**Microservices Patterns:**

source : <https://microservices.io/patterns/index.html>

**[1] Application architecture patterns**

*Context: You are developing a server-side enterprise application. It must support a variety of different clients including desktop browsers, mobile browsers, and native mobile applications. The application might also expose an API for 3rd parties to consume. It might also integrate with other applications via either web services or a message broker. The application handles requests (HTTP requests and messages) by executing business logic; accessing a database; exchanging messages with other systems; and returning a HTML/JSON/XML response. There are logical components corresponding to different functional areas of the application.*

*Problem: What’s the application’s deployment architecture?*

*Forces: (i) There is a team of developers working on the application (ii) New team members must quickly become productive (iii) The application must be easy to understand and modify (iv) You want to practice continuous deployment of the application (v) You must run multiple instances of the application on multiple machines in order to satisfy scalability and availability requirements (vi) You want to take advantage of emerging technologies (frameworks, programming languages, etc)*

Solution 1: **Monolithic Architecture**

1. a single Java WAR file. (ii) a single directory hierarchy of Rails or NodeJS code

Diagram

Description automatically generated

|  |  |
| --- | --- |
| Advantages | Drawbacks |
| Simple to develop - the goal of current development tools and IDEs is to support the development of monolithic applications | The large monolithic code base intimidates developers, especially ones who are new to the team. The application can be difficult to understand and modify. As a result, development typically slows down. |
| Simple to deploy - you simply need to deploy the WAR file (or directory hierarchy) on the appropriate runtime | Overloaded IDE - the larger the code base the slower the IDE and the less productive developers are.  Overloaded web container - the larger the application the longer it takes to start up. |
| Simple to scale - you can scale the application by running multiple copies of the application behind a load balancer | Continuous deployment is difficult.  Scaling the application can be difficult  Obstacle to scaling development. Obstacle to scaling development |

Solution 2: **Microservice Architecture**

Define an architecture that structures the application as a set of loosely coupled, collaborating services. Each service is:

1. Highly maintainable and testable - enables rapid and frequent development and deployment
2. Loosely coupled with other services - enables a team to work independently the majority of time on their service(s) without being impacted by changes to other services and without affecting other services
3. Independently deployable - enables a team to deploy their service without having to coordinate with other teams
4. Capable of being developed by a small team - essential for high productivity by avoiding the high communication head of large teams

Diagram

Description automatically generated

|  |  |
| --- | --- |
| Advantages | Drawbacks |
| Enables the continuous delivery and deployment of large, complex applications  (i) Improved maintainability (ii) Better testability (iii) Better deployability | Eliminates any long-term commitment to a technology stack.  (i) Developers must implement the inter-service communication mechanism |
| Each microservice is relatively small | Deployment complexity. |
| Improved fault isolation.  Eliminates any long-term commitment to a technology stack. | Increased memory consumption. The microservice architecture replaces N monolithic application instances with NxM services instances. If each service runs in its own JVM (or equivalent) |

**[2] Decomposition**

2.1 Context 1

The microservice architecture structures an application as a set of loosely coupled services. The goal of the microservice architecture is to accelerate software development by enabling continuous delivery/deployment.

Diagram

Description automatically generated

The microservice architecture does this in two ways:

(i) Simplifies testing and enables components to deploy independently

(ii) Structures the engineering organization as a collection of small (6-10 members), autonomous teams, each of which is responsible for one or more services

A service must be small enough to be developed by a small team and to be easily tested. A useful guideline from object-oriented design (OOD) is the Single Responsibility Principle (SRP). The SRP defines a responsibility of a class as a reason to change, and states that a class should only have one reason to change. *( In example: Invoice class does not have a responsibility to print itself)* . The application also be decomposed in a way so that most new and changed requirements only affect a single service. That is because changes that affect multiple services requires coordination across multiple teams, which slows down development. Another useful principle from OOD is the Common Closure Principle (CCP), which states that classes that change for the same reason should be in the same package.*(*”The classes in a component should be closed together against the same kind of changes. A change that affects a component affects all the classes in that component and no other components.”)

