Micro Service Architecture

Mehran Rivadeh

What are microservices?

- An architectural style that structures an application as a collection of services that are:
- Independently deployable
- Loosely coupled
- Organized around business capabilities
- Owned by a small team

Microservice Benefits

- The microservice architecture enables an organization to deliver large, complex applications:
- Rapidly,
- Frequently,
- Reliably and sustainably
- All necessity for competing and winning in today's world.

Rapid, frequent, reliable and sustainable delivery

- Process - DevOps as defined by the DevOps handbook
- Organization - a network of small, loosely coupled, cross-functional teams
- Architecture - a loosely coupled, testable and deployable architecture

Decomposition

- Decompose by business capability
- Decompose by subdomain
- Self-contained Service
- Service per team

Decompose by business capability

- Define services corresponding to business capabilities.
- A business capability is a concept from business architecture modeling.
- Business capabilities are often organized into a multi-level hierarchy.

Decompose by business capability(Examples)

- Product catalog management
- Inventory management
- Order management
- Delivery management

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.
- Subdomains can be classified as follows:
  - Core - key differentiator for the business and the most valuable part of the application
  - Supporting - related to what the business does but not a differentiator. These can be implemented in-house or outsourced.
  - Generic - not specific to the business and are ideally implemented using off the shelf software

Decompose by subdomain(Examples)

- Product catalog
- Inventory management
- Order management
- Delivery management

Self-contained service

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

Service per team

- Each service is owned by a team, which has sole responsibility for making changes.
- Ideally each team has only one service.

Data management

- Database per Service
- Shared database
- Saga
- Command-side replica
- API Composition
- CQRS
- Domain event
- Event sourcing

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.

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.

Saga

- Implement each business transaction that spans multiple services as 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.

Two ways of coordination

- There are two ways of coordination sagas:
  - Choreography - each local transaction publishes domain events that trigger local transactions in other services
  - Orchestration - an orchestrator (object) tells the participants what local transactions to execute

Command-side replica

- The solution consists of the following elements:
- Command service - the service that implements the command.
- Provider service - the service that owns the data that the command service needs
- Replica database - a read-only replica of the data from the provider service.
- The command service keeps the replica up to data by subscribing to domain events published by the provider service

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.

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.

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.

Transactional messaging

- Transactional outbox
- Transaction log tailing
- Polling publisher

Transactional outbox

- The solution is for the service that sends the message to first store the message in the database as part of the transaction that updates the business entities.
- A separate process then sends the messages to the message broker.

Transaction log tailing

- Tail the database transaction log and publish each message/event inserted into the *outbox* to the message broker.
- The mechanism for trailing the transaction log depends on the database

Testing

- Service Component Test
  - A test suite that tests a service in isolation using test doubles for any services that it invokes.
- Consumer-driven contract test
  - A test suite for a service that is written by the developers of another service that consumes it. The test suite verifies that the service meets the consuming service's expectations.
- Consumer-side contract test
- Verify that the client of a service can communicate with the service.

Deployment patterns

- Multiple service instances per host
- Service instance per host
- Service instance per VM
- Service instance per Container
- Serverless deployment
- Service deployment platform

Multiple service instances per host

- Run multiple instances of different services on a host (Physical or Virtual machine).
- There are various ways of deploying a service instance on a shared host including:
  - Deploy each service instance as a JVM process. For example, a Tomcat or Jetty instances per service instance.
  - Deploy multiple service instances in the same JVM. For example, as web applications or OSGI bundles.

Single Service Instance per Host

- Deploy each single service instance on its own host

Service Instance per VM

- Package the service as a virtual machine image and deploy each service instance as a separate VM

Service instance per container

- Package the service as a (Docker) container image and deploy each service instance as a container

Serverless deployment

- Use a deployment infrastructure that hides any concept of servers (i.e. reserved or preallocated resources)- physical or virtual hosts, or containers.
- The infrastructure takes your service's code and runs it.
- You are charged for each request based on the resources consumed.

Cross cutting concerns

- Microservice chassis
- Service Template
- Externalized configuration

Microservice chassis

- Create a microservice chassis framework that can be foundation for developing your microservices. The chassis implements:
    - Reusable build logic that builds, and tests a service. This includes, for example, Gradle Plugins.
    - Mechanisms that handle cross-cutting concerns. The chassis typically assembles and configures a collection of frameworks and libraries that implement this functionality.

Service Template

- Create a source code template that a developer can copy in order to quickly start developing a new service.
- A template is a simple runnable service that implements the required build logic and cross cutting concerns along with sample application logic.

Externalized configuration

- Externalize all application configuration including the database credentials and network location.
- On startup, a service reads the configuration from an external source, e.g. OS environment variables, etc.

Communication style

- Remote Procedure Invocation
- Messaging
- Domain-specific protocol
- Idempotent Consumer

Remote Procedure Invocation

- Use RPI for inter-service communication.
- The client uses a request/reply-based protocol to make requests to a service.

Messaging

- Use asynchronous messaging for inter-service communication. Services communicating by exchanging messages over messaging channels.
- There are several different styles of asynchronous communication:
  - Request/response - a service sends a request message to a recipient and expects to receive a reply message promptly
  - Notifications - a sender sends a message a recipient but does not expect a reply. Nor is one sent.
  - Request/asynchronous response - a service sends a request message to a recipient and expects to receive a reply message eventually
  - Publish/subscribe - a service publishes a message to zero or more recipients
  - Publish/asynchronous response - a service publishes a request to one or recipients, some of whom send back a reply

Domain-specific protocol

- Use a domain-specific protocol for inter-service communication.

Idempotent Consumer

- Implement an idempotent consumer, which is a message consumer that can handle duplicate messages correctly.
- Some consumers are naturally idempotent.
- Others must track the messages that they have processed in order to detect and discard duplicates.

Service discovery

- Client-side discovery
- Server-side discovery
- Service registry
- Self registration
- 3rd party registration

Client-side 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.

Server-side 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.

Service registry

- Implement a service registry, which is a database of services, their instances and their locations.
- Service instances are registered with the service registry on startup and deregistered on shutdown.
- Client of the service and/or routers query the service registry to find the available instances of a service.
- A service registry might invoke a service instance's health check API to verify that it is able to handle requests

Observability

- Log aggregation
- Application metrics
- Audit logging
- Distributed tracing
- Exception tracking
- Health check API
- Log deployments and changes

Log aggregation

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

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