**Microservice Architecture and Design Patterns**

This article introduces the mainstream common microservice patterns.

Microservices can have a positive impact on a business. Therefore, it is valuable to know how to deal with Microservice Architecture (MSA) and some microservice design patterns, some general goals or design principles of a microservice architecture. Here are four goals worth considering in a microservices architecture scenario.

1. Cost reduction: MSA will reduce the overall cost of designing, implementing and maintaining IT services

2. Accelerate the release speed: MSA will speed up the landing speed of the service from idea to deployment

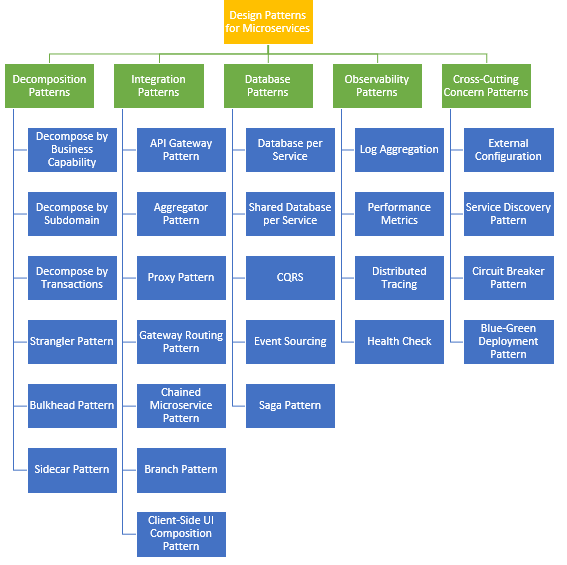
3. Enhanced resiliency: MSA will enhance the resiliency of our service network

4. Turn on visibility: MSA supports better visibility into services and networks

You need to understand a few design principles behind building a microservices architecture:

* Scalability
* Availability
* toughness
* flexibility
* independence, autonomy
* Decentralized Governance
* fault isolation
* automatic assembly
* Continuous Delivery with DevOps

Hearing the above principles, this can also bring some challenges and problems while the solution or system you implement is put into practice. These problems are also common in many solutions. These problems can be overcome by using the correct and matching design patterns. There are a few design patterns for microservices, which can be broadly grouped into five categories. Each class contains a number of specific design patterns. The following diagram illustrates these design patterns.



**decomposition mod**

**Break down by business function**

To put it bluntly, microservices are about applying the single responsibility principle and transforming services into loosely coupled ones. It can be broken down by business function. Define services that correspond to business functions. Business functionality is a concept from business architecture modeling. It is something a business does to create value. A business function often corresponds to a business object, such as:

* Order management is responsible for the order
* customer management is responsible for customers

**Break down by problem subdomain**

Decomposing an application by business function might be a good start, but you’ll eventually run into so-called “gods” that are harder to decompose. These classes will be common across multiple services. You can define some services that correspond to subdomains in Domain Driven [Design](https://mp.weixin.qq.com/mp/appmsgalbum?__biz=MzAwNjQwNzU2NQ==&action=getalbum&album_id=2248894963004522497&scene=173&from_msgid=2650353702&from_itemidx=1&count=3&nolastread=1#wechat_redirect) (DDD ). DDD refers to the problem space of an application — that is, the business — as a domain. A domain consists of multiple subdomains. Each subdomain corresponds to a different part of the business.

Subdomains can be divided into the following categories:

* Core — the core competencies of the business and the most valuable part of the application
* Support — related to the business but not a core competency. These can be implemented in-house or outsourced
* Generic — not specific to the business and ideally can be implemented using off-the-shelf software

An order management subdomain includes:

* Product catalog service
* Inventory Management Service
* Order Management Service
* Delivery management service

**Decomposition by transaction/**[**two-phase**](http://mp.weixin.qq.com/s?__biz=MzAwNjQwNzU2NQ==&mid=2650354402&idx=1&sn=9701a05e94df7026b87e52fe9b4294ff&chksm=83004600b477cf16d18f89680f8e2b3b8e59e9728cac2203c3520014dbdd9223af3710927a29&scene=21#wechat_redirect)**commit (2pc) pattern**

