Short Answer:

Answer the following questions with complete sentences in your own words. You are encouraged to conduct your own research online or through other methods before answering the questions. If you research online, please consult multiple sources before you write down your answers. You are expected to be able to explain your answers in detail (Provide examples to each question).

1. In your own word, please describe some of the advantages and disadvantages of a Monolithic Application.

A Monolithic Application is a software architecture where all components of the application are tightly coupled and built as a single, independent unit.

Advantages:

1. Simple Deployment: Since the application is built as a single unit, deployment is straightforward and does not require complex configuration.
2. Easy to Develop: With all components in the same codebase, developers can easily make changes and add new features without having to navigate between multiple codebases.
3. Cost-effective: Developing a monolithic application is often less expensive than developing a microservices architecture because there is less complexity and fewer moving parts.

Disadvantages:

1. Scalability Issues: As the application grows and receives more traffic, performance can suffer due to the tight coupling of components. Scaling a monolithic application often requires rearchitecting the entire application, which can be time-consuming and expensive.
2. Difficult to Test: Testing a monolithic application can be challenging due to the tight coupling of components and the large amount of code that needs to be tested.
3. Unnecessary Bloat: Monolithic applications can become bloated over time as new features are added, leading to slower performance and a more complex codebase that is difficult to maintain.

Outline

● Application Architecture

○ Monolithic

○ Microservices

● Microservices

○ Service Discovery

○ Service Communication

○ API Gateway

● Microservices Deployment

Monolithic

● We put everything in one application

Diagram

Description automatically generated

Monolithic

● This architecture has a number of benefits:

○ Simple to develop - the goal of current development tools and IDEs is to support the development of monolithic applications

○ Simple to deploy - we simply need to deploy the monolithic application as a whole

○ Simple to scale - we can scale the application by running multiple copies of the application behind a load balancer

Monolithic

● Ways to improve performance

○ Multi-Threading ○ Offline Processing

● But what if the number of requests sent to our application increases and our application reaches its limit?

Monolithic

 ● Scaling

 ○ Vertical: adding more power to an existing machine. For example, more powerful CPU or more RAM.

 ○ Horizontal: deploying multiple instances of our application.

Monolithic

● Vertical Scaling (not preferred)

○ More powerful machine is more expensive.

○ Always have a limit

 ■ What if the most powerful machine is still not powerful enough?

Monolithic

 ● Horizontal Scaling

○ Though cost money, but cheaper than vertical scaling. Multiple normal performance machines are cheaper than one high performance machine.

○ We can have as many instances as we want. The limit is the resources needed to make those machines.

Diagram

Description automatically generated

Horizontal Scaling

How does the client-side application know which server-side instance to request?

Load Balancer Scenario

● Imagine your web application is a table in a restaurant, and each table is an instance of the application that can serve a customer(handle a request).

 ● The load balancer in this example will be the host that waits at the restaurant’s entrance and leads the customers(request) to an empty table(idling application instance).

● The host(loader balancer) can bring up more tables(instances of the application) if the number of customers(requests) hits the current limit

Monolithic

● Load Balancer

● Load balancing – efficiently distributing incoming network traffic across a group of backend servers.

● Load balancer acts as a traffic distributer sitting in front of all your backend servers.

 ● It will route your client request to the most appropriate backend server which maximize speed, capacity utilization and ensures no server is over-worked. No impact on the performance.

Load Balancer

● Can be both software and hardware.

● Can be performed on both the client side and server side.

● Type: Application, Network, Classic

● Algorithm: Round-Robin, Weighted Round-Robin…

● Technologies: Nginx, HAProxy, Microsoft Azure Load Balancer.

Diagram

Description automatically generated

Monolithic

 ● However, once the application becomes too large, and the team grows in size, there are problems:

 ○ The large monolithic code base intimidates developers, especially ones who are new to the team

○ Overloaded IDE and web container

○ Might be scaling up the entire application just for one service.

○ Take a long time to deploy.

○ Requires a long-term commitment to a technology stack (Hard to adopt new technologies) ○ One error takes down the entire application.

 2. In your own word, please describe some of the advantages and disadvantages of a Microservice Application.

A Microservice Application is a software architecture where the application is divided into small, independent, and loosely coupled services that can be developed and deployed independently.

