26
Microservices Architecture

This chapter explains the microservices architecture and the benefits of using it to build your microservices-based database applications.

Topics:

26.1 About Microservices Architecture

A microservices architecture is an approach to developing applications as a collection of loosely coupled, autonomous services. A microservice in an application is a small, self-contained service with a limited contract. For example, a travel agency can implement a microservices application that provides airline, hotel, and car rental bookings as microservices.

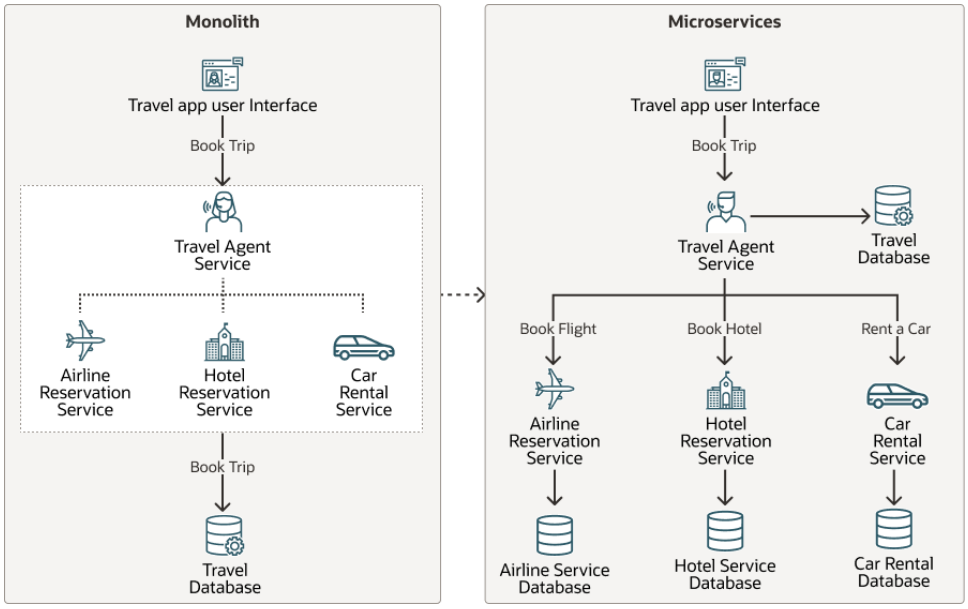
Each service in a microservices application implements a single business capability and communicates with other services through APIs or a messaging system. With loosely coupled services and encapsulated data, you can independently scale individual components, have a dedicated team for each service, and build your applications to achieve greater business agility and profits.

Why switch from a monolithic application to microservices?

- Increased complexity of the application architecture is making your application difficult to scale and manage.
- Changing or adding a single feature in a monolith requires you to change the code for the entire application, making it a time-consuming and expensive process.
- Your product is mature enough to scale up and your team has the necessary tools and skills to adopt microservices.

Monolith and Microservices illustrates a travel agency application as a monolithic application and as a micorservices-based application with flight reservation, hotel booking, and car rental services.

Figure 26-1 Monolith and Microservices



Benefits of Microservices

- Individual application components can scale independently. Applications operating within a cloud or hybrid environment can scale easily with independent development and deployment of services.
- Applications are easier to build, maintain, and deploy. Each service can be built and deployed independently, without affecting other services or rebuilding the entire application.
- Software development is parallelized. You have a team working on a separate codebase, making it easy to develop and test software, and adopt new technology.
- Applications can be built in multiple languages, like Java, .NET, or Python, and using various data types, such as JSON, Graph, and Spatial. Each service team can choose its own technology stack and tools because microservices communicate through language-agnostic APIs.
- Data ownership is decentralized. Each service has its own database, or there might be one database server but within that server each service has its own private schema and tables.

See Also:

- Learn About the Microservices Architecture (oracle.com)
- Differences Between Microservices and Monolithic Architectures

26.2 Features of Microservices Architecture

A microservices architecture must have these features:

Loosely-coupled Services

Microservices architecture requires breaking down your application into smaller component services. You can deploy each component service individually. Each self-contained service implements a core function and has clearly defined boundaries with data divided into bounded contexts. The database schema is restructured to identify which datasets each service needs.

Decentralized Data

Microservices ensure autonomy of services and teams. Services may not share the same data source or technology. For example, you could think of user registration and billing management as different teams, skilled in their respective areas, and working on a technology stack that is suited to their specific requirements and skillsets.

Independent Deployment

Independent services are developed and deployed in small, manageable units without affecting a large part of the codebase.

Automated Infrastructure

Microservices use automated infrastructure to operate and may require:

- An interservice communication system using asynchronous messaging and message brokers.
- A containerized system that allows you to focus on developing the services and lets the system handle the deployment and dependencies.

