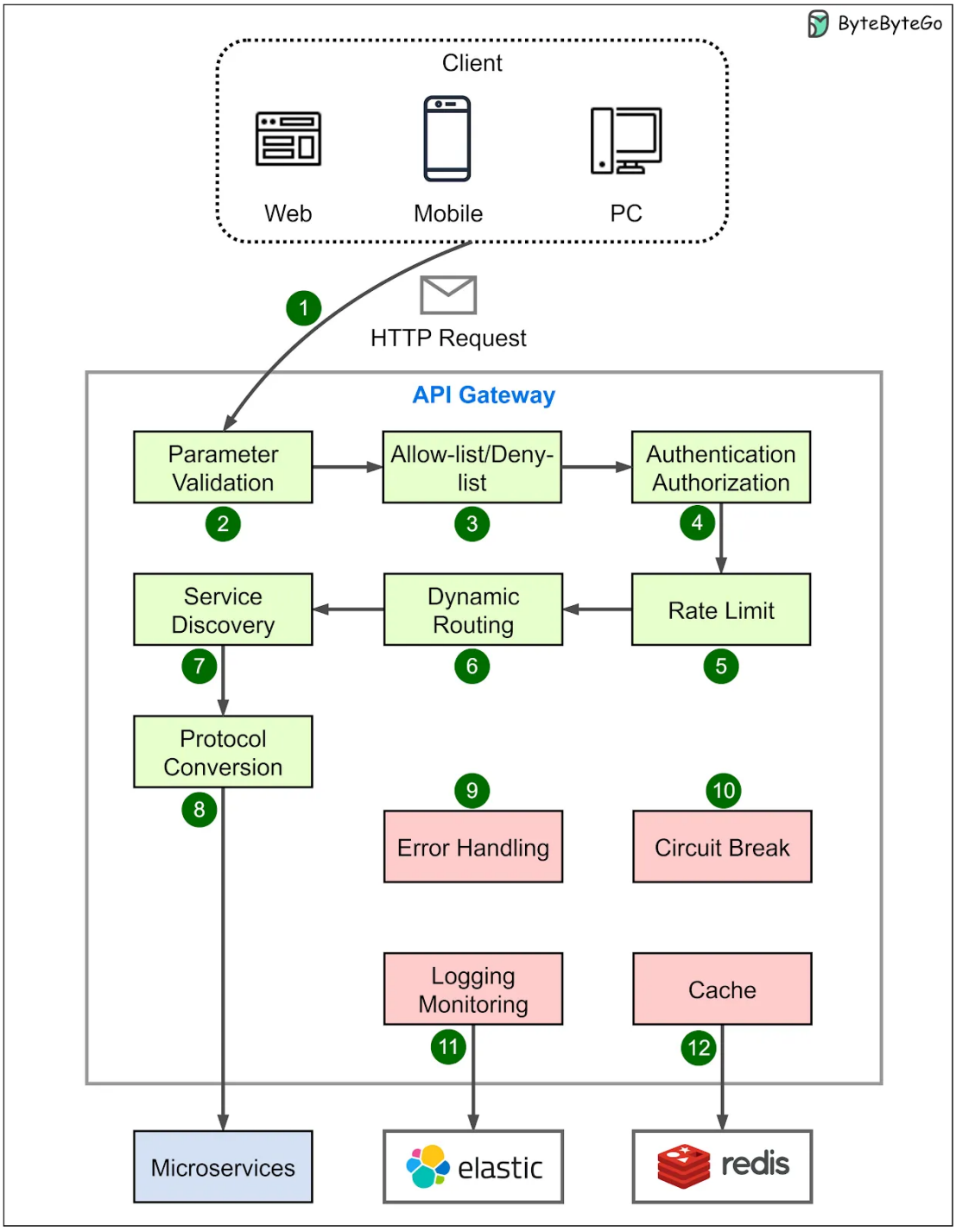
You can find more information on the definition of microservices and their basic components in our previous article.

We will now dive into the API gateway, an important request entry point in a microservices architecture.