Problem: How to decompose an application into services?

* *The* ***architecture must be stable***
* ***Services must be cohesive****. A service should implement a small set of strongly related functions.*
* *Services must conform to the* ***Common Closure Principle*** *- things that change together should be packaged together - to ensure that each change affect only one service*
* *Services must be* ***loosely coupled*** *- each service as an API that encapsulates its implementation. The implementation can be changed without affecting clients*
* *A service should be* ***testable***
* *Each service be small enough to be developed by* ***a “two pizza” team****, i.e. a team of 6-10 people*
* ***Each team that owns one or more services must be autonomous****. A team must be able to develop and deploy their services with minimal collaboration with other teams.*

Solution 1: **Decompose by business capability**

Define services corresponding to business capabilities. It is something that a business does in order to generate value. A business capability often corresponds to a business object, e.g.

(i) Order Management is responsible for orders (ii) Customer Management is responsible for customers

Business capabilities are often organized into a multi-level hierarchy. For example, an enterprise application might have top-level categories such as Product/Service development, Product/Service delivery, Demand generation, etc.

Diagram

Description automatically generated

|  |  |
| --- | --- |
| Advantages | Disadvantages |
| Stable architecture since the business capabilities are relatively stable | How to identify business capabilities? Identifying business capabilities and hence services requires an understanding of the business. |
| Development teams are cross-functional, autonomous, and organized around delivering business value rather than technical features |  |
| Services are cohesive and loosely coupled |  |

Solution 2: **Decompose by subdomain**

Define services corresponding to Domain-Driven Design (DDD) subdomains. DDD refers to the application’s problem space - the business - as the domain. A domain is consists of multiple subdomains. Each subdomain corresponds to a different part of the business.

Diagram

Description automatically generated

*Advantages : (same as Decompose by business capability pattern )*

*Disadvantages: How to identify the subdomains? Identifying subdomains and hence services require an understanding of the business.*

2.2 Context 2

*Consider, the FTGO(Food To GO) application, which is an online food delivery application. A client of application creates an order by making an HTTP POST /orders request and expects a response, say, within 600ms. Because the FTGO application uses the microservice architecture, the responsibilities that implement order creation are scattered across multiple services. The POST request is first routed to the Order Service, which must then collaborate with the following services:*

*(i) Restaurant Service - knows the restaurant’s menu and prices*

*(ii) Consumer Service - knows the state of the Consumer that places the order*

*(iii) Kitchen Service - creates a Ticket, which tells the chef what to cook*

*(iv) Accounting Service - authorizes the consumer’s credit card*

*The Order Service could invoke each of these services using synchronous request/response.*

A picture containing diagram

Description automatically generated

However, a key drawback of using synchronous request/response is that it reduces availability. That’s because if any of the Order Sevice’s collaborators are unavailable, it will not be able to create the order and must return an error to the client.

An alternative approach is to eliminate all synchronous communication between the Order Service and its collaborators by using the CQRS and Saga patterns. The Order Service can use the CQRS pattern to maintain a replica of the restaurant menu’s and there by eliminate the need to synchronously fetch data from the Restaurant Service. It can validate the order asynchronously by using the Saga pattern. The Order Service creates an Order in a PENDING state and sends back a response to the POST /order.

A picture containing diagram

Description automatically generated

A key benefit of this approach is that it improves availability. The Order Service always respond to a POST /orders request even when one of the other services is unavailable. One drawback, however, of using a saga to complete the creation of the order is that the response to the POST doesn’t tell the client whether the order was approved. The client must find out by periodically invoking GET /orders/{orderId}.

Problem: How should a service collaborate with other services when handling a synchronous request?