Reusable Systems

Developers can reuse code or reuse library functions (if using the same language and platform) in another feature or across services and teams.

Resilient Architecture

Microservices can withstand an entire application crash because if an independent service fails, you can expect to lose some part of the application functionality but other parts of the application can continue functioning.

26.3 Challenges in a Distributed System

To ensure data integrity and consistency, a transaction must have ACID (Atomicity, Consistency, Isolation, Durability) properties. In a monolith, we have a database system to ensure ACIDity because a local database transaction works on a single database system. A transaction has either all steps complete or no steps complete. If any step fails, the transaction is rolled back.

When designing a microservices-based architecture, you need to be aware of its challenges. Here we look at some common challenges.

Service Decomposition

When designing microservices, you must ensure that your monolith or legacy application is split into loosely coupled components with clearly defined boundaries. You need to check for the dependencies between components to see if they are sufficiently independent.

Complexity

A microservices application, being a distributed system, can have business transactions that span multiple systems and services. Service-level transactions (for clarity, let's call service-level transactions "sub-transactions") are called in a sequence or in parallel to complete the entire transaction. You must deal with additional complexity of creating a distributed system that uses an interservice communication mechanism and is designed to handle partial failures.

Distributed Transactions

In a microservices architecture, a monolithic system is decomposed into self-encapsulated services. With a database per microservice approach, to ensure that a transaction is complete, the transaction must span across multiple databases. If one of the sub-transactions fails, you must roll back the successful transactions that were previously completed. To maintain transaction atomicity, services need interservice communication and coordination to ensure that all the services commit on success of the transaction or the services that commit, roll back on failure of the transaction.

Transaction Consistency and Isolation

Another challenge involves handling concurrent requests. A sub-transaction works by keeping its data source (rows) locked until the result of the transaction is known. If multiple service calls try to simultaneously access the same data source, you must have a mechanism to handle concurrency and determine the amount of data that must be made available to concurrent transactions. In a Long-running Action (LRA) like a bank transaction that spans multiple services, data can be held in a locked state at multiple objects for the transaction lifecycle. To ensure that data remains consistent across services, you must implement a concurrency control mechanism using data locks.

Interservice Communication

In a monolith, application components use function calls to invoke each other. Microservices interact with each other over the network. Interservice communication using asynchronous messaging is essential when propagating changes across multiple microservices.

26.4 Solutions for Microservices

Two patterns, namely Two-phase Commit (2PC) and Saga, attempt to alleviate the challenges and bring in consistency in the distributed transaction management. 2PC is ideal for immediate transactions, and Saga works well if you wish to implement distributed transactions for long running action (LRA).

For enterprises looking for microservices solutions, Oracle has developed 12 proven patterns that are crucial for microservices success. You can find more information about these 12 patterns in the following section.

> **See Also:**
>
> Backend as a Service For The 12 Patterns For Microservices Success

26.4.1 Two-Phase Commit Pattern

Two-phase commit (2PC) is an atomic commit protocol where you have a coordinator that unilaterally decides the outcome of the transaction. All participants in a transaction must commit or roll back based on the coordinator's decision. All processes that occur in two or more tables, such as insert, update, delete, or all, commit or roll back simultaneously.

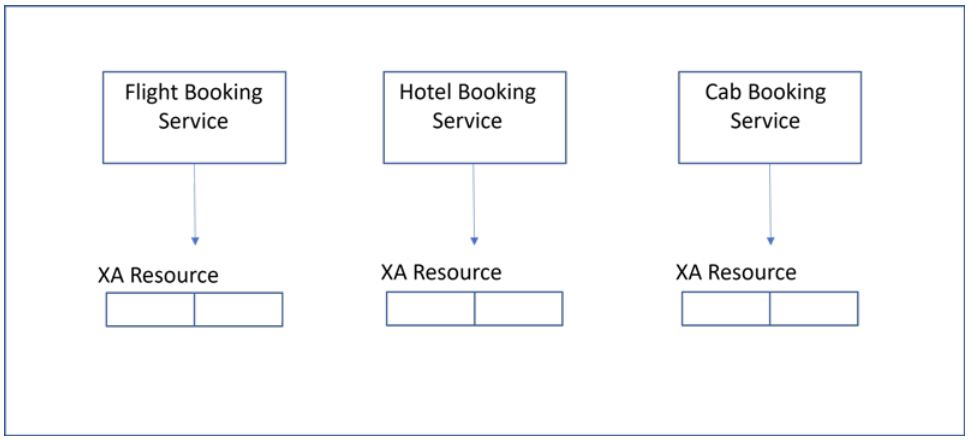
2PC has two phases: A prepare phase and a commit phase. In the prepare phase, the transaction coordinator sends a prepare command to each microservice. Each microservice checks if they can guarantee that they can commit the transaction and if yes, they send back a “prepared” response to the transaction coordinator.