1. What is API Gateway?

In a microservices architecture, an API gateway acts as a single entry point for client requests. The API gateway is responsible for request routing, composition, and protocol translation. It also provides additional features like authentication, authorization, caching, and rate limiting.

The diagram below shows the key steps:



Step 1: The client sends an HTTP request to the API gateway.

Step 2: The API gateway parses and validates the attributes in the HTTP request.

Step 3: The API gateway checks allow/deny lists.

Step 4: The API gateway authenticates and authorizes through an identity provider.

Step 5: Rate-limiting rules are applied. Requests over the limit are rejected.

Steps 6 and 7: The API gateway routes the request to the relevant backend service by path matching.

Step 8: The API gateway transforms the request into the appropriate protocol and forwards it to backend microservices.

Step 9: The API gateway handles any errors that may arise during request processing for graceful degradation of service.

Step 10: The API gateway implements resiliency patterns like circuit brakes to detect failures and prevent overloading interconnected services, avoiding cascading failures.

Step 11: The API gateway utilizes observability tools like the ELK stack (Elastic-Logstash-Kibana) for logging, monitoring, tracing, and debugging.

Step 12: The API gateway can optionally cache responses to common requests to improve responsiveness.

Besides request routing, the API gateway can also aggregate responses from microservices into a single response for the client.

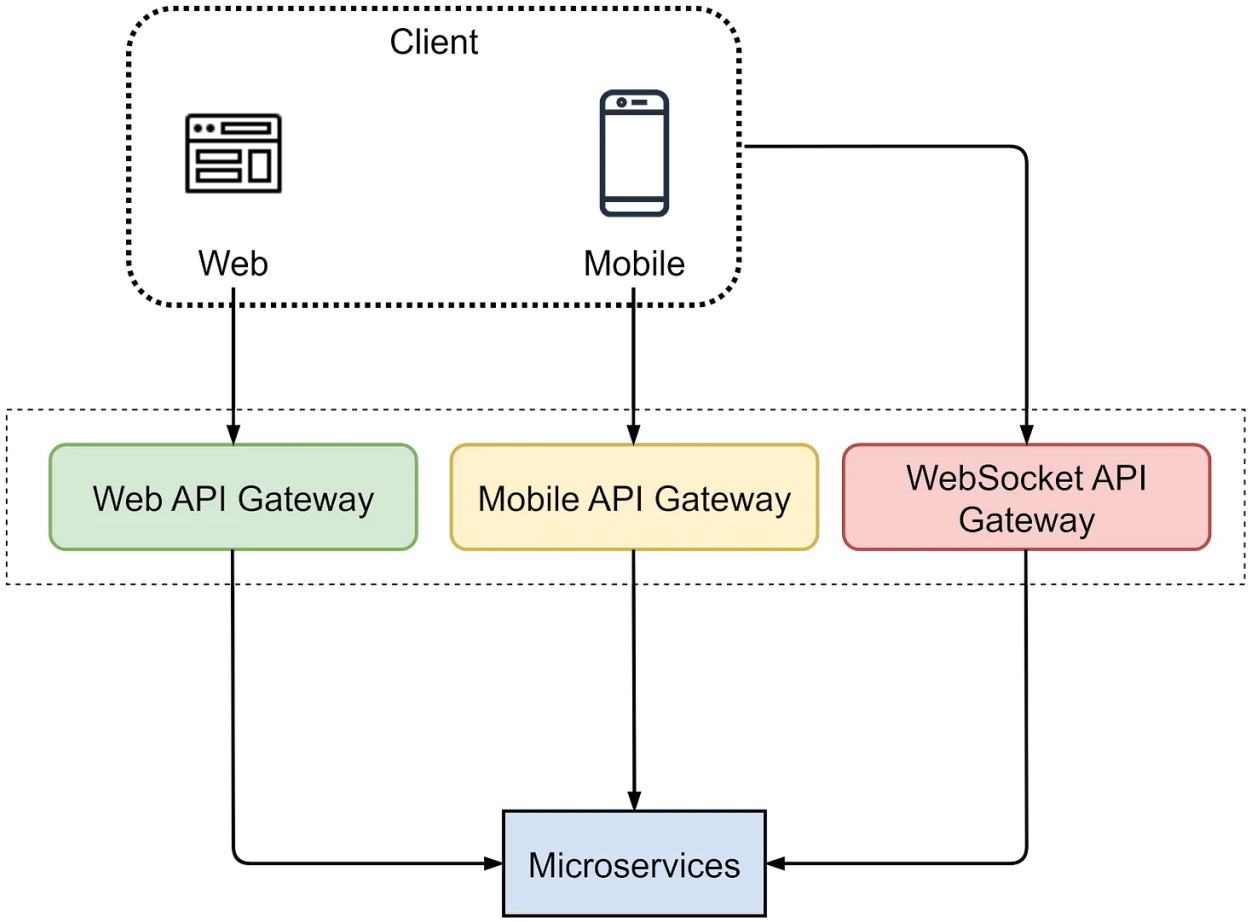
The API gateway is different from a load balancer. While both handle network traffic, the API gateway operates at the application layer, mainly handling HTTP requests; the load balancer mostly operates at the transport layer, dealing with TCP/UDP protocols. The API gateway offers more functions as it sees the request payload.

