A Crash Course on Microservices Design Patterns

BYTEBYTEGO
AUG 08, 2024 · PAID

Microservices architecture has gained popularity for its ability to improve the flexibility, testability, and scalability of software systems.

By breaking down a monolithic application into smaller, independently deployable services, microservices enable teams to develop, deploy, and scale each service independently.

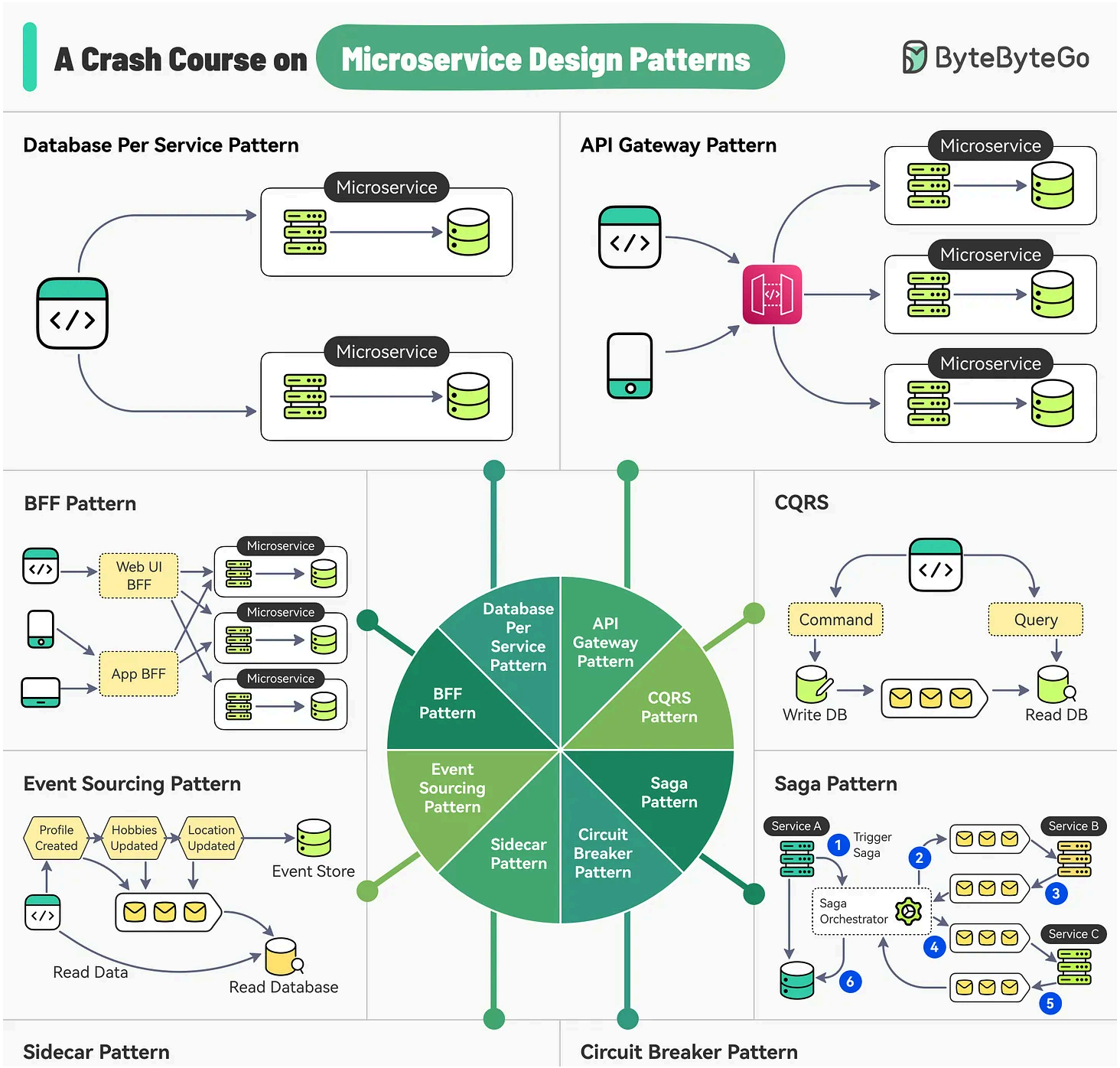
However, the implementation of microservices architecture comes with its own set of challenges such as:

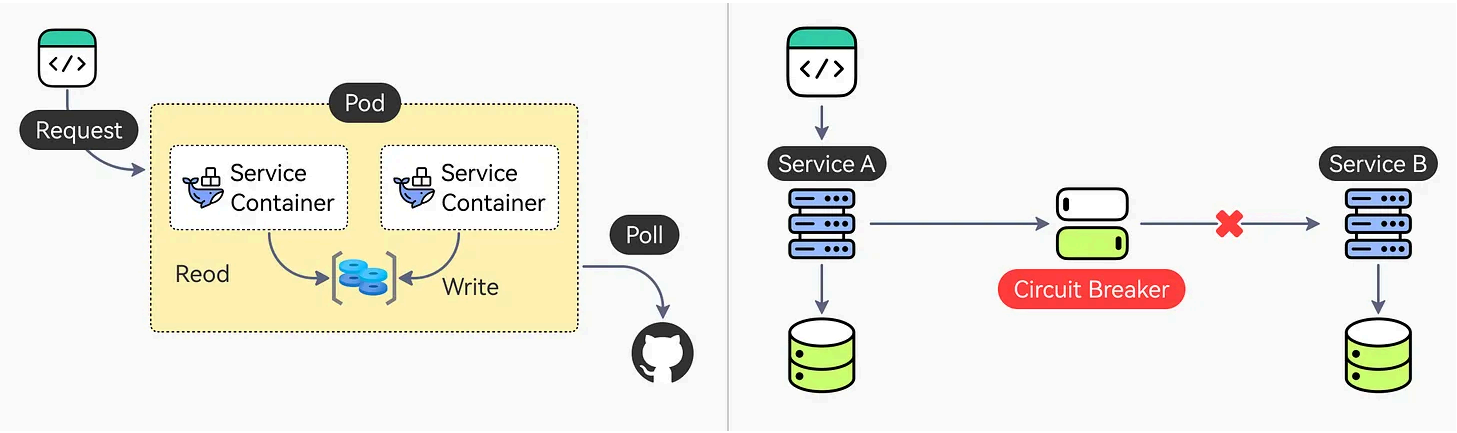
- **Data Consistency and Eventual Consistency:** In a microservices architecture, data is often distributed across multiple nodes, which can be located in different data centers or even different geographic regions. At any given point in time, there can be discrepancies in the state of data between various nodes. This phenomenon is known as eventual consistency.
- **Security:** Microservices architecture introduces a larger attack surface for malicious actors compared to monolithic systems. It's crucial to establish appropriate security mechanisms while building microservices. Design patterns such as the API Gateway pattern can help.

- **Scalability and Database Performance:** Microservices are known for their scalability. However, while it is relatively easy to scale the application layer by adding more instances, databases can become performance bottlenecks if not designed for scalability. Patterns such as Database per Service and CQRS help solve this challenge.

Design patterns provide proven solutions to common problems encountered in a microservices architecture. By applying an appropriate design pattern, these problems can be effectively addressed.

In this post, we'll explore the most popular microservices design patterns along with their benefits and adoption challenges.