1. The microservice architecture often distributes the responsibility of handling a request amongst multiple services
2. An operation is typically required to be highly available with a low response time
3. The availability of an operation is the product of the availabilities of the services that are invoked while handling a request
4. A service can retry a request to a failed collaborator but this increases response time.

Solution : **Self-contained Service**

Design a service so that it can respond to a synchronous request without waiting for the response from any other service.

1. One way to make a service self-contained is to implement needed functionality as a service module rather than a separate service. We could, for example, merge the Order Service and Restaurant Service.
2. Another way to make a service self-contained is for it to collaborate with other services using the CQRS and the Saga patterns. A self-contained service uses the Saga pattern to asynchronously maintain data consistency. It uses the CQRS pattern to maintain a replica of data owned by other services.

Diagram

Description automatically generated

Example

The Order Service in the FTGO application described earlier is an example of a self-contained service. The createOrder() operation, for example, queries a CQRS replica of data owned by the Restaurant Service to validate and price the order, and then initiates a saga to finish the creation of the order.

**Service per team**

**[3] Refactoring to microservices**

Problem : How do you migrate a legacy monolithic application to a microservice architecture?

Solution : **Strangler Application**

Modernize an application by incrementally developing a new (strangler) application around the legacy application. In this scenario, the strangler application has a microservice architecture.

Diagram

Description automatically generated

Problem : How do you prevent a legacy monolith’s domain model from polluting the domain model of a new service.

Solution : **Anti-corruption layer**

Define an anti-corruption layer, which translates between the two domain models.

**[4] Data management**

Context 1: Let’s imagine you are developing an online store application using the Microservice architecture pattern. Most services need to persist data in some kind of database. For example, the Order Service stores information about orders and the Customer Service stores information about customers.

Problem : What’s the database architecture in a microservices application?

(i) Services must be loosely coupled so that they can be developed, deployed and scaled independently

(ii) Some business transactions must enforce invariants that span multiple services. For example, the Place Order use case must verify that a new Order will not exceed the customer’s credit limit. Other business transactions must update data owned by multiple services.

(iii) Some business transactions need to query data that is owned by multiple services. For example, the View Available Credit use must query the Customer to find the credit Limit and Orders to calculate the total amount of the open orders.

Solution 1 : **Database per Service**

Keep each microservice’s persistent data private to that service and accessible only via its API. A service’s transactions only involve its database.

Diagram

Description automatically generated

The service’s database is effectively part of the implementation of that service. It cannot be accessed directly by other services.

*There are a few different ways to keep a service’s persistent data private. You do not need to provision a database server for each service. For example, if you are using a relational database then the options are:*

*Private-tables-per-service – each service owns a set of tables that must only be accessed by that service*

*Schema-per-service – each service has a database schema that’s private to that service*

*Database-server-per-service – each service has it’s own database server.*

*Private-tables-per-service and schema-per-service have the lowest overhead. Using a schema per service is appealing since it makes ownership clearer. Some high throughput services might need their own database server.*

Solution 2: **Shared database**

Use a (single) database that is shared by multiple services. Each service freely accesses data owned by other services using local ACID transactions.

Example:

Graphical user interface, text, application, email

Description automatically generated

Context 2:

You have applied the Database per Service pattern. Each service has its own database. Some business transactions, however, span multiple service so you need a mechanism to implement transactions that span services. For example, let’s imagine that you are building an e-commerce store where customers have a credit limit. The application must ensure that a new order will not exceed the customer’s credit limit. Since Orders and Customers are in different databases owned by different services the application cannot simply use a local ACID transaction.

Problem : How to implement transactions that span services?

Solution : **Saga**

Implement each business transaction that spans multiple services is a saga. A saga is a sequence of local transactions. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga. If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions.

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated**Context 3:**

You have applied the Microservices architecture pattern and the Database per service pattern. As a result, it is no longer straightforward to implement queries that join data from multiple services.

Problem : How to implement queries in a microservice architecture?