The API gateway differs from a load balancer in that it typically operates at the application layer to handle HTTP requests and understand message payloads, while traditional load balancers work at the transport layer to handle TCP/UDP connections without looking at the application data.

However, the lines can blur between these two types of infrastructure. Some advanced load balancers are gaining application layer visibility and routing capabilities resembling API gateways.

But in general, API gateways focus on application-level concerns like security, routing, composition, and resilience based on the payload, while traditional load balancers map requests to backend servers mainly based on transport-level metadata like IP and port numbers.

We often have separate API gateways tailored for different clients and their user experience requirements. The diagram below shows a typical architecture. We have different API gateways to handle requests from mobile devices and web applications because they have unique requirements for user experiences. Additionally, we separate WebSocket API Gateway because it has different connection persistence and rate-limiting requirements compared to HTTP gateways.

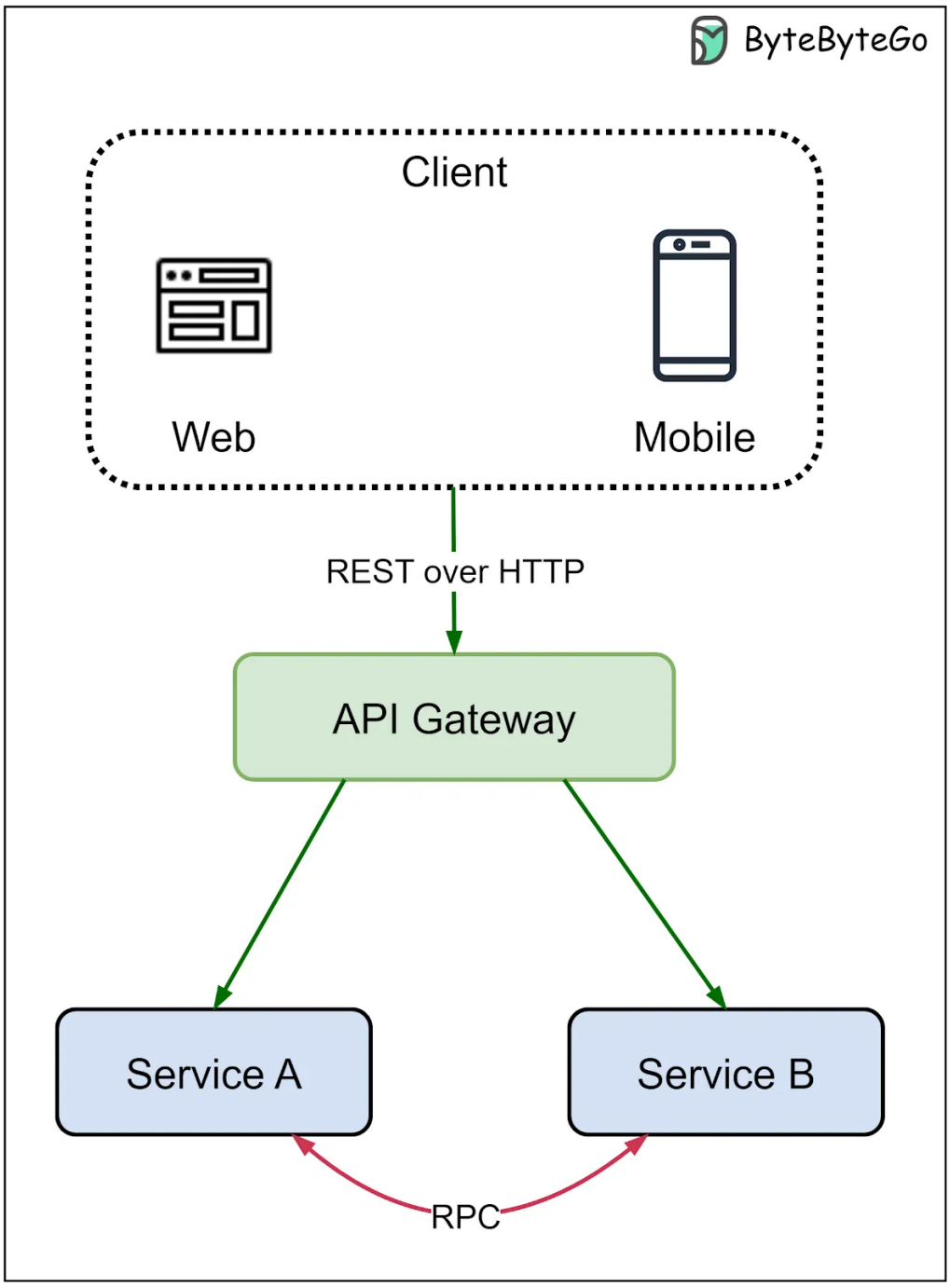


Some recent API gateway trends:

1. GraphQL support. GraphQL is a type system and a query language for APIs. Many API gateways now offer integration with GraphQL.
2. Service Mesh integration. Service meshes like Istio and Linkerd are used to handle communications among microservices. API gateways are integrating with them to enhance traffic management capabilities.