A Crash Course on
Microservice Design Patterns
ByteByteGo
Database Per Service Pattern
Microservice
Microservice
API Gateway Pattern
Microservice
Microservice
Microservice
BFF Pattern
Web UI BFF
App BFF
Microservice
Microservice
Microservice
Database Per Service Pattern
API Gateway Pattern
BFF Pattern
CQRS Pattern
Event Sourcing Pattern
Saga Pattern
Sidecar Pattern
Circuit Breaker Pattern
CQRS
Command
Query
Write DB
Read DB
Event Sourcing Pattern
Profile Created
Hobbies Updated
Location Updated
Event Store
Read Data
Read Database
Saga Pattern
Service A
Trigger Saga
Service B
Saga Orchestrator
Service C
1
2
3
4
5
6
Sidecar Pattern
Circuit Breaker Pattern

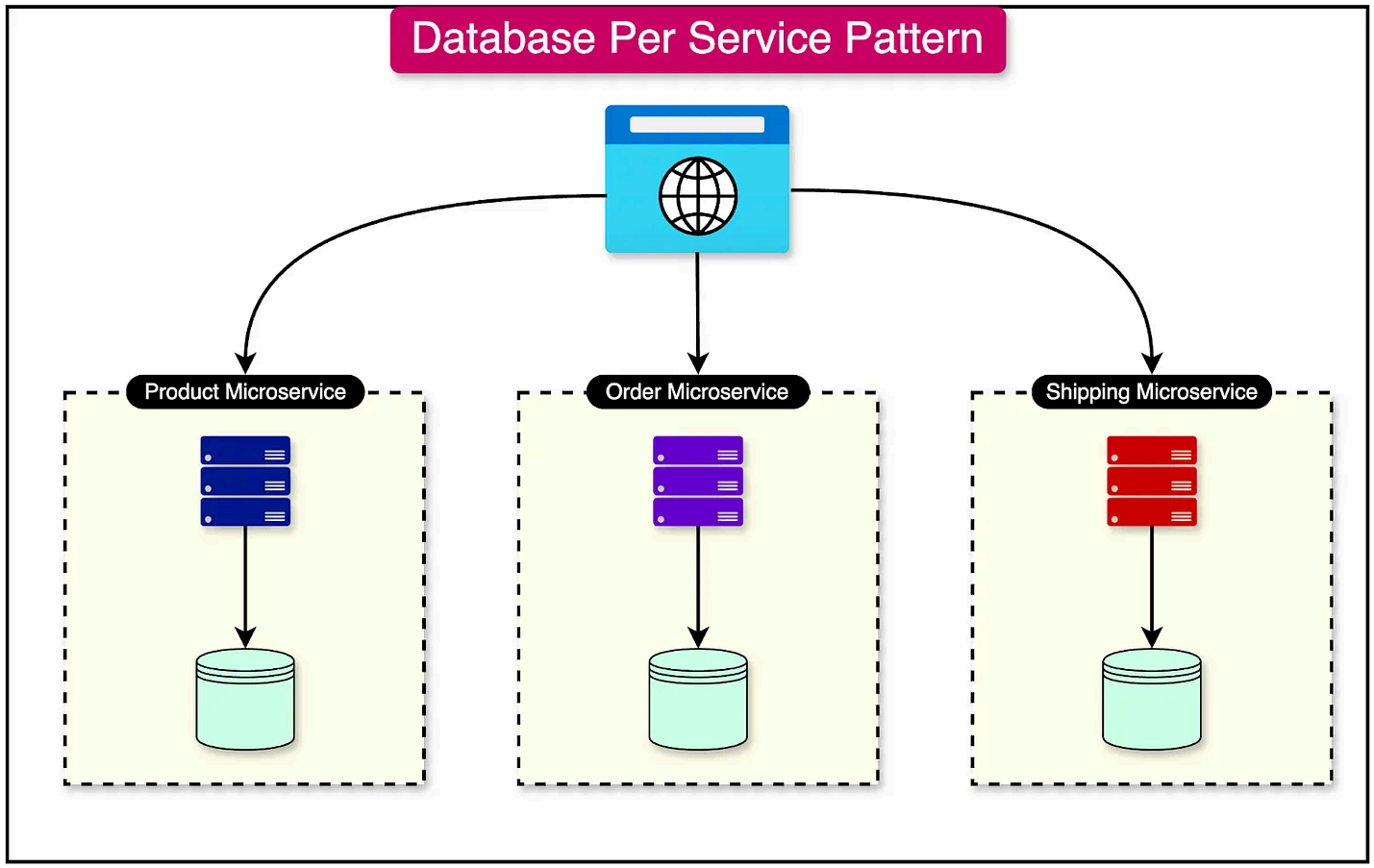


Database Per Service Pattern

As the name suggests, the Database Per Service pattern proposes that each microservice should have its dedicated database, ensuring data isolation and autonomy.

In this pattern, no microservice can directly access or manipulate the data managed by another microservice. Instead, any exchange or manipulation of data must be done through a set of well-defined APIs exposed by the microservice that owns the data. The result is an architecture that enforces a clear separation of concerns and promotes loose coupling between microservices.

The diagram below shows an example of the database per service pattern.



The database-per-service pattern is easier to implement when starting with a brand-new application.

However, when migrating an existing monolithic application to a microservices architecture, the demarcation between services is not as clear-cut. This is because functionality in a monolithic application is often written in such a way that different parts of the system can access data from

other parts informally. This tight coupling and lack of clear boundaries make it challenging to cleanly separate data ownership when transitioning to a microservices architecture.

When adopting the database-per-service pattern, two main areas require careful consideration:

- **Defining Bounded Contexts for Each Service:** Bounded contexts are a concept from the domain-driven design philosophy that helps define the boundaries and responsibilities of each microservice. It is crucial to identify and establish clear bounded contexts for each service so that each microservice encapsulates a specific domain or business capability. This process involves analyzing the application's domain model, identifying the entities and their relationships, and aligning them with the microservices architecture.
- **Managing Business Transactions Spanning Multiple Microservices:** In a microservices architecture, business transactions often span multiple microservices. When a single transaction involves data managed by different microservices, it becomes challenging to maintain data consistency and integrity. Distributed transactions and eventual consistency techniques need to be employed to handle such scenarios. This may also involve using techniques like the Saga pattern or event-driven architecture to coordinate and manage transactions across microservices.

Benefits

Some key advantages of the database-per-service pattern are as follows:

- By adopting the database-per-service pattern, microservices can achieve better data isolation, scalability, and autonomy.

- Each microservice can evolve its data model independently, without impacting other services.
- Additionally, this pattern allows for the use of different database technologies that best suit the specific needs of each microservice.