Solution : **API Composition**

Implement a query by defining an API Composer, which invoking the services that own the data and performs an in-memory join of the results.

Diagram

Description automatically generated

An API Gateway often does API composition.

Advantage : It a simple way to query data in a microservice architecture

Disadvantage : Some queries would result in inefficient, in-memory joins of large datasets.

**Context 4:**

Problem : How to implement a query that retrieves data from multiple services in a microservice architecture?

Solution : **Command and Query Responsibility Segregation (CQRS)**

Define a view database, which is a read-only replica that is designed to support that query. The application keeps the replica up to data by subscribing to Domain events published by the service that own the data.

Diagram

Description automatically generated

|  |  |
| --- | --- |
| Advantages | Disadvantages |
| * Supports multiple denormalized views that are scalable and performant * Improved separation of concerns = simpler command and query models * Necessary in an event sourced architecture | * Increased complexity * Potential code duplication * Replication lag/eventually consistent views |

**Context 5:**

A service often needs to publish events when it updates its data. These events might be needed, for example, to update a CQRS view. Alternatively, the service might participate in an choreography-based saga, which uses events for coordination.

Problem : How does a service publish an event when it updates its data?

Solution : **Domain event**

Organize the business logic of a service as a collection of DDD aggregates that emit domain events when they created or updated. The service publishes these domain events so that they can be consumed by other services.

**Context 6:**

A service command typically needs to update the database and send messages/events. For example, a service that participates in a saga needs to atomically update the database and sends messages/events. Similarly, a service that publishes a domain event must atomically update an aggregate and publish an event.

Problem : How to reliably/atomically update the database and send messages/events?

If the database transaction commits messages must be sent. Conversely, if the database rolls back, the messages must not be sent

Messages must be sent to the message broker in the order they were sent by the service. This ordering must be preserved across multiple service instances that update the same aggregate.

Solution : **Event sourcing**

A good solution to this problem is to use event sourcing. Event sourcing persists the state of a business entity such an Order or a Customer as a sequence of state-changing events. Whenever the state of a business entity changes, a new event is appended to the list of events. Since saving an event is a single operation, it is inherently atomic. The application reconstructs an entity’s current state by replaying the events.

Applications persist events in an event store, which is a database of events. The store has an API for adding and retrieving an entity’s events. The event store also behaves like a message broker. It provides an API that enables services to subscribe to events. When a service saves an event in the event store, it is delivered to all interested subscribers.

example : Customers and Orders is an example of an application that is built using Event Sourcing and CQRS. The application is written in Java, and uses Spring Boot. It is built using Eventuate, which is an application platform based on event sourcing and CQRS.

Diagram

Description automatically generated

**[5] Transactional messaging**

Context:

A service command typically needs to update the database and send messages/events. For example, a service that participates in a saga needs to atomically update the database and sends messages/events. Similarly, a service that publishes a domain event must atomically update an aggregate and publish an event.

Problem : How to reliably/atomically update the database and send messages/events?

If the database transaction commits messages must be sent. Conversely, if the database rolls back, the messages must not be sent

Messages must be sent to the message broker in the order they were sent by the service. This ordering must be preserved across multiple service instances that update the same aggregate.

Solution : **Transactional outbox**

A service that uses a relational database inserts messages/events into an outbox table (e.g. MESSAGE) as part of the local transaction. An service that uses a NoSQL database appends the messages/events to attribute of the record (e.g. document or item) being updated. A separate Message Relay process publishes the events inserted into database to a message broker.

Chart, scatter chart

Description automatically generated

Problem : **How to publish messages/events into the outbox in the database to the message broker?**

Solution : Transaction log tailing

Tail the database transaction log and publish each message/event inserted into the outbox to the message broker.