3. AI integration. API gateways are integrating with AI capabilities to enable smarter request routing or anomaly detection in traffic patterns.

2. What Are the Differences Between REST and RPC?

REST (Representational State Transfer) and RPC (Remote Procedure Call) are two common architectural patterns used for communications in distributed systems. REST is used for client-server communications, and RPC is used for server-server communications, as illustrated in the diagram below.



We summarize their differences in the table below.

1. REST is based on standard HTTP methods like GET, POST, PUT, and DELETE. RPC is based on faster protocols like HTTP/2.
2. REST commonly uses standard data formats such as JSON or XML for representing resources and their states. Data formats in RPC can be based on standard JSON or XML, or proprietary formats like ProtoBuf.
3. Resources are identified by URIs (Uniform Resource Identifiers) in REST, while in RPC, the data is usually used to represent a command or an action.
4. RPC is used in microservices communications and it is optimized for higher performance. For example, the payload is encoded and is compact. REST, on the other hand, is based on standard HTTP protocol, which is used for client-server communications.

	REST	RPC
Communication Protocol	Based on HTTP verbs (GET, POST, PUT, DELETE)	gRPC is based on HTTP/2
Data Schema	JSON, XML, HTML, plain text	JSON, XML, ProtoBuf, Thrift, FlatBuffers
API Style	Resource-oriented	Command- or action-oriented
Use Cases	Public APIs	Microservice communications
Performance	REST has lower performance due to its protocol and payload.	RPC has higher performance due to HTTP/2 and encoded compact payload.