Challenges

The database-per-service pattern also introduces some challenges such as:

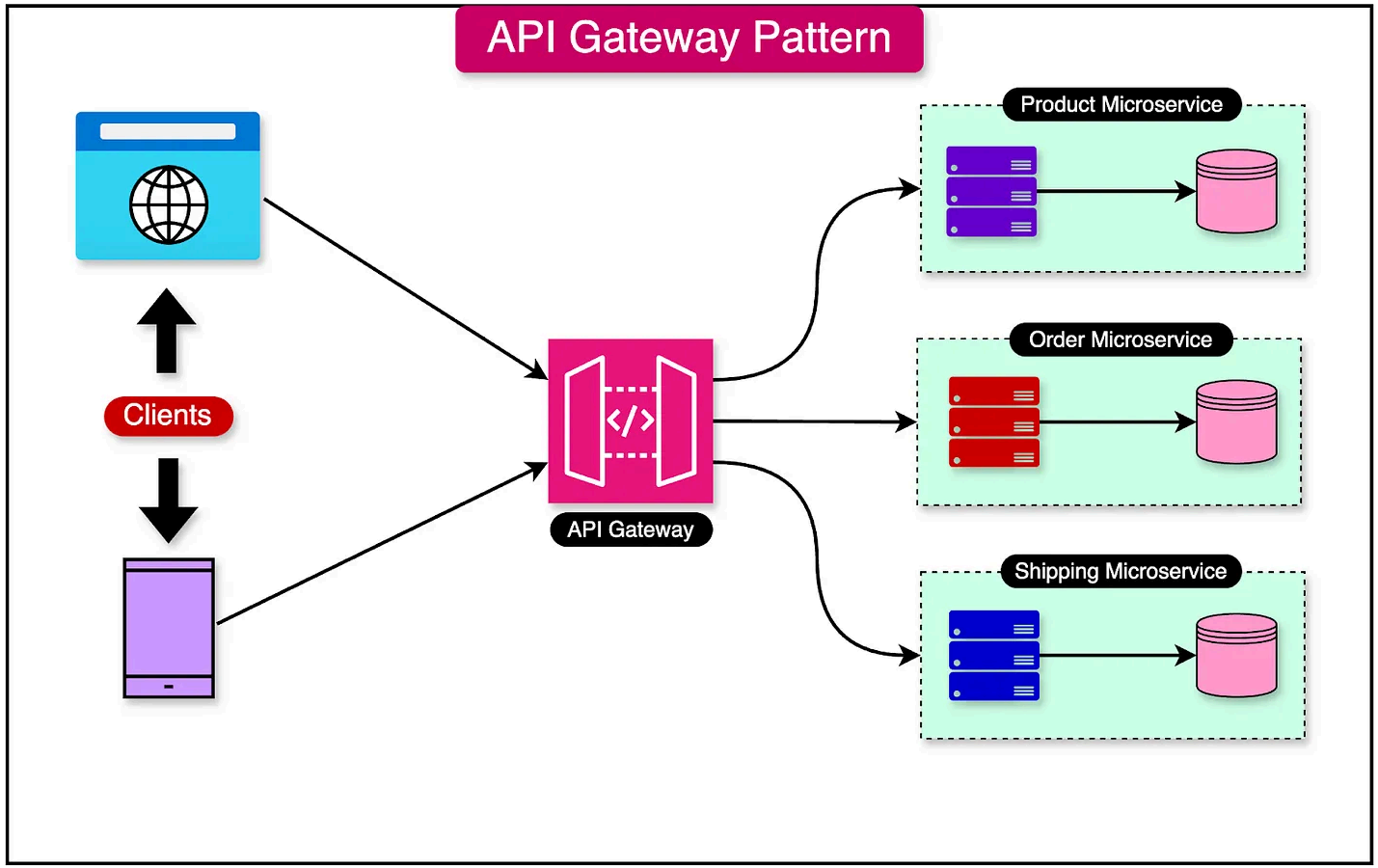
- Managing data consistency across microservices.
- Handling complex queries that span multiple services.
- Transaction management across a business process that spans multiple bounded contexts.

API Gateway Pattern

When microservices are designed in a granular manner, focusing on specific functions, the client application or the UI may need to communicate with many microservices to support a particular feature.

This can lead to challenges, such as increased complexity, latency, and security concerns.

The API Gateway pattern offers a solution to these issues by acting as an intermediary between the client application and the backend microservices. It serves as a single entry point for the client, simplifying communication and reducing the number of round trips required.



Here's how the API Gateway pattern works:

- **Reverse Proxy:** The API Gateway acts as a reverse proxy, routing client requests to the appropriate microservices based on the request's characteristics, such as the URL path or request parameters.
- **Request Aggregation:** The API Gateway can aggregate requests to multiple microservices and combine their responses before sending them back to the client. This is particularly

useful when the client needs data from multiple microservices to render a single view or perform a specific operation. By aggregating the requests and responses, the API Gateway reduces the number of round trips and improves performance.

- **Cross-Cutting Concerns:** The API Gateway can handle cross-cutting concerns that are common across multiple microservices. These concerns include authentication, authorization, rate limiting, caching, and logging. By centralizing these functionalities in the API Gateway, developers can ensure consistent implementation and reduce the burden on individual microservices.
- **Security:** The API Gateway serves as a single point of contact with users, enhancing security. It can enforce authentication and authorization policies, validate and sanitize incoming requests, and protect against common security threats such as SQL injection and cross-site scripting (XSS).
- **Protocol Translation:** The API Gateway can translate between different communication protocols used by the client and the microservices. For example, if the client uses HTTP/REST and some microservices use gRPC, the API Gateway can handle the protocol translation.

Benefit

The API Gateway pattern brings several benefits to a microservices architecture:

- The client application only needs to communicate with a single endpoint (the API Gateway) rather than managing multiple connections to individual microservices.
- The API Gateway reduces the number of round trips between the client and the microservices.

- The API Gateway provides a central location to implement cross-cutting concerns such as authentication, authorization, and logging.
- The API Gateway acts as a security gateway, enforcing authentication and authorization policies.

Challenges of API Gateway

The API Gateway pattern also introduces some challenges:

- It can become a single point of failure if not designed and implemented properly.
- The API Gateway can become a bottleneck if it performs a lot of processing.

Backends-for-Frontends Pattern

When adopting microservices, it is common to decouple the frontend and backend services. Clients communicate with the services using APIs.

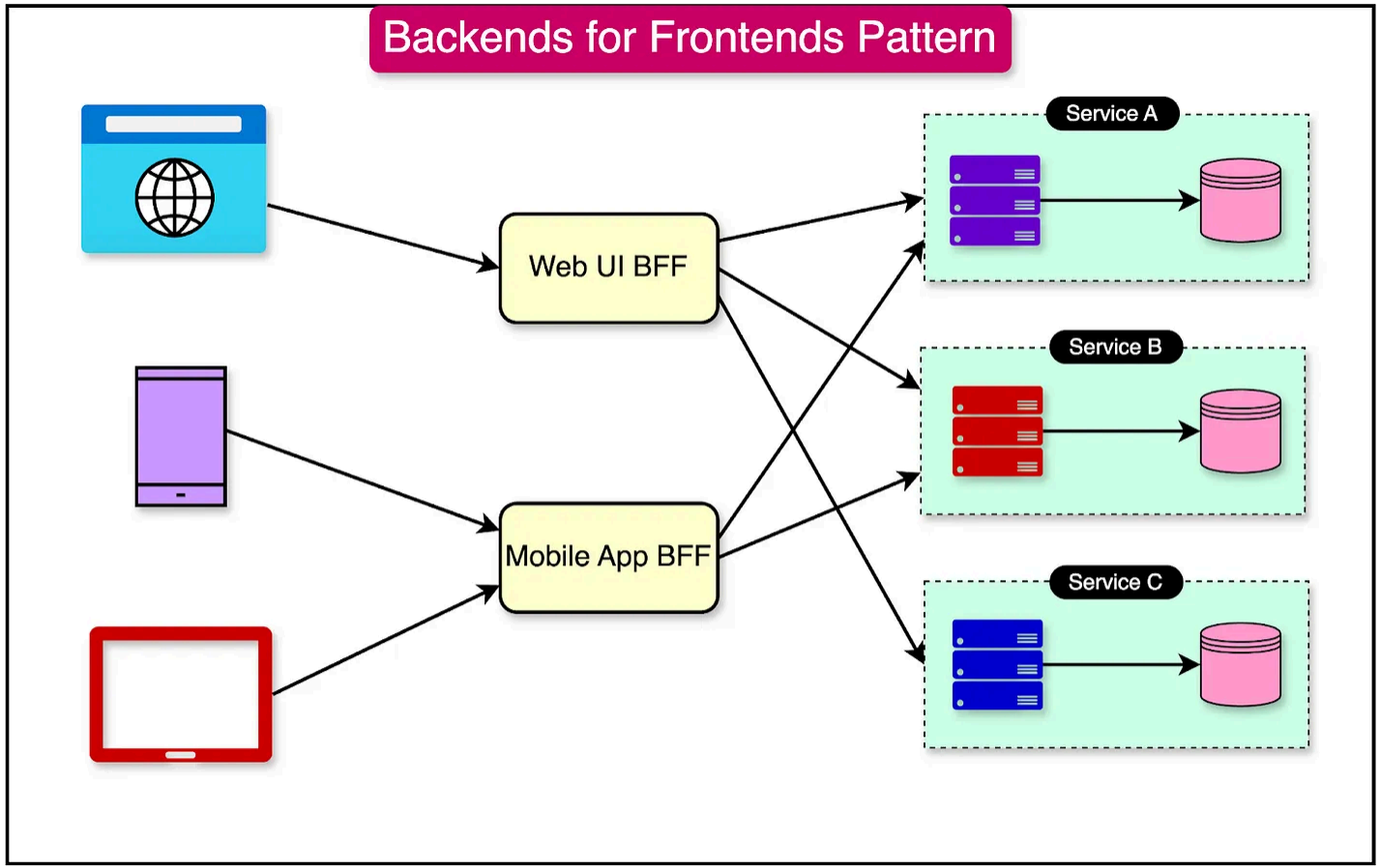
However, there are times when using the same backend microservices for different front-end clients, such as web and mobile applications, can lead to challenges.