Advantages:

1. Scalability: Microservices can be deployed and scaled independently, making it easier to manage the growth of the application and ensure that it remains performant.
2. Resilience: If one microservice fails, the rest of the application can continue to function, making the overall system more resilient and less prone to downtime.
3. Improved Development Speed: Developers can work on individual microservices in isolation, which can improve development speed and efficiency.
4. Better Maintainability: Microservices can be developed and maintained by small, focused teams, which can improve the overall quality of the application and reduce the risk of costly bugs.

Disadvantages:

1. Complexity: Managing a microservice architecture can be complex and requires a solid understanding of service discovery, load balancing, and other key concepts.
2. Increased Network Latency: With microservices, requests must travel over the network, which can add latency and reduce performance compared to a monolithic architecture.
3. Increased Cost: Developing and maintaining a microservice architecture can be more expensive than a monolithic architecture, due to the increased complexity and number of components.
4. Testing Challenges: Testing microservices can be challenging due to the need to test each service individually and ensure that they work well together as a system.

 3. What is the purpose of using Service Discovery?

Service Discovery is a technique used in microservice architectures to manage the location of services. The purpose of Service Discovery is to allow microservices to communicate with each other by dynamically discovering the location of services at runtime.

Service discovery enables microservices to be located and called by other services in a dynamic and scalable way, even as services are added, removed, or moved within the architecture. This eliminates the need for hard-coded service locations and allows for more flexible and scalable deployment options.

With Service Discovery, microservices can register themselves with a central registry and announce their availability to other services. This allows other services to dynamically discover the location of available services and communicate with them as needed.

In summary, the purpose of Service Discovery is to manage the location of microservices within a distributed architecture and facilitate communication between services in a dynamic and scalable manner.

 4. How can microservices communicate with each other?

Microservices can communicate with each other through APIs (Application Programming Interfaces). An API is a set of protocols, routines, and tools for building software applications. In a microservice architecture, each microservice exposes a set of APIs that can be used by other microservices to interact with it.

There are several ways for microservices to communicate with each other through APIs:

1. RESTful APIs: Representational State Transfer (REST) is a popular approach for building APIs for microservices. RESTful APIs use HTTP methods (such as GET, POST, PUT, and DELETE) to perform operations on resources identified by URIs.
2. Remote Procedure Calls (RPCs): RPCs are a way for one microservice to call a remote procedure or function in another microservice. This allows microservices to communicate and exchange data in a more direct way, without having to go through the overhead of an HTTP request.
3. Message-Based Communication: Microservices can also communicate with each other by exchanging messages. This approach allows microservices to send and receive messages asynchronously, decoupling the sender and receiver and improving overall system resiliency.

In general, the choice of communication method depends on the specific requirements of the microservice architecture and the nature of the interactions between microservices. For example, message-based communication may be more appropriate for real-time or event-driven scenarios, while RESTful APIs may be more appropriate for stateful interactions.

Microservice

 ● What are Microservices?

○ Microservices, also known as microservice architecture, it’s an architectural style that structures an application as a collection of small services.

● Microservices are loosely coupled, services connect to each other on a service endpoint, and they don’t need to know the internal implementations of the other.

 ● Every Microservices are managed by independent teams that give the autonomy to decide when to release a feature or fix a bug without waiting for the entire team’s release cadence.

Microservices

● Here are some key characteristics of microservices:

 ○ Microservices are small, independent, and loosely coupled.

 ○ Each microservice has a separate codebase, which can be managed by a small development team.

○ Microservices are deployed independently. A team can update an existing microservice without rebuilding and redeploying the entire application.

○ Microservices are responsible for persisting their data or external state in their respective databases. Unlike the monolithic architecture, microservices don't share databases.

○ Microservices communicate with each other by using well-defined APIs. Internal implementation details of each service are hidden from other services.

○ Supports polyglot programming. For example, microservices don't need to share the same technology stack, libraries, or frameworks.

Microservices

● The cons of microservices

○ The implementation of Microservice is complex

○ The services are loosely coupled, so the data could be duplicated

○ API calls need to be resilient

○ Maintenance could be complicated

Diagram

Description automatically generated

Microservices

● What if a service needs to communicate with another service?

● We keep the locations (hostname and port number) of all services in every service?

● We keep the locations of all services in a central place?

Microservices

● Service Discovery/Registry: Keep track of services and service addresses and endpoints. ○ Steeltoe ○ Consul ○ Docker

Diagram

Description automatically generated

Service Communication

● Synchronous communication

○ REST ○ gRPC ○ SignalR

● Asynchronous communication

○ Azure Service Bus, Messaging(Rabbit MQ, Kafka)

■ Often preferred way for inter-service communication

Synchronous vs Asynchronous Communication

Graphical user interface, application

Description automatically generated

5. What is the purpose of using API Gateway?

An API Gateway is a component in a microservice architecture that acts as an intermediary between client applications and microservices. The purpose of an API Gateway is to provide a single entry point for external consumers of microservices and to manage cross-cutting concerns such as security, rate limiting, caching, and routing.

An API Gateway can provide several benefits, including:

1. Routing: The API Gateway can route incoming requests to the appropriate microservice based on the requested URL, reducing the burden on individual microservices to handle all incoming requests.
2. Load Balancing: The API Gateway can balance incoming requests across multiple instances of a microservice, improving performance and reliability.
3. Security: The API Gateway can enforce security policies, such as authentication and authorization, protecting sensitive data and ensuring that only authorized users have access to the microservices.
4. Caching: The API Gateway can cache frequently requested data, reducing the load on individual microservices and improving performance.
5. Monitoring: The API Gateway can monitor the health of microservices and provide insight into the performance and usage of the microservices.

In summary, the purpose of an API Gateway is to provide a centralized and unified entry point for external consumers of microservices, manage cross-cutting concerns, and improve the performance, security, and reliability of the microservice architecture.

API Gateway

● What if we have 10 services, and each service provides 10 entry points, so we will have 100 different URLs?

● How will the front end know which endpoint to call for a specific service?

API Gateway

● API gateway is the single-entry point for all clients

● Work as a proxy service to route a request to the backend microservices

 ● Aggregate the results to send back to the consumer

Diagram, timeline

Description automatically generated

API Gateway

● When it comes to choosing an API gateway for our services, there are a variety of options:

Application

Description automatically generated with medium confidence

Different type of services

 Core Services:

● Core services are the fundamental building blocks of a microservice-based application. These services provide the basic infrastructure for building and deploying microservices and are usually low-level.

 Composite Services:

 ● Composite services, on the other hand, are built on top of core services and provide additional functionality and capabilities to the microservice architecture. These services are designed to address specific business needs and provide higher-level functionality.

Microservice Architecture

Diagram

Description automatically generated

Microservices

● How are Microservices Packaged and Deployed?

 ○ Docker ○ Kubernetes

Microservices

● Containerization -- Docker

○ Containerization is an approach to software development in which an application or service, its dependencies, and its configuration (abstracted as deployment manifest files) are packaged together as a container image. You can test the containerized application as a unit and deploy them as a container image instance to the host operating system (OS).

○ Just as the shipping containers allow goods to be transported by ship, train, or truck. Software containers act as a standard unit of software deployment that can contain different code and its dependencies.

Docker

● What is docker?

○ Docker is an open-source project for automating the deployment of applications as portable, selfsufficient containers that can run in the cloud or on-premises. Docker is also a company that promotes and evolves this technology, working in collaboration with cloud, Linux, and Windows vendors, including Microsoft.

○ Docker containers can run anywhere on Azure: on-premises in the customer's datacenter, in an external service provider, or in the cloud. Docker image containers can run natively on Linux and Windows.

Docker Image

 ● What is an image

 ○ When a developer uses Docker, they create an app or service and package it and its dependencies into a container image. An image is a static representation of the app or service and its configuration and dependencies.

○ It's this image that, when run, becomes our container. The container is the in-memory instance of an image.

○ A container image is immutable. Once you've built an image, the image can't be changed. Since you can't change an image, if you need to make changes, you'll create a new image. This feature is our guarantee that the image we use in production is the same image used in development and QA.

Graphical user interface, application

Description automatically generated

Microservices Deployment

● Here are some general steps for deploying microservices with Docker:

1. Create a .NET Core application: Start by creating a .NET Core application that you want to deploy as a microservice. You can use Visual Studio or the .NET CLI to create a new .NET Core project.

2. Create a Dockerfile: A Dockerfile is a script that contains instructions for building a Docker image. Create a Dockerfile that includes instructions for building an image of your .NET Core microservice.

 3. Build a Docker image: Use the Docker CLI to build an image of your .NET Core microservice based on the Dockerfile you created.

4. Push the Docker image to a registry: Push the Docker image you built to a Docker registry, such as Docker Hub or a private registry.

 5. Deploy the Docker image: Deploy the Docker image to a Docker host, such as a virtual machine or a cluster of machines running in the cloud. You can use tools such as Docker Compose or Kubernetes to deploy the Docker image and manage the containers that run your .NET Core microservice

6. What is CAP principal?

CAP Theorem, also known as Brewer's Theorem, is a concept in distributed systems that states that it is impossible for a distributed system to simultaneously provide all three of the following guarantees:

1. Consistency: All nodes in the system see the same data at the same time.
2. Availability: Every request to the system receives a response, without guarantee that it contains the most recent version of the data.
3. Partition Tolerance: The system continues to function even when network partitions occur, meaning that communication between nodes is lost.

According to the CAP Theorem, a distributed system can provide at most two of these guarantees at any given time. In practice, this means that trade-offs must be made between consistency, availability, and partition tolerance, depending on the specific requirements of the system.

For example, in a highly available and partition-tolerant system, consistency may be relaxed in favor of ensuring that the system remains available, even when network partitions occur. In contrast, a system that prioritizes consistency over availability and partition tolerance may require that all nodes have consistent data, even if this means sacrificing some level of availability or partition tolerance.

In summary, the CAP Theorem is a fundamental concept in distributed systems that highlights the trade-offs between consistency, availability, and partition tolerance and helps to guide the design of distributed systems.

 7. Explain cascading failure in microservice and how to prevent it

Cascading failure in microservices refers to a chain reaction of failures across multiple microservices, triggered by a single failure in one of the microservices. This can cause a domino effect, leading to a failure in multiple interconnected microservices and causing the entire system to become unavailable or unstable.

For example, consider a scenario where a failure in one microservice affects another microservice that it relies on, causing that microservice to also fail. This, in turn, could cause a failure in another microservice that relies on the second microservice, and so on. The result can be a widespread failure of the entire microservice architecture.

There are several ways to prevent cascading failure in microservices:

1. Circuit breakers: Implementing circuit breakers in microservices can help to prevent cascading failures by automatically stopping requests to a microservice if it begins to fail. This helps to prevent further failures in other microservices that rely on the failed microservice.
2. Load balancing: Load balancing can help to distribute the load across multiple instances of a microservice, reducing the impact of a single failure and preventing cascading failures.
3. Monitoring and logging: Monitoring and logging can help to detect failures in microservices early, allowing teams to take proactive measures to prevent cascading failures.
4. Isolation and Decoupling: Isolating microservices from each other and ensuring that they are decoupled can help to prevent cascading failures. This means that the failure of one microservice does not directly affect other microservices and the system as a whole.
5. Health checks: Health checks can be used to monitor the health of microservices and detect failures before they cause cascading failures. This allows teams to take proactive measures to prevent failures from spreading to other microservices.

In summary, cascading failure in microservices is a potential issue that can cause widespread failure in a microservice architecture. To prevent cascading failures, organizations can implement circuit breakers, load balancing, monitoring and logging, isolation and decoupling, and health checks.

8. What is Circuit breaker and its states?

A circuit breaker is a design pattern used in microservice architecture to prevent cascading failures by automatically stopping requests to a microservice if it begins to fail. The purpose of a circuit breaker is to detect failures in a microservice and stop further requests from being sent to the failing microservice, preventing the failure from spreading to other microservices and causing a cascading failure.

A circuit breaker operates in three states:

1. Closed state: In the closed state, requests are sent normally to the microservice and the circuit breaker does not intervene.
2. Open state: If the circuit breaker detects a failure in the microservice, it transitions to the open state, where it stops requests from being sent to the failing microservice.
3. Half-open state: After a period of time, the circuit breaker transitions to the half-open state, where it sends a limited number of requests to the microservice to determine if it has recovered from the failure. If the microservice is able to successfully process the requests, the circuit breaker returns to the closed state. If the microservice continues to fail, the circuit breaker returns to the open state.

In summary, a circuit breaker is a design pattern used in microservice architecture to prevent cascading failures by automatically stopping requests to a failing microservice. The circuit breaker operates in three states: closed, open, and half-open, and transitions between these states based on the state of the microservice it is monitoring.