Diagram

Description automatically generated

Polling publisher

**[5] Testing**

Service Component Test

Consumer-driven contract test

Consumer-side contract test

**[6] Deployment patterns**

Multiple service instances per host

Service instance per host

Service instance per VM

Service instance per Container

Serverless deployment

Service deployment platform

**[7] Cross cutting concerns**

Microservice chassis

Service Template

Externalized configuration

**[8] Communication style**

Remote Procedure Invocation

Messaging

Domain-specific protocol

Idempotent Consumer

**[9] External API**

Context : Online store (which use microservices architecture pattern) implement product detail page develop multiple version of product detail page:

e.g. 1. HTML/Java script based UI for desktop and mobile browsers

2. Native android/iphone clients (clients interact with server via REST APIs)

3. And, expose product details via REST API for the use by 3rd party applications.

Problem : How do the clients of a Microservices-based application access the individual services?

(1. microservices typically provide fine-grained APIs. (ii) different clients need different data (iii) n/w performance is different for different type of clients (iv) no. of services instance and their locations changes dynamically i.e. host and ports(v) partitioning into services can change over time and should be hidden from clients.

Solution 1: **API Gateway**

Implement an API gateway that is the single-entry point for all the clients.

It can handle request in two ways : (1)some requests simply proxied/routed to the appropriate service. (ii) It can handles other requests by fanning out to multiple services.

Diagram, timeline

Description automatically generated

Solution 2 : **Backends for Frontends**

A variation of the above pattern is "Backends of Frontends" pattern. It defines a separate API gateway for each kind of client.

Diagram

Description automatically generated

**[10] Service discovery**

Context:

Services typically need to call one another. In a monolithic application, services invoke one another through language-level method or procedure calls. In a traditional distributed system deployment, services run at fixed, well-known locations (hosts and ports) and so can easily call one another using HTTP/REST or some RPC mechanism. However, a modern microservice-based application typically runs in a virtualized or containerized environments where the number of instances of a service and their locations changes dynamically.

Diagram

Description automatically generated

Problem : How does the client of a service - the API gateway or another service - discover the location of a service instance?

Each instance of a service exposes a remote API such as HTTP/REST, or Thrift etc. at a particular location (host and port)

The number of services instances and their locations changes dynamically.

Virtual machines and containers are usually assigned dynamic IP addresses.

The number of services instances might vary dynamically. For example, an EC2 Autoscaling Group adjusts the number of instances based on load.

Solution : **Client-side service discovery**

When making a request to a service, the client obtains the location of a service instance by querying a Service Registry, which knows the locations of all service instances.

Timeline

Description automatically generated

Problem : How does the client of a service - the API gateway or another service - discover the location of a service instance?

Solution : **Server-side service discovery**

When making a request to a service, the client makes a request via a router (a.k.a load balancer) that runs at a well-known location. The router queries a service registry, which might be built into the router, and forwards the request to an available service instance.

Problem : How are service instances registered with and unregistered from the service registry?

Solution: **Self Registration**

A service instance is responsible for registering itself with the service registry. On startup the service instance registers itself (host and IP address) with the service registry and makes itself available for discovery. The client must typically periodically renew its registration so that the registry knows it is still alive. On shutdown, the service instance unregisters itself from the service registry.

Service registry

3rd party registration

**[11] Reliability**

Context:

You have applied the Microservice architecture. Services sometimes collaborate when handling requests. When one service synchronously invokes another there is always the possibility that the other service is unavailable or is exhibiting such high latency it is essentially unusable. Precious resources such as threads might be consumed in the caller while waiting for the other service to respond. This might lead to resource exhaustion, which would make the calling service unable to handle other requests. The failure of one service can potentially cascade to other services throughout the application.

Problem : How to prevent a network or service failure from cascading to other services?

Solution: **Circuit Breaker**

A service client should invoke a remote service via a proxy that functions in a similar fashion to an electrical circuit breaker. When the number of consecutive failures crosses a threshold, the circuit breaker trips, and for the duration of a timeout period all attempts to invoke the remote service will fail immediately. After the timeout expires the circuit breaker allows a limited number of test requests to pass through. If those requests succeed the circuit breaker resumes normal operation. Otherwise, if there is a failure the timeout period begins again.