Web and mobile applications have distinct requirements in terms of screen size, display, performance, network bandwidth, and other factors. Trying to accommodate these diverse needs within a single backend can become problematic.

This is where the Backends for Frontends (BFF) pattern comes into play.

The BFF pattern proposes creating separate backend services tailored specifically for each front-end client. In other words, each user interface (UI) gets a dedicated backend designed exclusively to cater to its unique requirements.

The diagram below shows an example of the BFF pattern.



Here's how the BFF pattern works:

- **Separate Backends:** Instead of having a single backend serving all frontend clients, the BFF pattern advocates for creating separate backend services for each frontend.
- **Customized APIs:** Each BFF is responsible for providing API endpoints optimized for the specific needs of its corresponding front end.
- **Facade for Downstream Microservices:** The BFF acts as a facade or gateway for the downstream microservices and abstracts away the complexities of the underlying microservice architecture for the client.
- **Reduced Chatty Communication:** The BFF layer reduces chatty communication between the front end and the downstream microservices.

Benefits of BFF Pattern

The BFF pattern offers several important benefits:

- By having separate backends for each frontend, developers can optimize the user experience for each client.
- The BFF abstracts away the complexities of the service architecture from the front-end developers, allowing them to focus on building the user interface.
- The BFF can perform aggregation and other optimizations, thereby improving the performance of the client application.
- The BFF allows for independent evolution and scaling of the client and service layers.

Challenges

The BFF pattern also introduces some challenges:

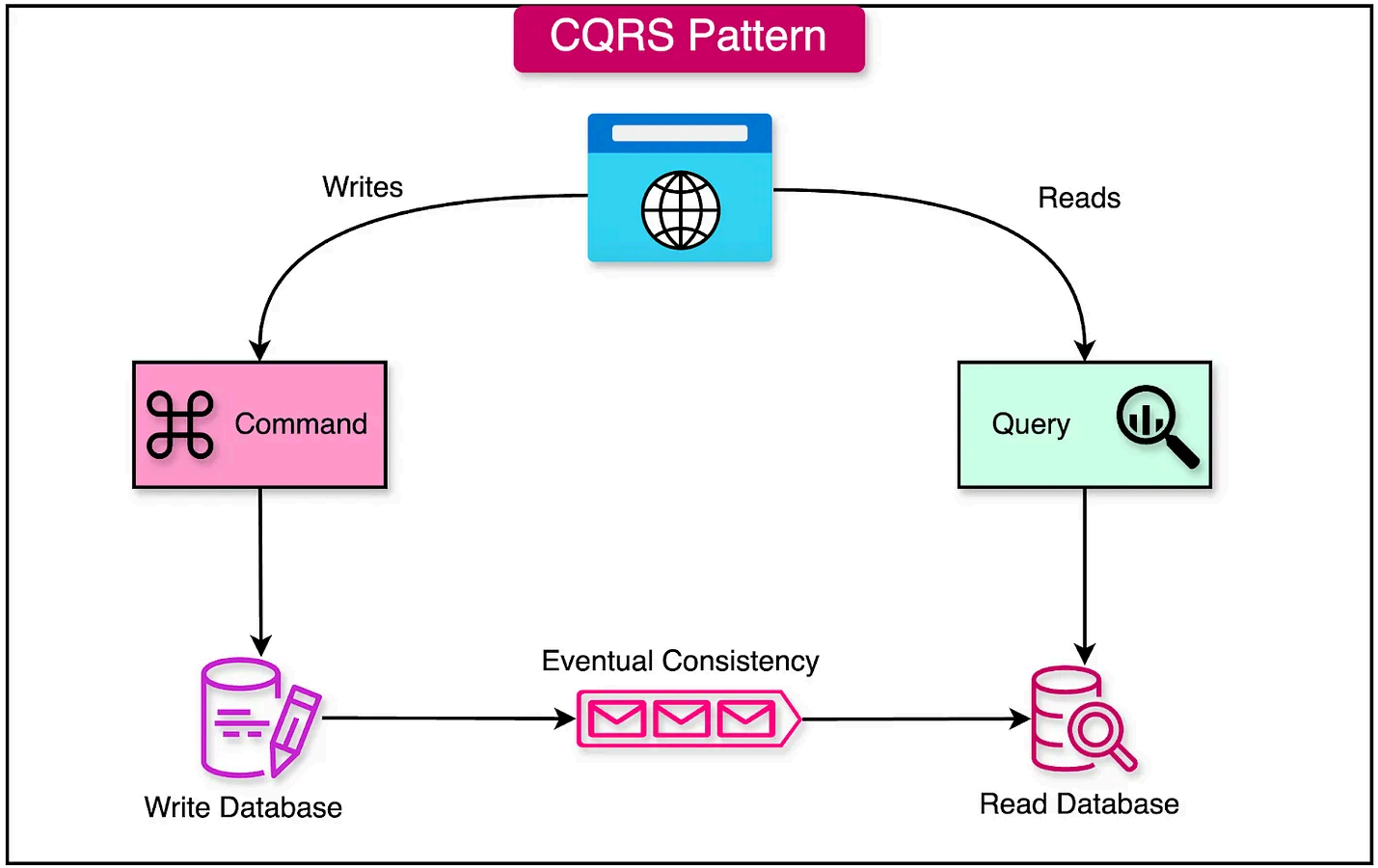
- Increased complexity since there are multiple backends to manage and maintain.
- The ownership of the BFF can become a point of contention.
- It requires an initial upfront investment of coordination effort between the client and service teams to ensure that the right BFFs are built.