You can decompose services through transactions. Then, there will be multiple transactions in the system. The transaction coordinator is one of the important participants in distributed transaction processing. A distributed transaction consists of two steps:

* Prepare Phase — During this phase, all participants of the transaction are ready to commit and notify the coordinator that they are ready to complete the transaction
* Commit or Rollback Phase — During this phase, the transaction coordinator issues a commit or rollback command to all participants

The problem with 2PC is that it is quite slow compared to the runtime of a single microservice. Even if these microservices are running on the same network, transaction coordination between them does slow down the system, so this approach is usually not suitable for high load situations.

**Strangler Pattern**

The above three design patterns we have seen are all used to decompose greenfield (Greenfield) applications, but often 80% of the work we do is for grayfield (brownfield) applications, which are Some large monolithic applications (legacy codebases).

Strangler mode can solve such problems. It creates two separate applications running side by side in the same URI space. Over time, until eventually, the newly refactored application “kills” or replaces the old application, at which point the old monolithic application can be shut down. The steps of strangling an application are transformation, coexistence and elimination:

* Transform — Create a new parallel site using modern methods.
* Coexist — Keep existing sites around for a while. Redirect visits to an existing site to a new site to gradually implement the desired functionality.
* Eliminate — Remove legacy functionality from an existing site.

**Bulkhead Pattern**

Keep an application’s elements relatively isolated from the pool so that other applications will continue to work normally. This pattern is called “bayonet” because it resembles the segmented partitions of a ship’s hull. Divide service instances into different groups based on consumer load and availability requirements. This design helps isolate failures and allows users to maintain service for some consumers even during a failure.

**sidecar mode**

This pattern deploys an application’s components into a single processor container to provide isolation and encapsulation. It also allows applications to be composed of heterogeneous components and technologies. This mode is called Sidecar because it resembles a sidecar attached to a motorcycle. In this mode, the sidecar is attached to the parent application and provides functional support for that application. Sidecars also share the same lifecycle with the parent application and are created and exited with the parent application. The sidecar pattern is sometimes called the sidekick pattern, and it’s the last decomposition pattern we’ll list in this article.

**Integrated mode**

**API Gateway Pattern**

When an application is decomposed into multiple smaller microservices, here are some issues that need to be addressed:

* There are multiple calls to multiple microservices from different channels
* Need to handle different types of protocols
* Different consumers may require different response formats

API gateways help to solve many of the problems raised by microservice implementations, not just the ones mentioned above.

* API Gateway is the single entry point for any microservice invocation
* It can be used as a proxy service to route requests to related microservices
* It can aggregate the results and send back to the consumer
* The solution can create a fine-grained API for each specific type of client
* It can also convert protocol requests and respond
* It can also take responsibility for authentication/authorization of microservices.

**Aggregator Pattern**

Once the business function is broken down into several smaller logical pieces of code, it is necessary to consider how to coordinate the data returned by each service. This responsibility cannot be left to the consumer.

Aggregator pattern helps to solve this problem. It discusses how to aggregate data from different services and then send the final response to the consumer. There are two implementations here:

1. A composite microservice will call all the required microservices, merge the data, and then transform and synthesize the data before sending it back

2. An API gateway can also divide requests into multiple microservices and then aggregate data before sending it to consumers

If you want to apply some business logic, it is recommended to choose a composable microservice. Beyond that, API gateways are already the established de facto standard as a solution to this problem.

**proxy mode**

For API Gateway, we just use it to expose our microservices to the outside world. With the introduction of API Gateway, we were able to get some API-level features like security and categorizing APIs. In this example, the API Gateway has three API modules:

1. Mobile API, which implements the API of the FTGO mobile client 2. Browser API, which implements the API of JavaScript applications running in the browser 3. Public API, which implements some third-party developers need API

**Gateway routing mode**

API Gateway is responsible for routing requests. An API Gateway implements some API operations by routing requests to the appropriate service. When API Gateway receives a request, it queries a route map that specifies which service to route the request to. A route map can map an HTTP method and path to a service’s HTTP URL. This is the same as reverse proxy functionality provided by web servers like NGINX.

**Chained Microservice Pattern**

A single service or microservice will have multiple levels of dependencies, for example: Sale’s microservice depends on the Product microservice and the Order microservice. The chained microservices design pattern will help you serve merged request results.

After microservice-1 receives the request, the request then communicates with microservice-2 and possibly microservice-3. All these services are called synchronously.

**branch mode**

A microservice may need to get data from multiple sources including other microservices. The Forked Microservices pattern is a hybrid of the Aggregator and Chained design patterns and allows for simultaneous request/response processing from two or more microservices. The called microservice can be a chain of microservices. Branching patterns can also be used to invoke different chains of microservices or a single chain based on your business needs.

**Client UI composition mode**

When developing services by decomposing business functions/subdomains, the service responsible for the user experience must pull data from multiple microservices. In a monolithic world, there used to be only one call from the UI to the backend service, which retrieved all the data and then refreshed/committed the UI page. However, it is different now.

For microservices, we have to design the UI as a framework with multiple sections/areas of the screen/page. Each block will call a separate backend microservice to ingest the data.

Frameworks like AngularJS and ReactJS can help us do this easily. These screens are called Single Page Applications (SPAs).

Each team develops a client-side UI component, such as an AngularJS directive, that implements the page/screen area it serves. The UI team is responsible for implementing the page framework that builds the page/screen by combining multiple service-specific UI components.

**database schema**

When defining a database schema for a microservice, we need to consider the following:

1. Services must be loosely coupled. so that they can be independently developed, deployed and scaled

2. Business transactions may enforce invariants across multiple services

3. Some business transactions need to query the data of multiple services

4. For scalability reasons, databases must sometimes be replicable and shared

5. Different services have different data storage requirements

**One database per service**

To solve the above problems, a database must be designed for each microservice. It must be dedicated to that service only. It should only be accessible through the microservice’s API. Other services cannot access it directly. For example, for relational databases, we can use a separate private table for each service (private-tables-per-service), a separate database schema for each service (schema-per-service), or a separate database server for each service (database-server-per-service).

**Shared database between services**

We have already said that in microservices, it is ideal to have a separate database for each service. Using a shared database is an anti-pattern in microservices. However, if the application is a monolith and is trying to be split into microservices, denormalization is not so easy.

In a later stage, we can move to a schema of a set of until we have done that. Sharing a database between services is not ideal, but it is a practical solution for the above scenarios. Most people consider this an anti-pattern for microservices, but for grayfield applications, it’s a great start to breaking down the application into smaller logical parts. It’s worth mentioning that this should not be applied to greenfield applications.

**Command and Query Responsibility Separation (CQRS)**

Once a separate database-per-service is implemented, a query requirement naturally arises, which requires combining data from multiple services. However this is not possible. CQRS recommends splitting the application into two parts — the command side and the query side.

* The command-side handles create, update and delete requests
* The query side handles the query part by using a materialized view

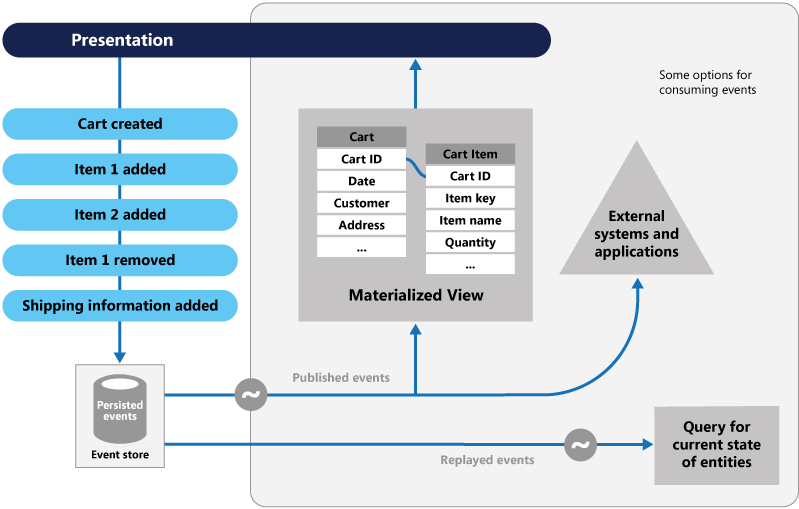
This is often used in conjunction with the event sourcing pattern, where events are created whenever any data changes. By subscribing to the event stream, we can keep the materialized view updated.

**event driven**

The vast majority of applications need to use data, and the typical approach is that the application maintains the current state. For example, in the traditional create, read, update and delete (CRUD) model, the typical data flow is to read data from storage. It also contains restrictions that lock data due to frequent use of transactions.

The event-driven pattern defines a method for handling data operations driven by a sequence of events, each recorded in an append-only store. Application code sends a series of events to the event store that imperatively describe each operation performed on the data, which are persisted to the event store. Each event represents a set of data changes (for example, AddedItemToOrder).

These events will be retained in an event store that acts as a system of record. Typical uses for events published by the event store are to maintain materialized views of entities when some action triggered by the application changes them, and to integrate with external systems. For example, a system might maintain a materialized view that populates a UI section of all customer orders. When the application adds a new order, adds or removes items from the order, and adds shipping information, events describing these changes are processed and used to update the materialized view. The figure below shows an overview of this mode.



**Saga mode**

When each service has its own database, and a business transaction spans multiple services, how can we ensure data consistency across services? Each request has a compensation request, which is executed when the request fails. This can be achieved in two ways:

* Choreography — Without central coordination, each service generates and listens to the other service’s events and decides if action should be taken. Choreography is a scheme that designates two or more parties. Neither party can control the other’s processes, or have any visibility into those processes, to coordinate their activities and processes to share information and value. Use choreography when coordination across control/visibility domains is required. Referring to a simple scenario, you can think of choreography as similar to network protocols. It specifies acceptable request and response patterns between parties. sage pattern
* Orchestration — An orchestrator (object) is responsible for the saga’s decision-making and business logic sequencing. At this point you can control all participants in the process. When they are all in one control domain, you can control the flow of that activity. Of course, this is usually when you’re assigned to an organization that has control over the business process. saga-pattern-orchestration

**observability pattern**

**log aggregation**

Consider a use case where an application contains multiple services. Requests typically span multiple service instances. Each service instance generates log files in a standard format. We need a centralized logging service that aggregates logs for each service instance.

Users can search and analyze logs. They can be configured to trigger alerts when certain messages appear in the log. For example, PCF has a log aggregator, which collects logs from each component of the PCF platform (router, controller, diego, etc.) on the application side. AWS Cloud Watch does the same.

**Performance**

As the service composition increases due to the introduction of a microservices architecture , it becomes even more critical to maintain monitoring of transactions so that these patterns can be monitored and alerts sent when problems occur.

Additionally, a metrics service is required to collect statistics about individual operations. It should aggregate metrics data from an application service, which it will use for reporting and alerting. Here are two models for aggregating metrics:

* Push — Service pushes metrics to metrics services such as NewRelic, AppDynamics
* Extract — Metrics service extracts metrics from services, such as Prometheus

**Distributed Link Tracing**

In a microservices architecture, requests typically span multiple services. Each service handles requests by performing one or more operations across multiple services. When troubleshooting, it is helpful to have a Trace ID, we can trace a request end-to-end.

The solution is to introduce a transaction ID. You can use the following methods:

* Assign a unique external request ID to each external request
* Pass the external request ID to all services handling the request link
* Include this external request ID in all log messages

**health examination**

After implementing a microservices architecture, there may be situations where services are started but cannot process transactions. Every service needs to have an API endpoint, such as /health, that can be used to check the health of the application. The API should check the state of the host, connections to other services/infrastructure, and any other specific logic.