After all the microservices are prepared, the coordinator directs the microservices to commit the change. If any microservice does not respond to the prepare command or sends back a failed response, the transaction coordinator sends a cancel command to all microservices. The sub-transactions are rolled back, canceling all the changes.

To ensure that a microservice can commit, 2PC ensures that the locks on modified data and the decision to prepare are in a durable storage. This lock remains active until the commit or rollback, and no other transaction can use this information. These locks can become bottlenecks that slow down your system. In an application handling multiple services, 2PC can create complexity and adverse performance impacts.

Using 2PC would reserve the flight, hotel, and cab services at the same time as shown in the Two-Phase Commit illustration below. If the transaction fails, none of the services are booked.

Figure 26-2 Two-Phase Commit



> **See Also:**
>
> Two-Phase Commit for more information about 2PC pattern.

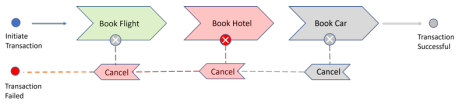
26.4.2 Saga Design Pattern

The Saga design pattern provides transaction management for microservices and enables business transactions to maintain data consistency across microservices. A Saga breaks up a distributed transaction into a sequence of independent local transactions (sub-transactions). Each service performs its own sub-transaction and publishes an event or message. The successive services listen to that event or message and perform the next sub-transaction.

A transaction spans multiple databases (PDBs) and may perform a sub-transaction for the database in each microservice. A business transaction is successful if all sub-transactions in the sequence complete successfully. If any of the sub-transactions fails, compensating transactions are invoked to “undo” the state in the database(s) affected by the changes of the preceding sub-transactions. Each sub-transaction can have ACID properties on a single database. Each sub-transaction in a Saga has a corresponding, compensating transaction that is executed if there is a rollback.

The Saga Transaction below illustrates the sequence of a Saga transaction when the transaction is successful and when the transaction fails prompting a rollback.

Figure 26-3 Saga Transaction



An important facet of the Saga pattern is the Saga coordinator. The Saga coordinator tells other participants what to do. The coordinator invokes Saga participants and with every response, the coordinator transitions to the next state. Asynchronous messaging is another important aspect of Sagas, which involves sending interservice messages over a queuing system.

The Saga pattern enables you to build more robust systems because Sagas use a failure management pattern. Every action has a compensating action for rollback, which helps ensure eventual data consistency and correctness across microservices.

26.4.2.1 Why Use Sagas?

Use Sagas for the following reasons:

- Perform a group of operations related to different microservices automatically.
- Rely on the Saga pattern to ensure data consistency across microservices and in different databases.
- Reduce the locking period and restrict the data locks to the duration of local transactions for improved concurrency.
- Avoid using Two-phase Commit (2PC) because of its extended locking period and performance constraints.
- Roll back or compensate if one of the transaction operations in the sequence of microservices fails.
- Use microservices that do not support 2PC.

Why using Sagas is a better option than using 2PC transactions?

- A 2PC transaction can cause extended locking period and incomplete results for queries until the 2PC transaction is committed or canceled. With Sagas, the data locks are placed only for the duration of the local transaction (that implements a microservice transaction), and not for the entire Saga lifecycle. Reducing the locking period improves the throughput of Saga transactions, resulting in better scalability for applications.
- Sagas are useful for long-lived transactions. 2PC is not ideal for long-lived transactions since the locking of resources for prolonged durations can affect performance and scalability.

When you use Sagas with Oracle Database, lock-free reservation features that are built into the database help improve concurrency and reduce bottlenecks.

26.4.2.2 Saga Implementation Approaches

There are two common saga implementation approaches, namely the orchestration and choreography models.

Orchestration

All communication between microservices is made through a centralized service called a Saga coordinator. The coordinator service is responsible for receiving the requests and calling the respective services. If any service fails, the coordinator service implements the roll back methods. You can use the orchestration model for complex workflows that need the Saga coordinator services.

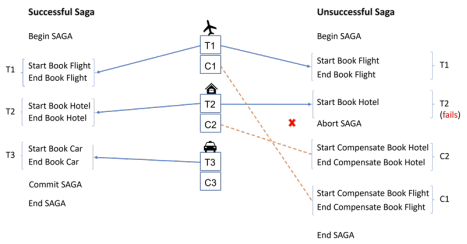
Choreography

In the choreography model, services communicate amongst each other and if any request fails, each service must have its own fallback method to roll back the transaction. If you have simpler workflows, you can use the choreography model.

26.4.2.3 Successful and Unsuccessful Sagas

Successful and Unsuccessful Sagas illustrates a successful and an unsuccessful Saga transaction.

Figure 26-4 Successful and Unsuccessful Sagas



26.4.2.4 Saga Flow

Taking the example of the travel agency application, let us look at a trip booking Saga flow.

