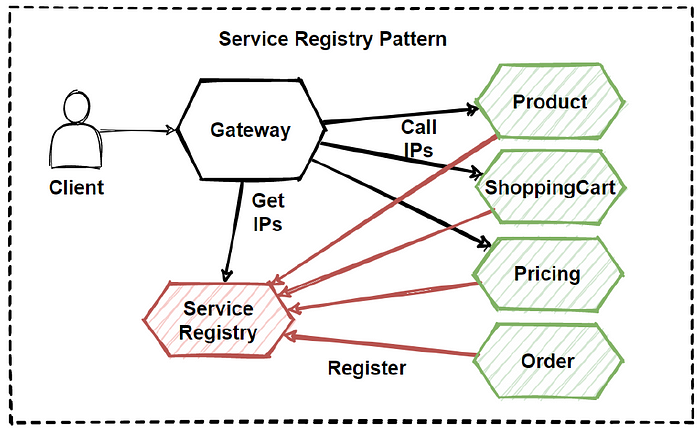
**Micro services Design Patterns**

[**https://learn.microsoft.com/en-us/azure/architecture/patterns/**](https://learn.microsoft.com/en-us/azure/architecture/patterns/)

1. **Service Registry**

* **Central service registry to discover Microservice by name.**
* Microservices can **register themselves with the registry**
* Tools - **Consul, Eureka (spring cloud)**
* Each service can be developed and deployed **independently, without hard-coding the endpoints** of other services in its code.
* The Service Registry Pattern enables the services to **locate each other dynamically**, making the system more flexible and resilient to changes.

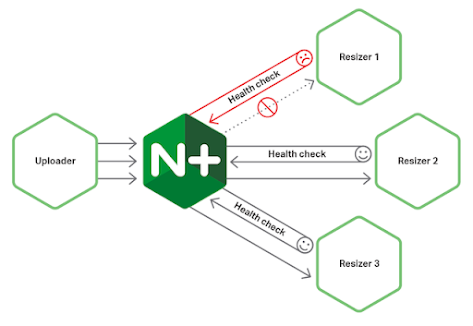


1. **Circuit Breaker**

* Prevent cascading failure by breaking circuit and enables applications to continue functioning when one or more services fail
* Used to **handle faults that may occur in a microservice**
* A circuit breaker acts as **a safety net between the client and the service**, protecting the client from failures in the service.
* The circuit breaker monitors the status of the service and, if the service is failing, open the circuit and prevent further requests sent to the service until the service has recovered
* Tools **- Netflix’s Hystrix or Spring Cloud Circuit Breaker**
* Benefits Fault tolerance, Resiliency, Improved performance (reduce time for failed or down operations), Reduced downtime, better user experience
* 3 Different states

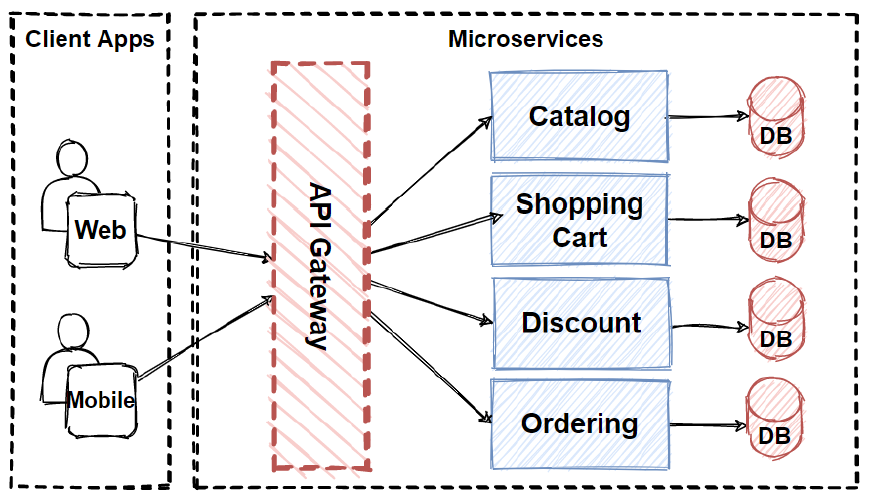
1. **Closed** (request routed to the application)
2. **Open** (Fails immediately)
3. **HalfOpen** (Limited number of requests allowed to pass, if successful, change the status to Closed state. Other than using a timer to determine when to switch to the Half-Open state, a circuit breaker can periodically ping the remote service or resource to determine whether it's become available again)

* **Use this pattern** - To prevent an application from trying to invoke a remote service or access a shared resource if this operation is highly likely to fail.
* **Not recommended** – 1. For handling access to local private resources in an application, such as in-memory data structure. 2. As a substitute for handling exceptions in the business logic of your applications.



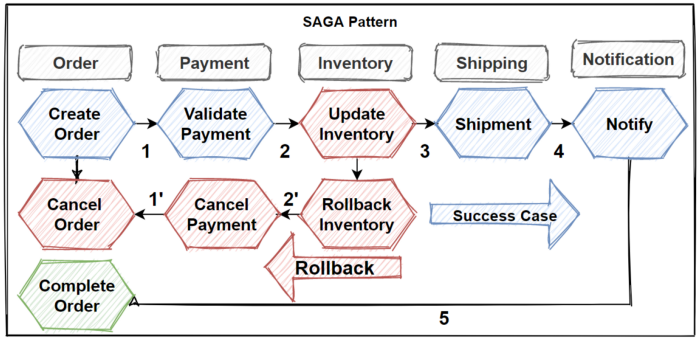
1. **API Gateway** - entry point for all incoming API requests (as a proxy)

* The main purpose **decouples the clients from the microservices**
* The additional layer of **security and governance, enforce policies**
* Can also **perform request/response transformation, rate limiting, authentication and authorization and caching**
* **Provide unified API that hides the internal details** of the Microservices and presents a **simpler and consistent interface to clients**



1. **Saga Pattern -** A way to manage transactions that involve multiple microservices.

* Ensure a series of transactions across multiple services are completed successfully. If not, roll back or undo all changes that have been made up to that point
* Consists of
* Sequence of local transactions each of which updates the state of a single services
* Corresponding set of compensating transactions that are used to undo the effects of the original transactions in case of a failure



Types of transactions

* ***Compensable transactions*** - reversed by processing another transaction with the opposite effect.
* A ***pivot transaction*** is the go/no-go point in a saga. If the pivot transaction commits, the saga runs until completion.
* ***Retryable*** *transactions* are transactions that follow the pivot transaction and are guaranteed to succeed.

Two types of workflow

**Choreography** - Exchange events without a centralised point of control. Simple, few participants and no coordination logic

**Orchestration –** good for complex workflows.

* **Use the Saga pattern** -
* Ensure data consistency in a distributed system without tight coupling.
* Roll back or compensate if one of the operations in the sequence fails.
* **Not recommended -**
* Tightly coupled transactions.
* Compensating transactions that occur in earlier participants.
* Cyclic dependencies.

1. **Event Sourcing Pattern** - for persisting and querying data.

Instead of storing the current state of an object, Event Sourcing

persists all events that occur in the application, **allowing the state of the object to be reconstructed at any point in time**.

In this pattern, every state change in the application is captured as an event and stored as a log of events.

**Benefits** Auditability, Scalability, Flexibility, Fault-tolerance

Event Sourcing provides an audit log of all changes that occur in the application.

1. **Command Query Responsibility Segregation (CQRS)**

Separates commands (write) and queries (read) into separate models, each with it's own database

Based on the idea that the models used for writing data are not the same as the models used for reading data

A **Command Model** receives commands from the client and writes to the database. The **Query Model** reads from the database and sends data to the client.

The pattern can be used **to improve the performance and scalability of a system**, as each model can be optimized for its specific task.

1. **Bulkhead Pattern**

Way of isolating different parts of a system so **that a failure in one part does not affect the rest of the system**. This pattern to - Isolate resources used to consume a set of backend services, especially if the application can provide some level of functionality even when one of the services is not responding

1. **Backends For Frontends (BFF)**
2. **Externalized Configuration**

Storing configuration data outside of the application code, making it easier to manage configuration changes

1. **Event-Driven Architecture**

Allows services to communicate with each other by emitting events.

1. **Database per Service**

Each service has its own database, which allows services to operate independently

**Data management patterns**

| Pattern | Summary |
| --- | --- |
| [Cache-Aside](https://learn.microsoft.com/en-us/azure/architecture/patterns/cache-aside) | Load data on demand into a cache from a data store |
| [CQRS](https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs) | Segregate operations that read data from operations that update data by using separate interfaces. |
| [Event Sourcing](https://learn.microsoft.com/en-us/azure/architecture/patterns/event-sourcing) | Use an append-only store to record the full series of events that describe actions taken on data in a domain. |
| [Index Table](https://learn.microsoft.com/en-us/azure/architecture/patterns/index-table) | Create indexes over the fields in data stores that are frequently referenced by queries. |
| [Materialized View](https://learn.microsoft.com/en-us/azure/architecture/patterns/materialized-view) | Generate prepopulated views over the data in one or more data stores when the data isn't ideally formatted for required query operations. |
| [Sharding](https://learn.microsoft.com/en-us/azure/architecture/patterns/sharding) | Divide a data store into a set of horizontal partitions or shards. |
| [Static Content Hosting](https://learn.microsoft.com/en-us/azure/architecture/patterns/static-content-hosting) | Deploy static content to a cloud-based storage service that can deliver them directly to the client. |
| [Valet Key](https://learn.microsoft.com/en-us/azure/architecture/patterns/valet-key) | Use a token or key that provides clients with restricted direct access to a specific resource or service. |

**Design and implementation pattern**

|  |  |
| --- | --- |
| **Pattern** | **Summary** |
| [Ambassador](https://learn.microsoft.com/en-us/azure/architecture/patterns/ambassador) | Create helper services that send network requests on behalf of a consumer service or application. |
| [Anti-Corruption Layer](https://learn.microsoft.com/en-us/azure/architecture/patterns/anti-corruption-layer) | Implement a façade or adapter layer between a modern application and a legacy system. |
| [Backends for Frontends](https://learn.microsoft.com/en-us/azure/architecture/patterns/backends-for-frontends) | Create separate backend services to be consumed by specific frontend applications or interfaces. |
| [CQRS](https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs) | Segregate operations that read data from operations that update data by using separate interfaces. |
| [Compute Resource Consolidation](https://learn.microsoft.com/en-us/azure/architecture/patterns/compute-resource-consolidation) | Consolidate multiple tasks or operations into a single computational unit |
| [Edge Workload Configuration](https://learn.microsoft.com/en-us/azure/architecture/patterns/edge-workload-configuration) | The great variety of systems and devices on the shop floor can make workload configuration a difficult problem. |
| [External Configuration Store](https://learn.microsoft.com/en-us/azure/architecture/patterns/external-configuration-store) | Move configuration information out of the application deployment package to a centralized location. |
| [Gateway Aggregation](https://learn.microsoft.com/en-us/azure/architecture/patterns/gateway-aggregation) | Use a gateway to aggregate multiple individual requests into a single request. |
| [Gateway Offloading](https://learn.microsoft.com/en-us/azure/architecture/patterns/gateway-offloading) | Offload shared or specialized service functionality to a gateway proxy. |
| [Gateway Routing](https://learn.microsoft.com/en-us/azure/architecture/patterns/gateway-routing) | Route requests to multiple services using a single endpoint. |
| [Leader Election](https://learn.microsoft.com/en-us/azure/architecture/patterns/leader-election) | Coordinate the actions performed by a collection of collaborating task instances in a distributed application by electing one instance as the leader that assumes responsibility for managing the other instances. |
| [Pipes and Filters](https://learn.microsoft.com/en-us/azure/architecture/patterns/pipes-and-filters) | Break down a task that performs complex processing into a series of separate elements that can be reused. |
| [Sidecar](https://learn.microsoft.com/en-us/azure/architecture/patterns/sidecar) | Deploy components of an application into a separate process or container to provide isolation and encapsulation. |
| [Static Content Hosting](https://learn.microsoft.com/en-us/azure/architecture/patterns/static-content-hosting) | Deploy static content to a cloud-based storage service that can deliver them directly to the client. |
| [Strangler Fig](https://learn.microsoft.com/en-us/azure/architecture/patterns/strangler-fig) | Incrementally migrate a legacy system by gradually replacing specific pieces of functionality with new applications and services. |

**Messaging patterns**

|  |  |
| --- | --- |
| **Pattern** | **Summary** |
| [Asynchronous Request-Reply](https://learn.microsoft.com/en-us/azure/architecture/patterns/async-request-reply) | Decouple backend processing from a frontend host, where backend processing needs to be asynchronous, but the frontend still needs a clear response. |
| [Claim Check](https://learn.microsoft.com/en-us/azure/architecture/patterns/claim-check) | Split a large message into a claim check and a payload to avoid overwhelming a message bus. |
| [Choreography](https://learn.microsoft.com/en-us/azure/architecture/patterns/choreography) | Have each component of the system participate in the decision-making process about the workflow of a business transaction, instead of relying on a central point of control. |
| [Competing Consumers](https://learn.microsoft.com/en-us/azure/architecture/patterns/competing-consumers) | Enable multiple concurrent consumers to process messages received on the same messaging channel. |
| [Pipes and Filters](https://learn.microsoft.com/en-us/azure/architecture/patterns/pipes-and-filters) | Break down a task that performs complex processing into a series of separate elements that can be reused. |
| [Priority Queue](https://learn.microsoft.com/en-us/azure/architecture/patterns/priority-queue) | Prioritize requests sent to services so that requests with a higher priority are received and processed more quickly than those with a lower priority. |
| [Publisher-Subscriber](https://learn.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber) | Enable an application to announce events to multiple interested consumers asynchronously, without coupling the senders to the receivers. |
| [Queue-Based Load Leveling](https://learn.microsoft.com/en-us/azure/architecture/patterns/queue-based-load-leveling) | Use a queue that acts as a buffer between a task and a service that it invokes in order to smooth intermittent heavy loads. |
| [Saga](https://learn.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga) | Manage data consistency across microservices in distributed transaction scenarios. A saga is a sequence of transactions that updates each service and publishes a message or event to trigger the next transaction step. |
| [Scheduler Agent Supervisor](https://learn.microsoft.com/en-us/azure/architecture/patterns/scheduler-agent-supervisor) | Coordinate a set of actions across a distributed set of services and other remote resources. |
| [Sequential Convoy](https://learn.microsoft.com/en-us/azure/architecture/patterns/sequential-convoy) | Process a set of related messages in a defined order, without blocking processing of other groups of messages. |
| [Pipes and Filters](https://learn.microsoft.com/en-us/azure/architecture/patterns/pipes-and-filters) | Break down a task that performs complex processing into a series of separate elements that can be reused. |
| [Sidecar](https://learn.microsoft.com/en-us/azure/architecture/patterns/sidecar) | Deploy components of an application into a separate process or container to provide isolation and encapsulation. |
| [Static Content Hosting](https://learn.microsoft.com/en-us/azure/architecture/patterns/static-content-hosting) | Deploy static content to a cloud-based storage service that can deliver them directly to the client. |
| [Strangler Fig](https://learn.microsoft.com/en-us/azure/architecture/patterns/strangler-fig) | Incrementally migrate a legacy system by gradually replacing specific pieces of functionality with new applications and services. |

**The Twelve Factors**

I. Codebase

One codebase tracked in revision control, many deploys

II. Dependencies

Explicitly declare and isolate dependencies

III. Config

Store config in the environment

IV. Backing services

Treat backing services as attached resources

V. Build, release, run

Strictly separate build and run stages

VI. Processes

Execute the app as one or more stateless processes

VII. Port binding

Export services via port binding

VIII. Concurrency

Scale out via the process model

IX. Disposability

Maximize robustness with fast startup and graceful shutdown

X. Dev/prod parity

Keep development, staging, and production as similar as possible

XI. Logs

Treat logs as event streams

XII. Admin processes

Run admin/management tasks as one-off processes

**Network latency in microservice architecture and how to reduce it? Circuit breaker**

resilience4j

**Closed** – if the call to the operation is unsuccessful the proxy increments this count. If the number of recent failures exceeds a specified threshold within a given time period, the proxy is placed into the Open state. At this point the proxy starts a timeout timer, and when this timer expires the proxy is placed into the Half-Open state.

**Open** – The request from the application fails immediately and an exception is returned to the application.

**Half open** – A limited number of requests from the application are allowed to pass through and invoke the operation. If these requests are successful, it's assumed that the fault that was previously causing the failure has been fixed and the circuit breaker switches to the Closed state. If any request fails, the circuit breaker assumes that the fault is still present so it reverts to the Open state and restarts the timeout timer to give the system a further period of time to recover from the failure.

The Half-Open state is useful to prevent a recovering service from suddenly being flooded with requests

Resilience4J sample  
resilience4j:

circuitbreaker:

instances:

order:

failureRateThreshold: 50

slowCallRateThreshold: 50

slowCallDurationThreshold: 500ms

permittedNumberOfCallsInHalfOpenState: 4

slidingWindowType: COUNT\_BASED

slidingWindowSize: 10

minimumNumberOfCalls: 4

waitDurationInOpenState: 30s

**Sliding Window** - to store and aggregate the outcome of calls

1. Count-based sliding window: This type aggregates the outcome of the last N calls. For example, if you set slidingWindowSize to 10, the circuit breaker calculates the failure rate and slow call rate based on the last 10 calls.
2. Time-based sliding window: In this approach, you sample requests over a specific time duration (e.g., 10 seconds)

minimumNumberOfCalls - minimum number of calls required before the circuit breaker evaluates its state

**AutoScaling**

Considerations:

Instances protected from scale-in won’t be terminated even during scale-in events.

It doesn’t protect against health check failures, Spot Instance interruptions, or manual terminations.

Detaching a protected instance loses its scale-in protection setting.

New instances inherit the group’s protection setting.

**Optimize Data Transfer**:

Use efficient data formats like Protocol Buffers or MessagePack instead of JSON. These formats are smaller and faster, reducing serialization and deserialization overhead1.

**Service Mesh**:

Implement tools like Istio or Linkerd. Service meshes manage and optimize communication between microservices, handling features like load balancing, retries, and circuit breaking1.

**Co-locate Services**:

* Kubernetes Scheduler and Service Affinity:

Ask the scheduler to place related services (deployed as Pods) close to each other. This reduces network latency by ensuring they run on the same node or in the same region/zone/rack.

* Run Services in the Same Pod:

Place two services in the same Kubernetes pod. Although they remain independently developed and released, they can communicate over localhost, improving performance.

* Co-locate Services in the Same Process:

Share the same process for multiple services. While this resembles a monolith, it balances development independence with occasional performance gains2.

**Edge Computing:**

Deploy microservices at the periphery, closer to end users. Processing data near its origin reduces round-trip latencies significantly3.

**Use caching**

How do you improve the performance of an API within the context of microservices?

**Reduce Memory Footprint**:

Keep microservices lightweight by minimizing their memory footprint and business logic. Smaller services tend to perform better.

Design each microservice with performance and security in mind from the start1.

**Design for Performance and Security**:

Tailor each microservice to specific business requirements.

Optimize for performance and security during the design phase.

**Cache OAuth Tokens:**

Caching OAuth tokens can reduce the overhead of generating tokens repeatedly. Avoid frequent calls to the OAuth token generation API by caching tokens.

**Use Asynchronous Non-Blocking Requests:**

Asynchronous requests improve concurrency and prevent blocking. Synchronous requests can cause performance bottlenecks for other services.

**Cache Database Queries:**

Cache responses for non-frequently changing data in the database.

This reduces hits to the database and prevents overloading it.

**Use Indexing in Database RDBMS Tables:**

Proper indexing improves query performance in relational databases. Optimize your database schema to support efficient queries.

**Use Database Connection Pooling:**

Reusing existing database connections reduces the overhead of creating new connections. Implement connection pooling to manage database connections efficiently.

**Leverage Server-Side Caching:**

Cache microservice responses based on input parameters.

This reduces execution time and improves overall performance.