3. What is a Configuration Manager?

In a microservices architecture, each service needs to maintain configurations for aspects like database connections, service ports, logging levels, etc. The number of services and configurations can be massive. These numerous configurations across potentially hundreds of services are centralized and maintained in a configuration

manager. Additionally, configurations often need to vary across environments. A configuration manager simplifies managing this configuration sprawl across environments compared to handling configurations on a per-service basis.

A configuration manager has the following key responsibilities:

1. Centralized configuration management so each service can retrieve configurations from the store. This avoids having to release new service versions just for configuration changes. Relying on a distributed key-value store is a common implementation.
2. Runtime configuration updates that allow services to reload updated configurations without restarting or re-deploying. This facilitates continuous delivery of config changes.
3. Configuration versioning and history tracking comparable to source code version control systems. This links code changes to related configurations, enabling the rollback of configurations alongside associated code when needed.
4. Access controls around configurations, especially sensitive credentials or settings. Role-based visibility rules allow appropriate teams to view or modify configurations related to their microservices. For example, only the SRE team is allowed to view production configurations or only the payments team is allowed to modify payment channel addresses.
5. Environment-specific configurations that distinguish between test, QA, and staging configurations from production. Deploying configurations can leverage similar workflows as code deployment pipelines.
6. Support multi-tenancy. Custom configuration partitioning in multi-tenant scenarios where different clients or customers require distinct configurations reflecting their preferences. For example, Merchants have different preferences on an eCommerce platform. Configuration isolation prevents conflict across clients.

A configuration manager itself needs to be highly available. If the configuration store has downtime or cannot respond to client services, those services will be unable to start or function properly. This could lead to widespread system outages. To mitigate this, configuration managers often utilize clustering for increased resilience from replication and redundancy across nodes.

Popular open-source configuration managers include HashiCorp Consul, etcd, and Apache ZooKeeper. Managed cloud solutions like Spring Cloud Config Server also exist.

Configuration managers also integrate with infrastructure like container orchestrators. For example, Kubernetes uses etcd as a backend database for service discovery and coordination between Kubernetes components. So etcd powers both centralized configuration management as well as underlying Kubernetes cluster management. This demonstrates how configuration managers enable multiple key roles - config store and orchestrator integration - in modern environments.

4. What are Common Microservices Fault Tolerance Approaches?

Building fault tolerance is crucial when designing microservices architectures. Netflix created a library called Hystrix to help provide latency and fault tolerance for microservices. It implements several common resilience patterns that remain relevant even as new tools have emerged.

Hystrix pioneered patterns like circuit breakers and fallbacks that are useful for creating robust microservices. Although Netflix now recommends Resilience4j as a more modern resilience library, the core approaches used in Hystrix still apply. Let's overview some of these key techniques:

Circuit breaker

The circuit breaker pattern monitors requests to services. If failure rates cross a threshold, the circuit "trips" to stop sending requests that are likely to fail. This prevents cascading failures across interconnected services.

Fallback

Fallback provides a default or downgraded response if a service call fails or times out. This creates a more graceful user experience during outages.

Request caching

Caching the results of repeat requests reduces duplicate calls to underlying services for the same data.

Request coalescing

Combines multiple concurrent requests for the same data into batches. This avoids making duplicate, simultaneous calls for the same resource. Instead, requests are grouped and sent as a single aggregated batch. This reduces load on the downstream services.

Bulkhead isolation

Bulkhead isolation sets resource usage limits to prevent failures from cascading across components and services. For example, thread pools can be configured with maximum concurrent execution limits to create isolation boundaries. If one component or service starts failing or using up too many resources, the bulkheads ensure failures and resource exhaustion are "contained" within that boundary instead of spreading widely across the system. This prevents full system outages.

Health monitoring

Tracking health metrics allows informing decisions about circuit breaker thresholds, bulkhead sizes, and other resilience parameters.

5. How do We Manage Distributed Transactions?

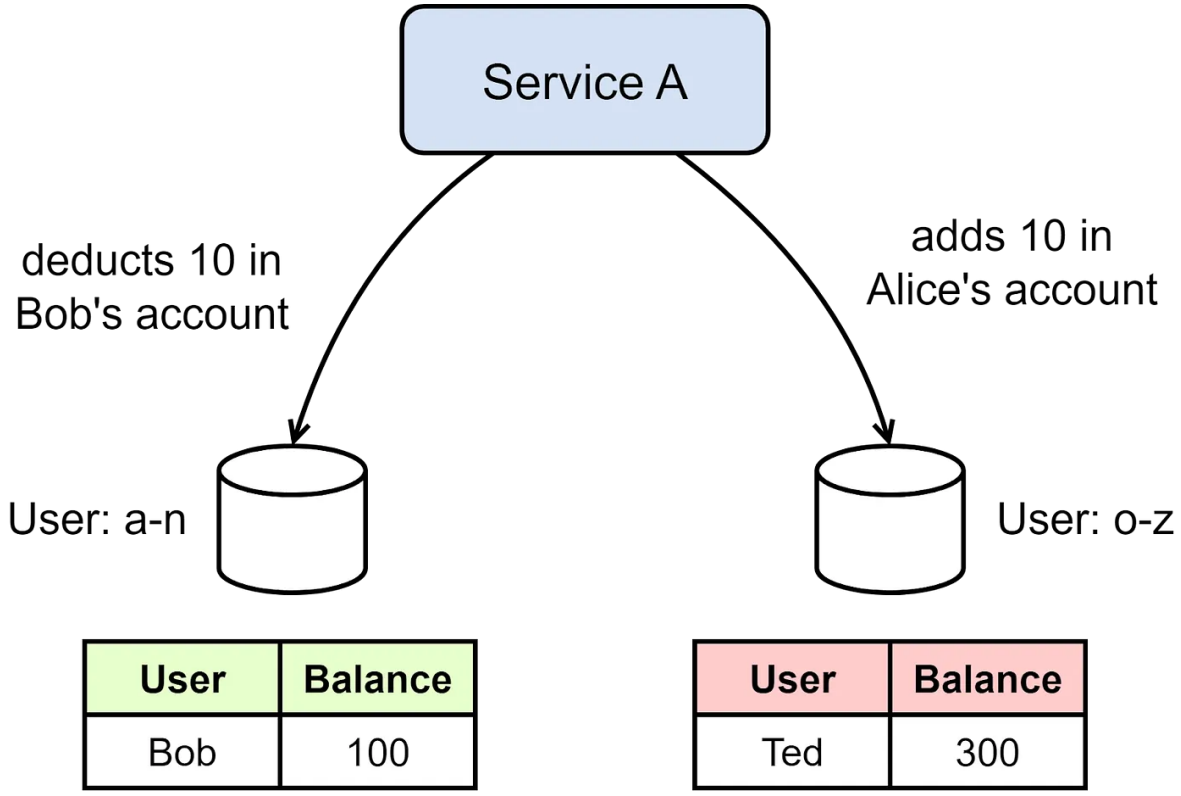
A transaction is a unit of work consisting of a sequence of operations executed as a single, indivisible, and atomic unit within a computing system. Transactions have ACID properties:

1. Atomicity - A sequence of operations is done in its entirety or not at all.
2. Consistency - The database remains in a consistent state after the transaction.
3. Isolation - Concurrent transactions are isolated from each other.
4. Durability - Data persists after a transaction is committed, even after system failures.

Transactions become “distributed” because databases become distributed to handle massive amounts of data. Distributed transactions aim to follow ACID properties, but achieving them introduces additional challenges compared to transactions on a centralized database. Let’s review the properties:

1. Atomicity - A distributed transaction may span multiple resources. If any part fails, all resources must be rolled back, which is complex to coordinate. A more relaxed protocol like 2PC (Two-Phase Commit) can be used in this scenario.
2. Consistency - In a distributed environment, this property means multiple resources need to maintain a consistent state with each other. If one resource fails or loses data, the overall system state becomes inconsistent. To deal with this, we often apply "eventual consistency" as a compromise. The idea is that even if some data is temporarily inconsistent across nodes, over time the nodes will "eventually" converge back to a consistent state. So we relax the absolute consistency guarantee in favor of the system as a whole reaching consistency eventually, accepting short periods of inconsistency to improve availability and partition tolerance.
3. Isolation - Concurrent transactions across nodes may cause resource conflicts. This is challenging in distributed environments. Optimistic locking or snapshot isolation can help achieve isolation.
4. Durability - Data must persist across enough distributed nodes to reconstruct the data even when some nodes fail. This links to consistency - the persistent data must also be kept consistent. Typically, data is replicated across multiple nodes for redundancy. Consensus algorithms like Raft or Paxos can then establish agreement between the nodes on the correct "golden copy" of the data in case replicas become inconsistent after failures.

The diagram below shows an example. Assume Bob wants to transfer $10 to Alice. Their accounts reside in separate databases, so a single database transaction cannot deduct from Bob’s account and increase Alice’s atomically. We need to coordinate the two operations to meet the requirements discussed earlier.



Managing distributed transactions is complex. Some frameworks and patterns help to address this challenge:

Try-Confirm-Cancel (TCC)

TCC is a distributed transaction pattern. It divides each transaction into Try, Confirm, and Cancel steps. The Try phase attempts to execute its local transaction. The Confirm phase commits it. The Cancel phase rolls back the transaction if Try or Confirm fails. TCC ensures consistency without relying on a central coordinator.

Two-Phase Commit (2PC)

2PC is a protocol that ensures atomicity and consistency of distributed transactions. Either all participants commit their changes or all roll back to their initial state. It has two phases - prepare and commit.

In the prepare phase, a centralized coordinator (transaction manager) sends a request asking each participant to prepare for the transaction. Participants reply with either a vote to commit or abort.

In the commit phase, if all participants voted to commit, the coordinator sends a "commit" message instructing them to make the changes permanent. If any voted to

abort or if a failure occurs, the coordinator sends a "rollback" message, rolling all nodes back to their state before the transaction in the prepare phase.

Saga

The Saga pattern handles distributed transactions without a central coordinator. Instead, it decomposes a long-running transaction into a sequence of smaller, independent transactions (sagas). Each saga manages its own local commit and rollback logic. This provides more flexible and scalable distributed transaction handling.

The Saga pattern handles distributed transactions without a central coordinator. Instead, it decomposes a long-running transaction into a sequence of smaller, independent transactions called sagas.

Each saga manages its own transactional boundaries and can complete independently. If a saga fails, it compensates by triggering a compensating transaction that counteracts its previous changes, keeping the system consistent.

For example, transferring $10 from Bob to Alice could involve these sagas:

1. Withdraw $10 from Bob's account - Compensate by depositing $10 back into Bob's account
2. Deposit $10 into Alice's account - Compensate by withdrawing $10 from Alice's account

Each saga manages its own transaction boundaries. If the deposit fails, the credit saga is compensated by withdrawing the accidental deposit. This keeps the system consistent.

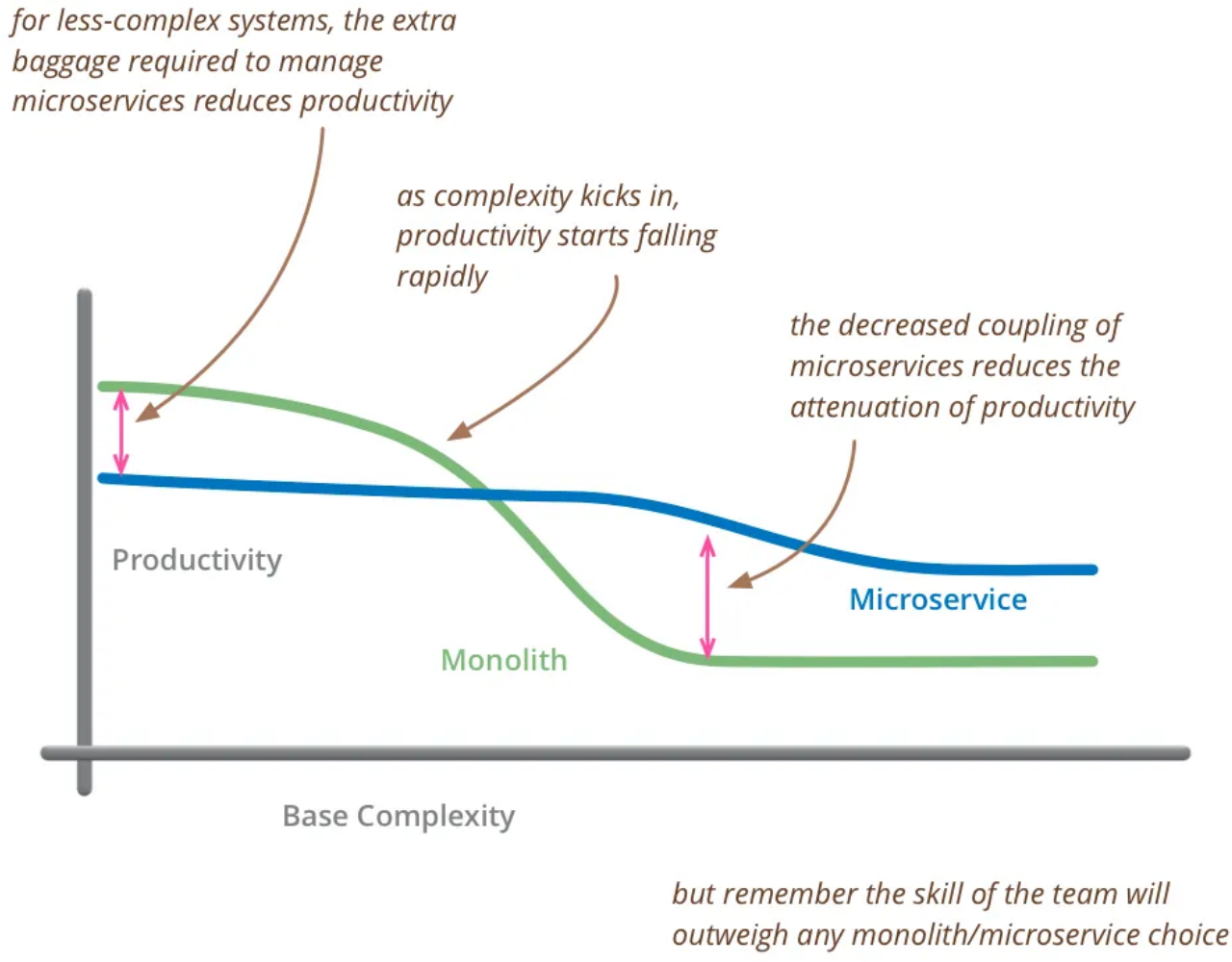
The key benefit of the Saga pattern is decentralization - no central coordinator is needed, enabling flexible and scalable distributed transactions. Participants only need to coordinate with their saga's predecessor and successor rather than a global coordinator.

The tradeoff is that sagas sacrifice atomicity in favor of availability. Changes are eventually consistent rather than perfectly coordinated.

6. How do we Choose Between Monolithic and Microservices Architectures?

The choice between a monolithic architecture and a microservices architecture is not always straightforward. There are advantages and disadvantages to each approach that should be carefully weighed based on the specific needs and context of a software system and development team.

As Martin Fowler's diagram belows illustrates, we generally recommend starting with a monolithic architecture when business needs are simple, the software system is not yet complex, and the development team is relatively small. As complexity increases over time, monolithic systems become harder to maintain and scale. At this inflection point, transitioning to a microservice architecture allows for better modularity, scalability, and technology flexibility. The microservices approach also enables faster feature velocity for large engineering teams working across multiple domains.



Source: martinfowler.com

Now we have covered 13 popular microservice interview questions across two issues. We hope this overview of common microservices interview topics helps prepare you to demonstrate your distributed systems design knowledge and have productive discussions exploring this architecture. Best of luck in your upcoming interviews!

124 Likes · 14 Restacks

3 Comments

Write a comment...

3 more comments...