1. A travel agency application user sends a book trip request from the application UI, which is sent to the travel agency service.

2. The travel agency service calls on the Saga coordinator to begin the Saga.
3. The coordinator includes a Saga identifier (Saga ID) in its response.
4. The travel agency registers itself with the Saga. The agency sends the Saga completion (compensation or commit) callback to the coordinator. Each participant in a Saga must provide the Saga with a callback that is used at the time of Saga completion.
5. The travel agency adds the Saga ID to a request call to the flight microservice to book a flight.
6. The participant microservice (flight) contacts the coordinator to join the Saga (register itself to the Saga). The participant service also sends its Saga completion (compensation or commit) callback to the coordinator.
7. The travel agency repeats the same process for other participants, such as the hotel and car rental services.
8. The participant microservices execute the business process.
9. After the travel agency receives all replies, it determines whether to commit or roll back the Saga and informs the coordinator.
10. The coordinator calls the appropriate callbacks (compensation or commit) for the Saga participants and returns the control to the travel agency.
11. The travel agency confirms the success or failure of the Saga to the coordinator and ends the Saga.

26.4.3 Backend as a Service For The 12 Patterns For Microservices Success

If you search for microservices success stories, you find that the industry is divided on the advantages that microservices can bring to businesses.

Traditionally, monoliths have been the standard architecture where the entire application has a single domain and data access pattern, keeping the transactional boundaries local within a single database; and rarely do applications need a distributed transaction. This brings in some simplicity to data design, but it also results in a spaghetti pattern of accesses to the tables in a schema, and a hugely complex entity-relationship setup. Therefore, one big disadvantage of monoliths is the complexity of making changes and the time it takes to launch a new feature in the application.

Microservices (microservices architectures), on the other hand, advertises its advantages as agility, which it achieves by bounding the context and using loose coupling between the various microservices. This limits the access patterns (to the data from a microservice) to be local, and in very rare cases, rely on the transactions across microservices, supported by Sagas, which are asynchronous and scalable.

However, some of the advantages of microservices are negated by (1) the overhead of setting up the infrastructure for integration of microservices with APIs or messaging, and (2) the overhead of testing even a single microservice change when the number of microservices is in the hundreds. Bounded contexts are important to identify such data layouts that force loose coupling of data accesses across microservices, moving away from the tight coupling used in monoliths. Extracting microservices from existing monoliths is also possible using the Strangler pattern, where independent functions can be determined by doing a bottoms-up affinity analysis. However, due to such issues, most enterprises struggle to build and deploy microservices architectures.

For enterprises using microservices, Oracle has leveraged its vast experience to develop the following 12 patterns for microservices success:

1. **Bounded contexts** - design it upfront, or break monoliths into microservices with a data refactoring advisor and

   Loose coupling - decouple data by isolating schema to the microservice, and using a reliable event mesh
2. **Transactional Outbox** - sending a message and a data manipulation operation in a single local transaction
3. **Reliable Event Mesh** - Event Mesh for all events – with high throughput transactional messaging and pub/sub; with event transformations and event routing, alongside Kafka (if needed)
4. **Sagas** - transactions across microservices with support from Event Mesh and Escrow journaling in the database
5. **Security** for microservices - securing each endpoint from an API gateway, to load balancer, to event mesh, to the database
6. **Polyglot microservices** - support for microservices and message formats in a variety of languages using JSON as the payload
7. **Unified observability** for microservices - metrics, logs, and traces into a single dashboard for tuning and self-healing
8. **Event Aggregation** - events are ephemeral and notify/trigger real-time action; after which they are aggregated into the database – the database is the ultimate compacted topic
9. **CQRS** - Command Query Responsibility Segregation, where operational and analytical copies of data are available for microservices
10. **CI/CD** for microservices - a multi-tenant database makes it a natural fit for microservice + PDB to deploy/build independently; and also an app (with Jenkins or GitHub Actions) and schema (Liquibase, Flyway, and EBR in the database)
11. **AI Sandbox for Microservices** - an experimental sandbox to run RAG and related techniques to increase enterprise accuracy for Chatbots and other use cases – making them ready for deployment
12. **Backend as a Service (BaaS)** - a microservices infrastructure right sized for test/dev and production deployments (small, medium) on any cloud or on-premises deployments; for Spring Boot Apps

The Backend as a Self-Service (BaaS) pattern brings all the preceding patterns into a successful dev/test and production environment by making the deployment of the microservices platform easy.

The next chapter discusses about Oracle Backend as a Self-Service (OBaaS), which is also called the Oracle Backend for Microservices and AI platform.

Spring Boot is the most popular microservices framework for Java, and OBaaS enables Java developers to simplify the task of building, testing, and operating Spring Boot-based microservices.