Cloud Sales System high level architecture

Cloud Sales System is a solution for cloud sales, which will serve customers in Europe and APAC.

Cloud Sales System contains:

1. web portal where Crayons customers can log in and perform the actions.

2. web API where Crayons customers can integrate their systems, and perform the same operations without any user interaction. System to System.

Presumptions:

1) This application will use Microsoft Azure Resources

2) This application will use Azure SQL for databases

3) This application is a Multi-Tenant application, allowing multiple customers to share common infrastructure and resources

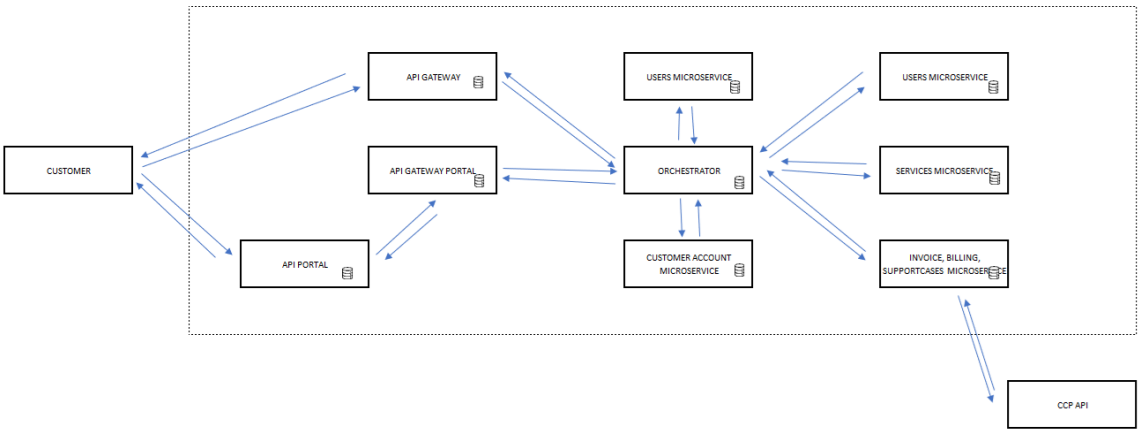
4) CCP has an endpoint that provides information about the CCP user

Technologies that would be used:

1) Docker for containerization and Kubernetes automating deployment, scaling, and management of containerized applications
2) Azure Cache for Redis for cache
3) AzureSql databases for database storage
4) Azure KeyVault would be leveraged to store secrets for the deployments process
5) Azure service bus for request queueing if needed, and other queing operations
6) OpenTelemetry on Azure for telemetry data
7) .net for development

High Level Architecture

Application uses gateway pattern and a microservices architecture with saga (Orchestration Based Saga)



Api Gateway

An API gateway is a data-plane entry point for API calls that represent client requests to target applications and services.

You can offload functionality from individual microservices to the gateway, which simplifies the implementation of each microservice by consolidating cross-cutting concerns into one tier. This approach is especially convenient for specialized features that can be complex to implement properly in every internal microservice, such as the following functionality:

- Authentication and authorization
- Service discovery integration
- Response caching
- Retry policies, circuit breaker, and QoS
- Rate limiting and throttling
- Load balancing
- Logging, tracing, correlation
- Headers, query strings, and claims transformation
- IP allowlisting

Saga Orchestrator

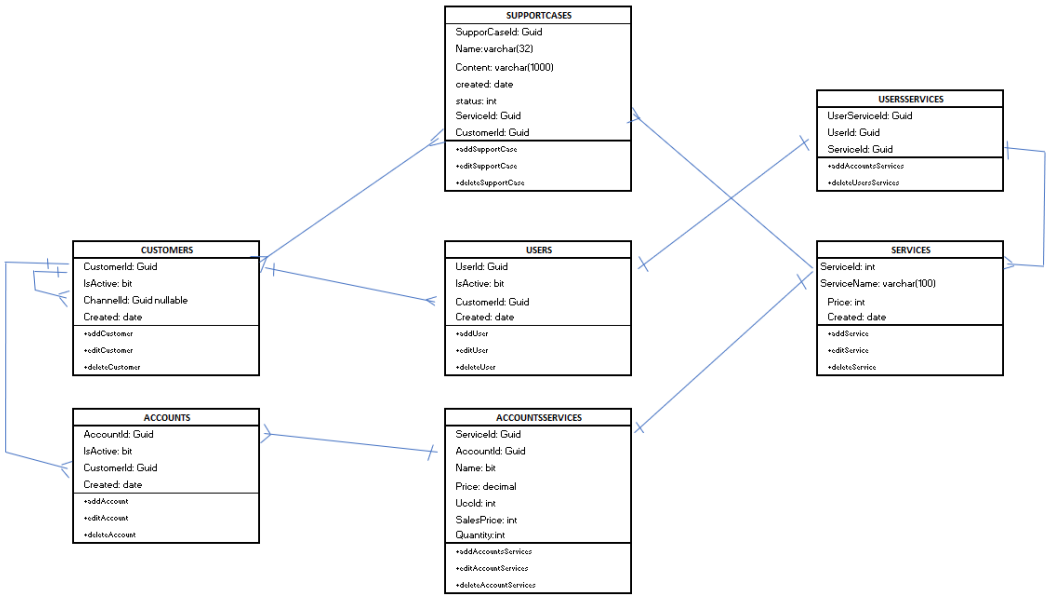
The saga orchestration pattern uses a central coordinator (*orchestrator*) to help preserve data integrity in distributed transactions that span multiple services. In a distributed transaction, multiple services can be called before a transaction is completed.

- The Saga Orchestrator is responsible for coordinating the execution of Saga activities.
- It defines the order of activities and handles any compensating actions when a failure occurs.
- The Orchestrator is a central component that manages the Saga's state and progress.

In our case each microservices would hold a database for a different type of information.

So we would have a microservice for customer, user, support cases, accounts, services

This is a simplified presentation of the database that can fit the description for this scenario.



In real world situations the scenario for Cloud Sales System would be much more complex. For example User would be presented by many tables (table with address information, table for billing information, previous activity and so on), so would Customers and so on. These could be databases separated logically into database for users, database for customers and so on.

The saga orchestration would give as a possibility to effect changes to those databases without the change to other information in other databases since they are dedicated to other microservices.

Back of the envelope calculations

Server Number Estimate:

Assumptions:

1. The average number of users: 50,000 per day
2. The average number of requests per user: 10
3. Average response time per request: 50ms

Total requests per day = 100,000 × 10 = **50,000** requests

Total requests per second = 500,000 / (24 × 60 × 60) ≈ **5.8**

*Number of servers required = RoundUp (Total requests per second / Maximum requests per second per server)*number of microservices*

Number of servers required = 9 servers

Database Storage Estimate

Assume a system has to Write to Read ratio= 1:100

And, there are 1M Write calls per day.

Let's assume each write request is 1000 B = 1KB

And, we have to store it for say 10 years

Storage per Day = 1M * 1KB = 1 GB (gigabyte)

Storage per Year = 1GB * 365 days = 360 GB

Storage for 10 Years = 360 GB * 10 = 3.6 TB (terabytes)

There will be some storage for audit, user, security, etc. Let's assume it will be 0.4 TB.

Total Storage for 10 yrs ~3.6 +0.4 = 4 TB.

Bandwith Estimate

Write to Read Ratio = 1:100
Write Request per day = 1 000000
Read Requests per day = 100 000000

Per request size: 1KB

Total Read Requests Size: 100 000000* 1 KB = 100 GB per day.

Bandwidth (per second) = 100 GB/day / (24*3600) = 1157.40741 KB/sec = ~1 MB per second.

And, Write is 10% of reading requests, so 1 MBPS /10 = 0.01 MBPS

Cache Estimate

Write to Read Ratio = 1:100
Write Request per day = 1000000
Read Requests per day = 100000000

Per request size: 1KB

Total Read Requests Size: 100000000* 1 KB = 100 gigabytes/ day.

Cache Estimates =20% of Read storage(20*100 GB)/100= **20 GB.**

Api Request Flow Diagram for Invoice Download