**Cross-Cutting Concern Patterns**

**External configuration**

A service usually also calls other services and databases. For each environment like dev, QA, UAT, Prod, the URL of the API endpoint or some configuration properties may be different. Changes to any of these properties may require rebuilding and redeploying the service.

To avoid code modification, configuration can be used. Put all configuration out, including endpoint URLs and certificates. Applications should load them at startup or at runtime. These can be accessed by the application at startup or refreshed without restarting the server.

**Service Discovery Mode**

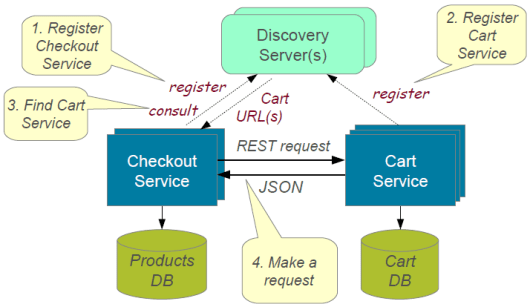
When microservices appeared, we needed to solve some problems in calling services.

With container technology, IP addresses can be dynamically assigned to service instances. Every time the address changes, the consumer service is interrupted and needs to be changed manually.

For consumer services, they must remember the URL of each upstream service, which becomes tightly coupled.

To do this, a service registry needs to be created that will keep the metadata of each producer service and the configuration of each service. Service instances should be registered with the registry on startup and unregistered on shutdown. There are two types of service discovery:

* Client: Example: Netflix Eureka
* Server: For example: AWS ALB

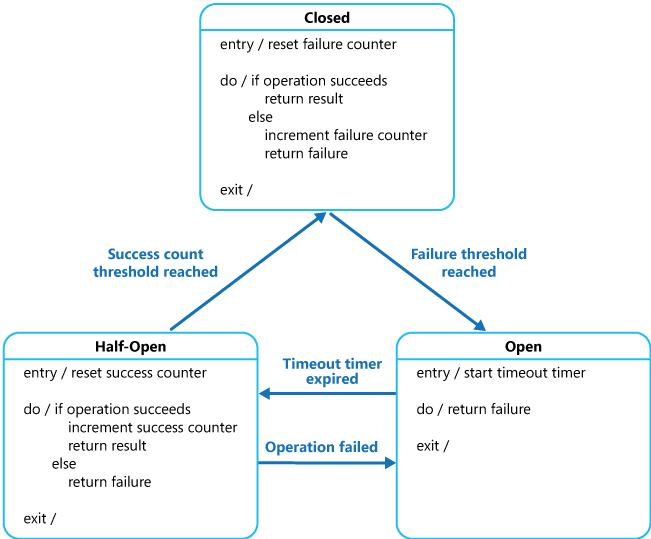


**Fuse Mode**

A service usually retrieves data by calling other services, and the downstream service may have died by this time. In this case, there are two problems: First, requests will continue to arrive at the dead service, exhausting network resources, and degrading performance. Second, the user experience will be bad and unpredictable.

Consumer services should call remote services through a proxy that behaves like a circuit breaker . When the number of consecutive faults exceeds the threshold, the circuit breaker will trip, and within the timeout period, all attempts to invoke the remote service will immediately fail. After the timeout expires, the circuit breaker will allow a limited number of test requests to pass.

If these requests are successful, the circuit breaker will resume normal operation. Otherwise, if a failure occurs, the timeout period will start over again. If certain operations have a high probability of failure, adopting this mode can help prevent the application from continuing to try to call remote services or access shared resources after the failure.



**Blue-green deployment model**

When using a microservices architecture, an application can be split into many microservices. If we were to stop all services and then deploy the improved version, the downtime would be significant and impact the business. Also, rolling back would be a nightmare. The blue-green deployment pattern avoids this.

Implementing a blue-green deployment strategy can be used to reduce or eliminate downtime. It does this by running two identical production environments, Blue and Green. Suppose Green is the existing active instance and Blue is the new version of the application. At any time, only one environment is active, and that active environment serves all production traffic. All cloud platforms offer options for implementing blue-green deployments.