**[12] Security**

Problem: How to communicate the identity of the requestor to the services that handle the request?

(Services often need to verify that a user is authorized to perform an operation)

Solution : **Access token**

The API Gateway authenticates the request and passes an access token (e.g. JSON Web Token) that securely identifies the requestor in each request to the services. A service can include the access token in requests it makes to other services.

**[13] Observability**

Problem : How to understand the behaviour of an application and troubleshoot problems?

(Any solution should have minimal runtime overhead)

Solution: **Log aggregation**

Use a centralized logging service that aggregates logs from each service instance. The users can search and analyse the logs. They can configure alerts that are triggered when certain messages appear in the logs.

Solution 2 : **Application metrics**

Instrument a service to gather statistics about individual operations. Aggregate metrics in centralized metrics service, which provides reporting and alerting. There are two models for aggregating metrics:

push - the service pushes metrics to the metrics service

pull - the metrics services pulls metrics from the service

Problem : How to understand the behaviour of users and the application and troubleshoot problems?

(It is useful to know what actions a user has recently performed: customer support, compliance, security, etc.)

Solution: **Audit logging**

Record user activity in a database.

Problem : How to understand the behaviour of an application and troubleshoot problems?

(External monitoring only tells you the overall response time and number of invocations - no insight into the individual operations. Log entries for a request are scattered across numerous logs)

Solution: **Distributed tracing**

Instrument services with code that:

Assigns each external request a unique external request id

Passes the external request id to all services that are involved in handling the request

Includes the external request id in all log messages

Records information (e.g. start time, end time) about the requests and operations performed when handling a external request in a centralized service

Problem :How to understand the behaviour of an application and troubleshoot problems?

(Exceptions must be de-duplicated, recorded, investigated by developers and the underlying issue resolved. Any solution should have minimal runtime overhead)

Solution: **Exception tracking**

Report all exceptions to a centralized exception tracking service that aggregates and tracks exceptions and notifies developers.

Problem : How to detect that a running service instance is unable to handle requests?

(An alert should be generated when a service instance fails

Requests should be routed to working service instances)

Solution: **Health check API**

A service has an health check API endpoint (e.g. HTTP /health) that returns the health of the service. The API endpoint handler performs various checks, such as the status of the connections to the infrastructure services used by the service instance the status of the host, e.g. disk space application specific logic

A health check client - a monitoring service, service registry or load balancer - periodically invokes the endpoint to check the health of the service instance.

Problem : How to understand the behaviour of an application and troubleshoot problems?

(It useful to see when deployments and other changes occur since issues usually occur immediately after a change)

Solution: **Log deployments and changes**

Log every deployment and every change to the (production) environment.

**[14] UI patterns**

Context : You have applied the Microservice architecture pattern. Services are developed by business capability/subdomain-oriented teams that are also responsible for the user experience. Some UI screens/pages display data from multiple service. Consider, for example, an Amazon-style product detail page, which displays numerous data items including:

Basic information about the book such as title, author, price, etc.

Your purchase history for the book

...

...

Problem : How to implement a UI screen or page that displays data from multiple services?

Solution: **Server-side page fragment composition**

Each team developers a web application that generates the HTML fragment that implements the region of the page for their service. A UI team is responsible for developing the page templates that build pages by performing server-side aggregation (e.g. server-side include style mechanism) of the service-specific HTML fragments.

Problem : How to implement a UI screen or page that displays data from multiple services?

Solution: **Client-side UI composition**

Each team develops a client-side UI component, such an AngularJS directive, that implements the region of the page/screen for their service. A UI team is responsible implementing the page

API Gateway: (i) redirecting incoming requests (routing the request) (ii) security (iii) load balancing (server side) (check client side load balancing)

Eureka : naming service

configuration service: