Name: Aditi Das
Student ID: 202001259

IT497
Date: 24th April 2023

1. Design a microservices architecture

Microservices architecture has gained widespread popularity as a modern approach for building cloud applications that are characterized by resilience, high scalability, independent deployability, and rapid evolution.

i. Azure compute option for microservices

Compute in the context of microservices architecture refers to the hosting model for the computing resources on which an application runs. There are two popular approaches for implementing microservices: a service orchestrator that manages services on dedicated nodes, and a serverless architecture using functions as a service. While other options exist, these approaches are proven methods for building microservices, and an application may utilize both approaches.

Service orchestrators are responsible for deploying and managing a set of services, including tasks such as scaling the number of service instances, service discovery, and load balancing network traffic. On the other hand, serverless architecture deploys code, and the hosting service takes care of placing that code onto a virtual machine and executing it.

When deciding between a service orchestrator approach and a serverless approach, various factors should be considered, including manageability, flexibility and control, portability, application integration, cost, and scalability.

ii. Interservice communication for microservices

This article delves into the challenges and trade-offs associated with communication between microservices. Effective and robust communication is crucial when multiple small services collaborate to achieve a unified business activity. The article explores the trade-offs between asynchronous messaging and synchronous APIs, as well as the challenges in designing resilient interservice communication. These challenges include ensuring resiliency, load balancing, distributed tracing, service versioning, and encryption with mutual authentication.

There are two primary messaging patterns for communication between microservices: synchronous communication, where the caller waits for a response from the receiver, and asynchronous message passing, where a service sends messages without waiting

for a response. The article provides a detailed examination of the advantages and disadvantages of each pattern.

Asynchronous messaging offers benefits such as reduced coupling, support for multiple subscribers, and failure isolation. However, it also presents challenges such as coupling with the messaging infrastructure, potential end-to-end latency, and associated costs. Ultimately, the choice between asynchronous messaging and synchronous APIs depends on the specific use case, system requirements, and communication challenges faced by the microservices architecture.

iii. APIs for microservices

In microservices architecture, the design of APIs plays a critical role as all communication between services occurs through messages or API calls. APIs need to be efficient to prevent excessive network traffic and should have well-defined semantics and versioning schemes to avoid breaking other services during updates. There are two types of APIs: public APIs and backend APIs, each with distinct requirements.

REST over HTTP is the most commonly used choice for public APIs, while backend APIs should consider network performance and explore alternatives such as gRPC, Apache Avro, and Apache Thrift. Factors such as efficiency, Interface Definition Language (IDL), serialization, framework and language support, compatibility, and interoperability should be taken into account.

When designing RESTful APIs, it is crucial to model the domain and avoid leaking internal implementation details. Additionally, the needs of different clients should be considered. The article provides additional resources for further information on this topic.

iv. API gateways in microservices

The article explores the concept of an API gateway in microservices architecture. Acting as a reverse proxy between clients and services, an API gateway routes requests from clients to services and performs cross-cutting tasks like authentication, SSL termination, and rate limiting. By sitting between clients and services, an API gateway helps address challenges associated with exposing services directly to clients, such as complex client code, coupling between the client and backend, and latency.

The article discusses three primary design patterns of API gateways, namely gateway routing, gateway aggregation, and gateway offloading. It also provides examples of

various gateway technologies such as reverse proxy servers, service mesh ingress controllers, Azure Application Gateway, Azure Front Door, and Azure API Management, along with their features, deployment, and management considerations.

Additionally, the article describes how to deploy Nginx or HAProxy to Kubernetes, offering insights into practical implementation of API gateways in a containerized environment.

v. Data considerations for microservices

The article delves into the challenges of managing data in a microservices architecture. With each service being responsible for its own data, ensuring data integrity and consistency becomes crucial. To avoid coupling between services, it is recommended that each service maintains its own data store and does not share it with other services. This approach often leads to polyglot persistence, where each service can choose the most suitable data storage technology based on its unique requirements.

The article also outlines various approaches to managing data in a microservices architecture, such as embracing eventual consistency, utilizing patterns like Scheduler Agent Supervisor and Compensating Transaction for transactions, storing only necessary data, employing event-driven architecture, and ensuring services are coherent and loosely coupled. Furthermore, the article includes an example of selecting data stores for a hypothetical drone delivery application, illustrating how different services may utilize different data storage technologies based on their specific needs.

vi. Design patterns for microservices

Microservices architecture is a software development approach that involves breaking down a large application into small, autonomous services that can be deployed independently, with the aim of enhancing application release velocity. However, this approach also presents its own set of challenges. In this article, various microservices design patterns are discussed, including Ambassador, Anti-corruption layer, Backends for Frontends, Bulkhead, Gateway Aggregation, Gateway Offloading, Gateway Routing, Sidecar, and Strangler Fig. These design patterns are employed to address these challenges and increase the resilience of the system by enabling services to function independently while also allowing for shared functionality and request aggregation.