Command Query Responsibility Segregation Pattern

Command Query Responsibility Segregation (CQRS) is an architectural pattern that separates the read and write operations of an application into distinct paths.

By segregating these responsibilities, CQRS enables optimized performance and independent scaling of read and write operations.

In a microservices architecture, the CQRS pattern can be implemented by having each microservice maintain separate databases for reads and writes. The write database is used for handling write operations and persisting the state changes, while the read database is optimized for serving complex queries and aggregations.



Here's how the CQRS pattern works in conjunction with microservices:

- **Domain Events:** When a microservice performs a write operation that results in a state change, it publishes a domain event. Domain events represent the significant occurrences within the bounded context of the microservice.
- **Updating Read Database:** An event listener processes the received domain events and updates the read database. The read databases are specifically designed and optimized for

supporting efficient querying and aggregation operations.

- **Serving Queries:** When a client application needs to retrieve data, it sends a query request to the query processing service that fetches data from the read database

Benefits of CQRS

By separating the read and write paths, CQRS offers several benefits:

- The read database can be designed and optimized specifically for complex queries and aggregations. It can employ denormalization techniques, materialized views, and caching mechanisms to ensure fast response times for read operations.
- CQRS allows the read and write paths to scale independently based on their specific performance and resource requirements. The write path can be scaled to handle high volumes of write operations, while the read path can be scaled to handle a large number of concurrent queries. This enables better resource utilization and scalability.
- By maintaining a separate read database, the query logic can be simplified and optimized without impacting the write operations. The read database can be structured in a way that supports efficient querying, making it easier to develop and maintain complex query logic.

Challenges

CQRS also introduces some challenges:

- It requires careful coordination and synchronization between the read and write databases to ensure data consistency.

- The eventual consistency model of CQRS may not be suitable for all scenarios, particularly those requiring strong consistency guarantees.

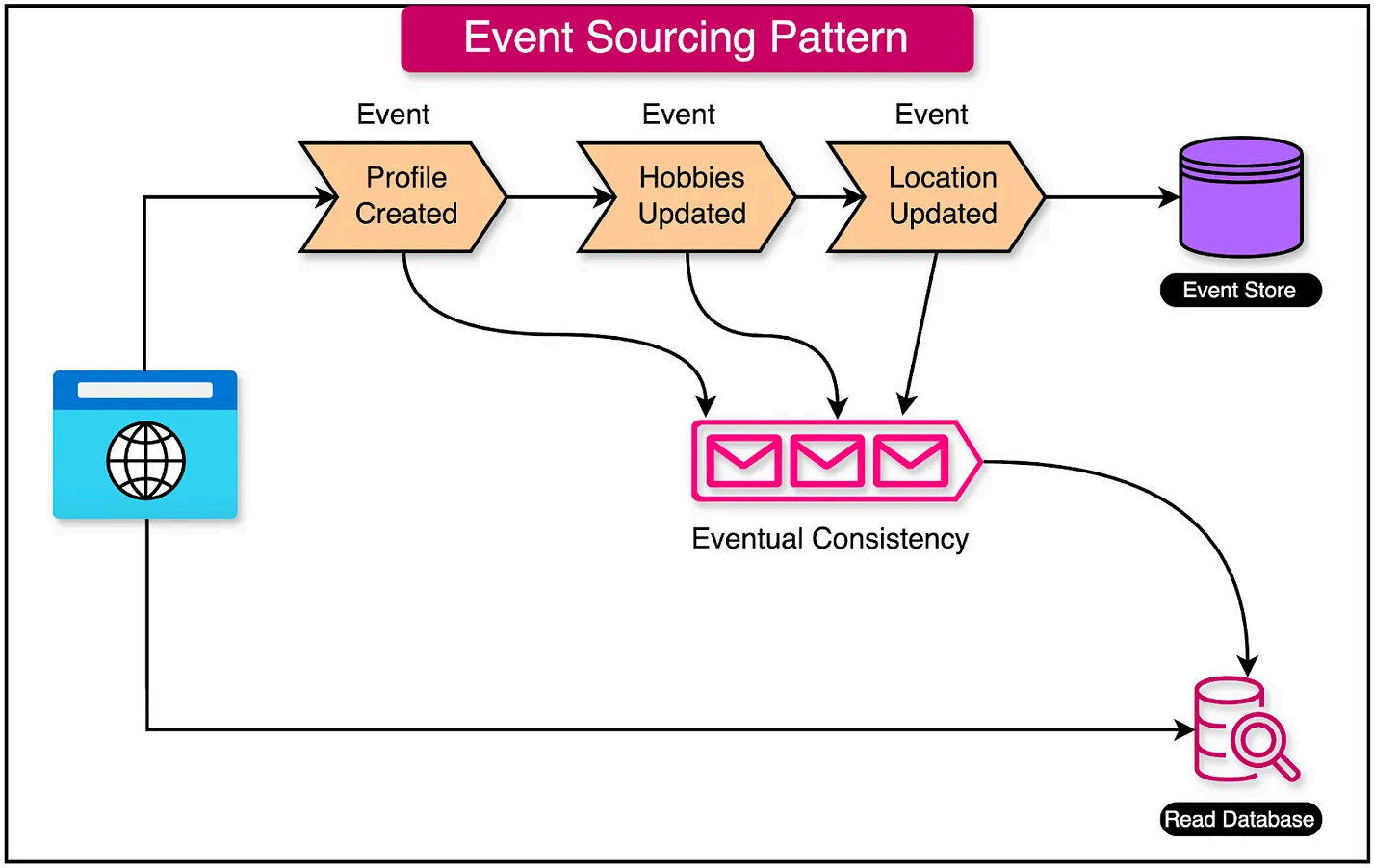
Event Sourcing Pattern

Traditional CRUD (Create, Read, Update, Delete) models have certain limitations that can impact system performance, responsiveness, and scalability. These limitations arise from the processing overhead associated with updating and retrieving data directly from a database.

The Event Sourcing pattern is a powerful approach that can help overcome these limitations.

In this pattern, the state of an entity is not modified in a database. Instead, every change event in the entity's state is captured and persisted in the database. Also, this pattern is often used in conjunction with the CQRS pattern to decouple read and write commands.

The diagram below shows an example of the Event Sourcing pattern where a user's social media profile information is stored as a series of updates in an event store.



In this context, events could include actions such as creating a profile, updating their hobbies, or modifying their current location. Each of these events represents a specific change in the state of the user's profile.

Here's how the Event Sourcing pattern works:

- **Event Store:** Every event that occurs in the system is stored in an event store. The event store acts as a repository for all the events, maintaining a complete history of the system's state changes.
- **Message Queue:** When an event is stored in the event store, it is also added to a messaging queue for propagation to other components of the system.
- **Materialized View:** The events stored in the event store are used to construct a materialized view database. This database provides a read-optimized representation of the current state of the entities, allowing users to access the information in real time.
- **Event Replay:** If a user requests the state of an entity at a specific point in history, the system can replay the events from the event store up to that point. This enables the reconstruction of the entity's state at any given time, providing a historical record of the system's behavior.

Benefits of Event Sourcing

By storing events instead of directly updating data records, the Event Sourcing pattern offers several benefits:

- The overhead on write operations is reduced since they only involve appending events to the event store. The event store can handle a high volume of write operations efficiently, as it primarily focuses on appending events.
- The event store maintains a complete history of all the changes in the system. This provides an audit trail that can be used for debugging, troubleshooting, and compliance purposes.

- The materialized view database can be optimized for read operations, allowing it to scale independently.
- The Event Sourcing pattern enables the system to add new features and behavior with minimal impact. New events can be introduced without modifying the existing event history.

Cons

The Event Sourcing pattern also introduces some challenges:

- Careful design and implementation to ensure the consistency and integrity of the event store and the materialized view.
- Increased complexity in terms of event versioning, event schema evolution, and handling of long-running processes.

Saga Pattern

In a microservices architecture, business transactions often span multiple services.

For example, placing an order on an e-commerce platform involves interactions between various services such as the order service, payment service, and shipping service. Ensuring the consistency and integrity of such distributed transactions can be challenging.

The Saga pattern addresses this challenge by breaking down a business transaction into a sequence of local transactions, each handled by a different service.

In this pattern, each service involved in the business transaction performs its local transaction and publishes an event upon completion. This event triggers the next transaction in the sequence, which is handled by another service. The chain of transactions continues until the entire business transaction is completed.

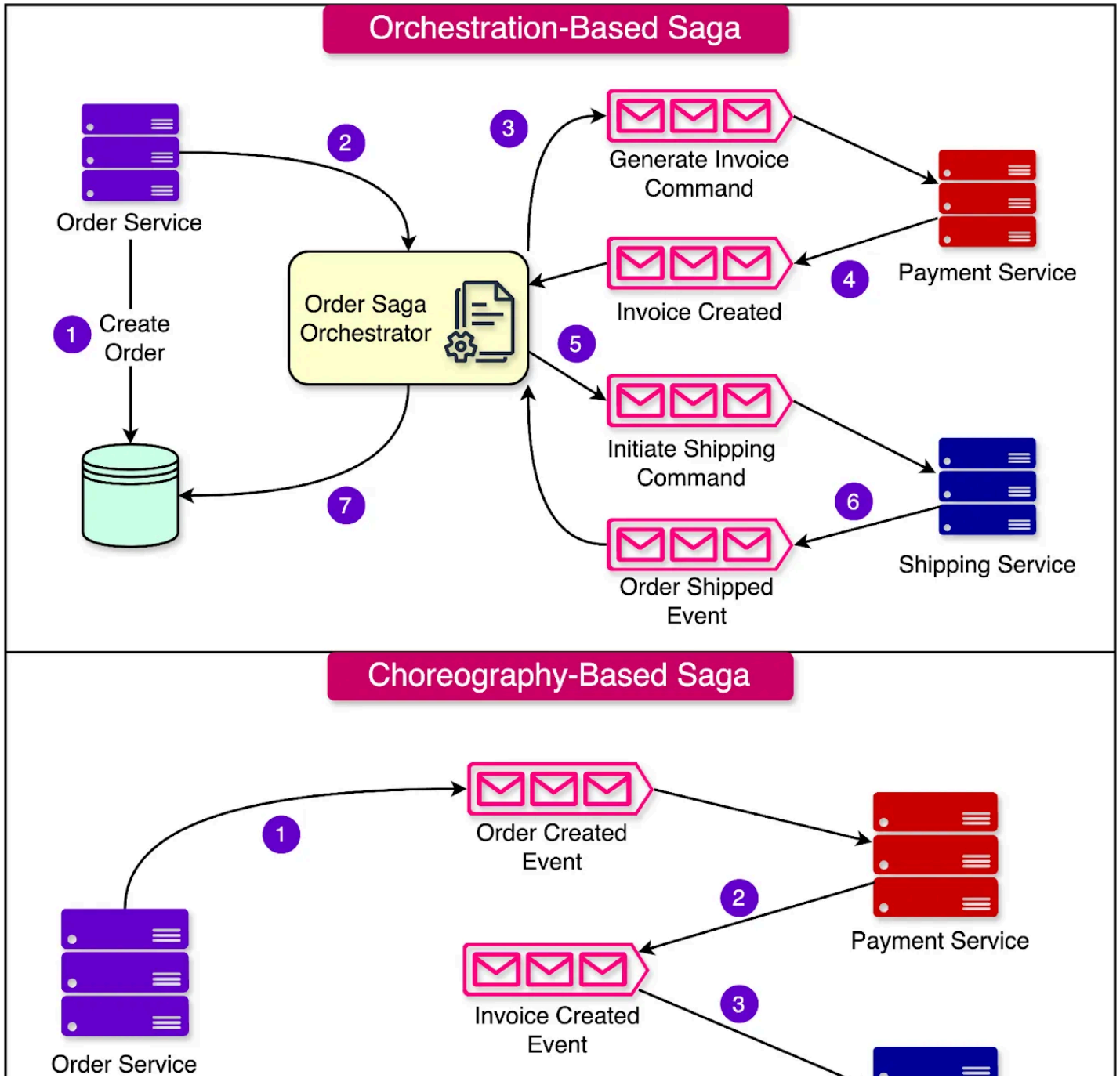
If any particular transaction in the chain fails, the Saga initiates a rollback process by executing a series of compensating transactions. These compensating transactions undo the impact of all the previous transactions, ensuring that the system remains in a consistent state.

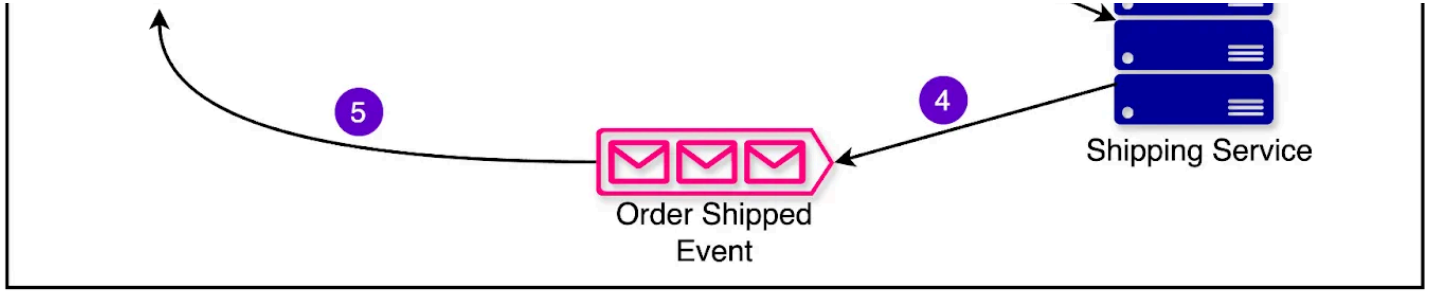
There are two main approaches to implementing the Saga pattern:

- **Orchestration-Based Saga:**
    - In this approach, there is a central orchestrator (like a conductor in an orchestra) that coordinates the steps of the Saga and makes decisions.
    - The workflow of the Saga is explicitly defined in the orchestrator, making it easier to visualize and reason about the process.
    - The orchestrator is responsible for invoking the local transactions on the participating services and handling the compensating actions if necessary.
- **Choreography-Based Saga:**
    - In this approach, each service involved in the Saga communicates directly with others by publishing and subscribing to events.
    - Services have autonomy and make decisions locally based on the events they receive.
    - When a service completes its local transaction, it publishes an event that triggers the next step in the Saga.

- If a transaction fails, the services publish a compensating event that triggers the rollback process.

The diagram below shows two different types of Saga implementations in the context of an e-commerce order flow.


Orchestration-Based Saga
Order Service
1
Create Order
2
Order Saga Orchestrator
3
Generate Invoice Command
Payment Service
4
Invoice Created
5
Initiate Shipping Command
6
Shipping Service
Order Shipped Event
7
Choreography-Based Saga
1
Order Created Event
Order Service
2
Payment Service
Invoice Created Event
3



Orchestration-based Sagas work well for complex and well-defined processes with strict dependencies between the steps. On the other hand, Choreography-based Sagas work well for dynamic workflows where services can change independently and for situations that require loose coupling between services.

Benefits of the Saga Pattern

The Saga pattern offers several benefits in handling distributed transactions in a microservices architecture:

- By breaking down a business transaction into a sequence of local transactions, the Saga pattern ensures that the system remains in a consistent state.
- The Saga pattern allows services to operate independently and perform their local transactions, enabling better scalability and performance.

Challenges

Implementing the Saga pattern also comes with some challenges:

- Coordinating and managing the sequence of transactions across services can introduce complexity.

- The Saga pattern relies on eventual consistency, meaning that the system may be in an inconsistent state temporarily until all the transactions are completed or compensated. This may not be suitable for scenarios that require strict consistency guarantees.
- Designing and implementing compensating transactions can be challenging, especially when dealing with external systems or irreversible actions.

Sidecar Pattern

In a microservices architecture, an application typically consists of two types of functionalities: domain and operational.

While domain functionalities benefit from loose coupling, operational functionalities such as logging, monitoring, authentication, and circuit breakers often require a high-coupling implementation for optimal performance and standardization across the organization.

To avoid each service team reinventing the wheel for operational functionalities while ensuring consistent solutions, the Sidecar pattern can be employed.

The Sidecar pattern involves deploying a minimum of two containers:

- **Application container:** This container houses the core logic that serves the business requirements of the microservices. It focuses on domain-specific functionality.
- **Sidecar container:** This container augments the application container by providing operational functionalities such as logging, monitoring, configuration management, and service discovery.

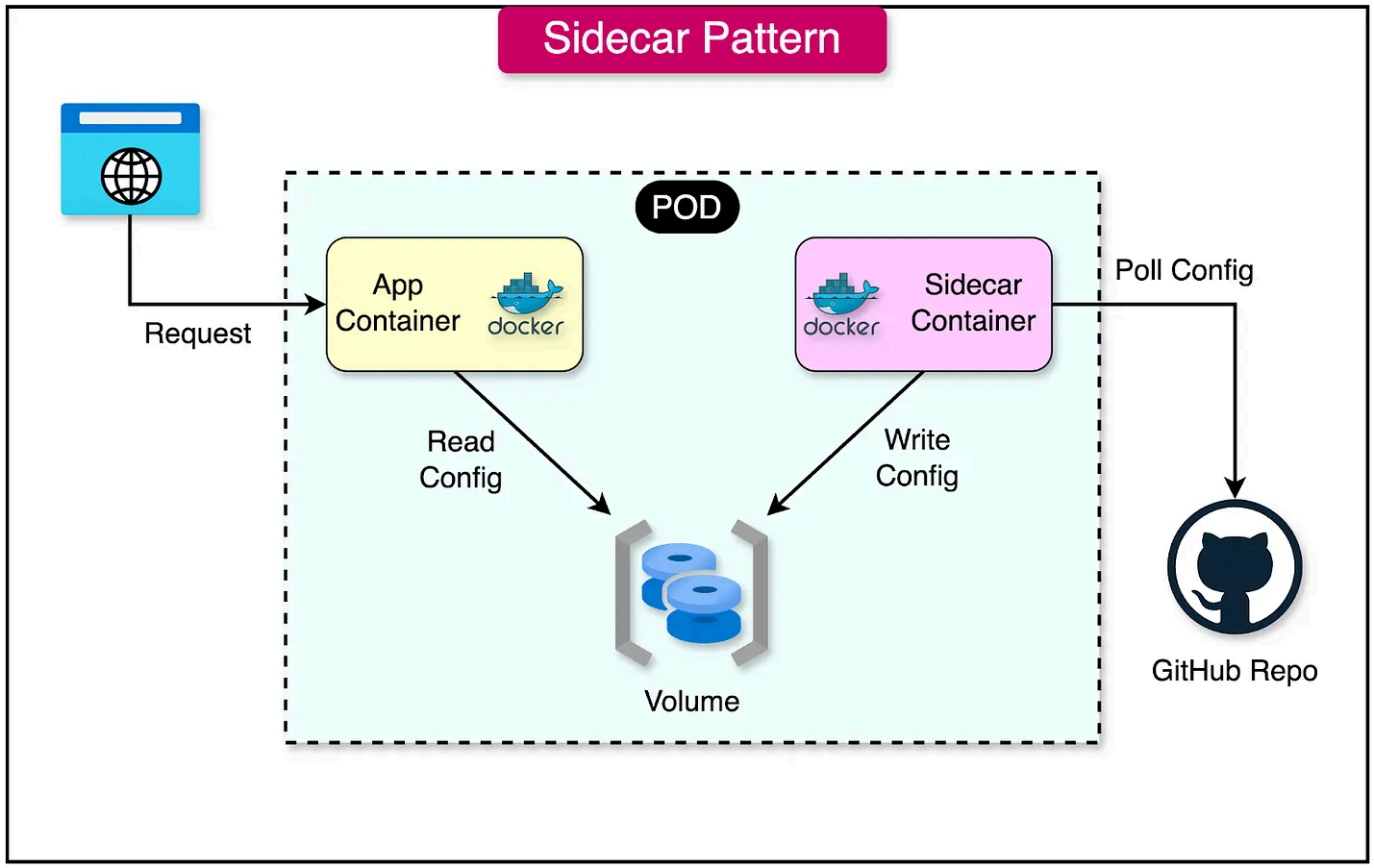
These containers are co-scheduled to the same machine using a container orchestration platform like Kubernetes. By leveraging the Pod API in Kubernetes, the application container and the sidecar container can share resources, including:

- Parts of the filesystem
- Hostname and network
- Namespaces

This co-scheduling and resource sharing enables seamless communication and coordination between the application and the sidecar container.

To illustrate how the Sidecar pattern works, let's consider an example scenario.

Suppose you have an application that reads configuration settings from a file. Now, you want to upgrade the application to read the configuration from a GitHub repository without modifying the application code.



Here's how the Sidecar pattern can facilitate this upgrade:

- The Configuration Manager Sidecar starts up and checks the GitHub repository that has the configuration file.
- It compares the configuration in the repository with the current configuration.

- If changes are detected, the sidecar downloads the updated configuration from the repository and saves it to the shared filesystem.
- Finally, the sidecar signals the application to reconfigure itself based on the updated configuration.

Benefits of Sidecar Pattern

In the context of microservices architecture, the Sidecar pattern offers several advantages:

- The Sidecar pattern allows for a clear separation between the core application logic and the operational functionalities.
- By encapsulating operational functionalities in a sidecar container, the same sidecar can be reused across multiple microservices. This promotes standardization of operational functionalities throughout the system.
- The sidecar container can be scaled independently from the application container, allowing for fine-grained resource allocation.
- The Sidecar pattern provides flexibility in terms of adding or modifying operational functionalities without impacting the core application.

Challenges

The Sidecar pattern also introduces some challenges:

- Introducing sidecars adds an extra layer of complexity to the system.
- Managing multiple containers for each microservice can increase the operational overhead.

- Running sidecars alongside each application container consumes additional resources such as memory and CPU.

Circuit Breaker Pattern

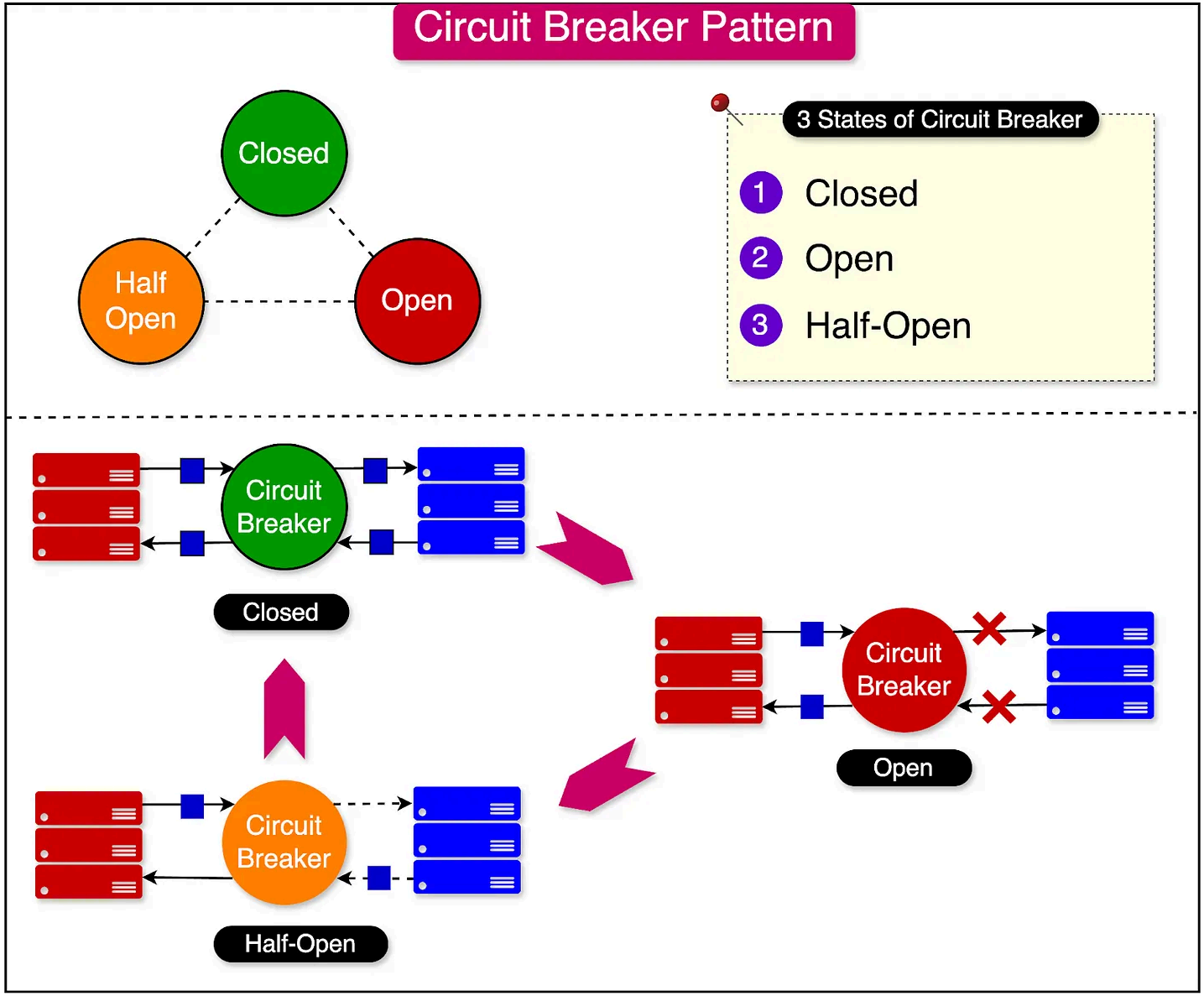
In a microservice architecture, communication between services is a fundamental aspect of fulfilling business requirements. However, this communication is not always reliable. Failures can occur due to various reasons, such as poor network connectivity, timeouts, or service unavailability.

Retrying failed calls is a common approach to handling such failures. In many cases, retrying can resolve the issue and allow the communication to proceed successfully. However, there are situations where multiple failures occur consecutively, and retrying becomes a waste of time and resources.

This is where the Circuit Breaker pattern comes into play.

The Circuit Breaker pattern introduces a proxy that sits between the communicating microservices. This proxy monitors the number of failures that occur within a specified period. If the failure count exceeds the defined threshold, the proxy prevents any further requests from going through.

The diagram below shows the various states of the Circuit Breaker pattern



The behavior of the Circuit Breaker pattern can be compared to an electric circuit breaker. Just like an electric circuit breaker automatically turns off when the power supply becomes unstable

or experiences issues, the Circuit Breaker pattern blocks communication between microservices when the failure threshold is reached.

The Circuit Breaker pattern operates in three distinct states:

- **Closed State:**
  - This is the initial state of the circuit breaker.
  - Microservices communicate normally, and the circuit breaker proxy monitors the number of failures within a defined period.
  - If the failure count exceeds the threshold, the circuit breaker transitions to the Open State.
- **Open State:**
  - In this state, the circuit breaker completely blocks communication between the microservices.
  - Instead of allowing requests to pass through, the circuit breaker can provide a graceful standard output to the client, indicating that the service is currently unavailable.
  - The Open State persists until a specific timeout period ends.
- **Half-Open State:**
  - After the timeout period in the Open state, the circuit breaker enters the Half-Open state.
  - In this state, the circuit breaker allows a limited number of requests to pass through to the target microservice.

- If these requests are successful, the circuit breaker assumes that the issue has been resolved and switches back to the Closed state.
- If the requests continue to fail, the circuit breaker reverts to the Open state and blocks communication again for the defined timeout period.

Benefits of Circuit Breaker Pattern

The Circuit Breaker pattern offers several benefits in a microservices architecture:

- By preventing excessive retries and blocking communication, the Circuit Breaker pattern helps improve the resilience of the system.
- The Circuit Breaker pattern enables fail-fast behavior.
- During the Open state, the circuit breaker can provide a fallback or default response to the client, allowing for a graceful degradation of functionality.
- It also provides valuable insights into the health and performance of the microservice.

Challenges

Some challenges or considerations with the Circuit Breaker pattern are as follows:

- It can be difficult to determine the optimal failure threshold and timeout periods.
- The use of external libraries to implement a circuit breaker may increase the complexity of the service.

Summary

In this article, we've explored the most popular microservices design patterns in great detail. We've understood the application of these patterns along with their benefits and challenges.

Let's recap the design patterns in brief:

- The Database Per Service pattern encourages the use of a separate database for each service.
- The API Gateway pattern supports the use of an API Gateway that acts as an intermediary between the client application and the backend microservices.
- Backends-for-Frontends (BFF) pattern proposes the use of separate backend services tailored specifically for each front-end client.
- The CQRS pattern separates the read and write operations of an application into distinct paths for more flexibility and better scalability.
- The Event Sourcing pattern is a powerful approach to handling data operations in a system driven by a sequence of events.
- The Saga pattern helps break down a business transaction into a sequence of local transactions, each handled by a different service.
- The Sidecar pattern allows the co-location of additional operational functionalities within an independent container.
- The Circuit Breaker pattern provides a mechanism to handle failures in microservice communication by defining a threshold for the number of failures between two services.

346 Likes · 26 Restacks

1 Comment

Write a comment...

Elvis Vusoh Elvis's Substack Aug 8

Hey I'm new on this platform. Please can someone help me, how does the sessions and classes work

LIKE (4) REPLY SHARE