Micro Services

Microservice architecture is a design approach that structures an application as a collection of small, independent services, each serving a specific business capability. These services are loosely coupled and can be developed, deployed, and scaled independently.

A monolithic architecture is a design approach where an application is developed as a single unit and deployed as a single artifact. A microservices architecture, on the other hand, is a design approach where an application is developed as a collection of small, independent services that can be deployed, scaled, and maintained independently. In a monolithic architecture, changes to one part of the application can affect the entire application, while in a microservices architecture, changes can be isolated to a single service.

Benefits of a microservice architecture include:

- Improved scalability and flexibility
- Easier deployment and maintenance
- Improved fault tolerance
- Ability to use different technologies for different services
- Better team autonomy and ownership

Microservices design patterns:

1. An **API Gateway** is a single-entry point for all the client requests, which acts as an interface between the client and microservices. The gateway provides a unified interface for clients to interact with microservices and provides a way to manage and monitor the microservices.
2. **Circuit Breaker**: Circuit Breaker is a design pattern used to handle failures in a distributed system. It is used to prevent cascading failures in microservices by failing fast and providing fallback responses. Circuit Breaker monitors the health of microservices and, if necessary, opens the circuit and redirects the client to the fallback response.
3. **Service Registry**: Service Registry is a design pattern used to manage and discover microservices. It acts as a directory for microservices and provides a way to discover and locate the services. Service Registry also provides metadata about the services, such as their endpoints and their health status.
4. **Event-driven Architecture**: Event-driven architecture is a design pattern that emphasizes the production, detection, consumption, and reaction to events. In a microservices architecture, events are used to communicate between services. Services can produce events, consume events, and react to events. This pattern decouples services from each other and enables scalability and agility.
5. **Saga:** Saga is a design pattern used to manage long-running transactions in a distributed system. It is used to maintain data consistency in a distributed system by breaking the transaction into smaller, atomic, and compensating transactions. Saga keeps track of the status of each transaction and provides a way to roll back the entire transaction if any of the smaller transactions fail.
6. **Choreography**: Choreography is a design pattern used to coordinate the communication between microservices. In choreography, each microservice publishes events when it performs an action, and other microservices subscribe to these events to react to them. This pattern enables a decentralized communication between microservices and decouples them from each other.

These are just a few examples of the design patterns used in microservices architecture. There are many other patterns available, and each one has its own strengths and weaknesses.

Microservices design patterns

1. **Service Registry Pattern:** used to keep track of the services available in the system and their locations.
2. **API Gateway Pattern:** used to provide a single-entry point to the system, handling all requests from clients and routing them to the appropriate service.
3. **Circuit Breaker Pattern**: used to prevent cascading failures in a system by breaking the circuit when a service fails and handling the failure gracefully.
4. **CQRS (Command and Query Responsibility Segregation) Pattern**: used to separate the read and write operations in a system, allowing for better scalability and performance.
5. **Event Sourcing Pattern:** used to store events that have happened in the system, allowing for easy tracking and auditing of changes.
6. **Saga Pattern:** used to manage distributed transactions across multiple services, ensuring that all changes are either committed or rolled back.
7. **Sidecar Pattern**: used to attach a separate process to a microservice, providing additional functionality like service discovery, load balancing, and security.
8. **Strangler Pattern:** used to gradually replace a monolithic system with a microservices architecture, by gradually migrating functionality to the new system while keeping the old system operational.
9. **Bulkhead Pattern:** used to isolate different parts of a system, preventing a failure in one part from affecting the rest of the system.
10. **Gateway Aggregation Pattern:** used to combine multiple backend services into a single API gateway, simplifying the client's access to multiple services.

Domain-based Design (DBD) pattern

Domain-driven design (DDD) is a software development methodology that focuses on designing software systems based on a deep understanding of the business domain. It is especially useful in building microservices, where each microservice represents a distinct business capability.

The Domain-based Design (DBD) pattern is an extension of the DDD approach that uses domain modeling to drive microservice design. It involves decomposing a monolithic system into a set of microservices, each of which is based on a specific business domain.

Here is an example of how the DBD pattern works in practice:

Let's say we are building an e-commerce platform, and we have identified three distinct domains: Products, Orders, and Payments. Each of these domains has its own set of business rules, and they are all interdependent.

To implement the DBD pattern, we would create a separate microservice for each of the three domains, each with its own API and database. The Products microservice would be responsible for managing the product catalog, while the Orders microservice would handle order processing, and the Payments microservice would handle payment processing.

Each microservice would have a clearly defined boundary, and they would communicate with each other through well-defined APIs. This would ensure that each microservice could evolve independently, without affecting the others.

To ensure that the microservices remain aligned with the business domain, we would use domain modeling techniques to define the entities, aggregates, and value objects that make up each microservice. We would also use event-driven architecture to ensure that each microservice could respond to changes in the domain.

In summary, the DBD pattern is an approach to microservice design that is based on a deep understanding of the business domain. It involves decomposing a monolithic system into a set of microservices, each of which is based on a specific business domain and communicating through well-defined APIs.

Event-driven microservices

Event-driven microservices is an architectural approach where microservices communicate and collaborate through the exchange of events. In this pattern, services are decoupled and independent, and they interact by producing and consuming events asynchronously.

Here's how event-driven microservices work:

1. **Event Generation**: Each microservice can generate events based on certain triggers or actions within its own domain. These events represent meaningful occurrences or state changes in the service.

2. **Event Publishing**: When an event is generated, the microservice publishes the event to an event bus or message broker. The event bus acts as a central communication channel, enabling the distribution of events to interested consumers.

3. **Event Subscription**: Other microservices can subscribe to the event bus and express interest in receiving specific types of events. This allows them to be notified when relevant events occur.

4. **Event Processing**: When a microservice receives an event it is interested in, it processes the event and performs any necessary actions or updates its internal state accordingly. This can involve executing business logic, updating databases, or triggering other events.

Benefits of event-driven microservices:

1. **Decoupling**: Microservices are decoupled from each other, as they only interact through events. This enables independent development, deployment, and scaling of services without affecting the overall system.

2. **Scalability**: Services can scale independently based on the demand for events they generate or consume. This allows for better utilization of resources and improved performance.

3. **Flexibility**: The event-driven approach provides flexibility in adding or removing services, as long as they adhere to the event schema. New services can be easily integrated into the system by subscribing to relevant events.

4. **Resilience**: Event-driven microservices are inherently resilient to failures. If a service is temporarily unavailable, events can be buffered and processed later when the service is back online.

Overall, event-driven microservices enable loose coupling, scalability, and flexibility in building complex distributed systems, making them well-suited for handling asynchronous and event-based architectures.

Orchestration-based design pattern

Orchestration-based design pattern is one of the popular approaches to building microservices architectures. It involves using a central orchestrator service to manage the interactions between the various microservices. The orchestrator service is responsible for coordinating the requests and responses between the microservices, and it decides which microservice should handle a particular request based on the routing logic.

Here's an example of how the orchestration-based design pattern works in a microservices architecture:

Suppose we have an e-commerce application with multiple microservices. The microservices are as follows:

- Product Catalog Service
- Order Management Service
- User Management Service
- Payment Service
- Shipping Service

When a customer wants to purchase a product, they will first interact with the Product Catalog Service to select the product they want. Once they have selected a product, they will be directed to the Order Management Service, where they will be prompted to enter their shipping and billing information.

After the customer enters their information, the Order Management Service will send a request to the Payment Service to process the payment. If the payment is successful, the Order Management Service will then send a request to the Shipping Service to ship the product.

In an orchestration-based design pattern, **a central orchestrator service manages the interactions between the microservices**. The orchestrator service receives the customer's request and then decides which microservices to route the request to based on the routing logic.

In this example, the orchestrator service would route the request from the customer to the Product Catalog Service. Once the customer selects a product, the orchestrator service would route the request to the Order Management Service. The Order Management Service would then use the orchestrator service to send requests to the Payment Service and the Shipping Service.

The orchestrator service would monitor the responses from each of the microservices and then compile the responses into a single response that is returned to the customer.

Overall, the orchestration-based design pattern provides a centralized way to manage the interactions between the microservices. This approach allows for greater control and flexibility in managing the complex interactions between the microservices, making it a popular approach for building microservices architectures.

Bounded context in microservices

Bounded context is a concept in Domain-Driven Design (DDD) that defines a clear boundary within a larger system in which a particular domain model applies. It is the specific responsibility of a microservice within a larger system. Each bounded context encapsulates a specific domain and its functionality, and it can have its own data store, business logic, and user interface. This way, bounded contexts enable teams to independently develop, deploy, and scale their services without interfering with other services.

Bounded contexts help to maintain a clear separation of concerns, and they provide a common language and understanding of the system's components and their interactions. By defining clear boundaries between different domains, bounded contexts also enable better communication and collaboration among teams working on different parts of the system.

Session Management in Microservices

Sessions can be managed between microservices in several ways depending on the specific use case and the technologies being used. Some common approaches include:

1.Sticky sessions: In this approach, the load balancer routes all requests from a particular client to the same instance of a microservice. The microservice maintains the session state for that client, allowing it to retrieve it on subsequent requests. This approach can be effective for small-scale applications but can become problematic in large-scale deployments.

2.Distributed sessions: In this approach, session state is stored in a shared data store such as a database or a distributed cache. Each microservice instance can retrieve and update the session state as needed. This approach can be more scalable than sticky sessions but can also introduce issues such as data consistency and synchronization.

3.Token-based authentication: In this approach, the client sends a token with each request, which is used to authenticate and authorize the request. The token contains all the information needed to maintain the session state, including user ID, permissions, and other metadata. This approach can be simple and scalable but can also be less secure than other approaches.

4.OAuth: OAuth is a widely used protocol for authentication and authorization between applications. It allows users to log in to one application and grant permission for other applications to access their data. This approach can be more secure and scalable than other approaches but can also be more complex to implement.

Overall, session management in microservices can be complex and requires careful consideration of the specific requirements of the application and the technologies being used.

Microservices Interaction

Microservices interact with each other through APIs (Application Programming Interfaces). Each microservice exposes a set of APIs that can be used by other microservices or external systems to interact with it. When one microservice needs to communicate with another, it sends a request over the network to the API of the target microservice. The target microservice processes the request and sends a response back.

There are several communication patterns that can be used between microservices, such as **synchronous request-response, asynchronous messaging, and event-driven architecture**. The choice of communication pattern depends on the specific requirements of the system.

Microservices can also use service registries and discovery mechanisms to locate other microservices. When a microservice starts up, it registers itself with the service registry, providing information about its API and other relevant details. Other microservices can then query the service registry to discover the available services and their endpoints. This allows microservices to be added, removed, or scaled up or down dynamically, without requiring manual configuration updates.

Microservices Interaction

There are multiple ways to join microservices together. One common way is to use a service mesh, which is a dedicated infrastructure layer for managing service-to-service communication within a microservices architecture. Service meshes provide features such as traffic management, load balancing, service discovery, encryption, and observability.

One popular service mesh technology is Istio, which uses a sidecar proxy (Envoy) to intercept all network communication between microservices. The sidecar proxy is responsible for handling features such as traffic routing, circuit breaking, and security. Istio also provides a control plane for managing and configuring the service mesh.

Another way to join microservices together is through API gateways, which act as a front-end interface for all incoming requests to a microservices system. The API gateway is responsible for routing requests to the appropriate microservice and for handling cross-cutting concerns such as authentication, rate limiting, and caching.

Finally, microservices can also be joined together through direct service-to-service communication using REST or gRPC APIs, message queues such as Kafka or RabbitMQ, or other communication protocols such as GraphQL or WebSocket.

Asynchronous Messaging

Asynchronous messaging is a popular communication pattern for microservices. In this pattern, **microservices communicate with each other by exchanging messages through a message broker**. The message broker acts as an intermediary between the sender and receiver microservices, enabling them to communicate asynchronously.
Some of the benefits of using asynchronous messaging for microservices communication are:
1.Loose coupling: Asynchronous messaging decouples the sender and receiver microservices, allowing them to operate independently of each other. This makes it easier to change or replace a microservice without affecting the others.
2.Scalability: Asynchronous messaging enables horizontal scaling, where multiple instances of a microservice can process messages concurrently. This improves the throughput of the system, allowing it to handle a larger volume of requests.
3.Resilience: Asynchronous messaging provides fault tolerance, as messages can be stored in the message broker until the recipient microservice is available to process them. This ensures that messages are not lost or discarded, even if a microservice is temporarily unavailable.
Some of the popular message brokers used for microservices communication are Apache Kafka, RabbitMQ, and Apache ActiveMQ. These message brokers provide features such as message queuing, pub/sub messaging, and message filtering, which make it easier to implement asynchronous messaging for microservices.

Working of Microservices Communication pattern in asynchronous messaging using kafka

There are multiple ways to join microservices together. One common way is to use a **service mesh**, which is a dedicated infrastructure layer for managing service-to-service communication within a microservices architecture. Service meshes provide features such as traffic management, load balancing, service discovery, encryption, and observability.

One popular service mesh technology is Istio, which uses a sidecar proxy (Envoy) to intercept all network communication between microservices. The sidecar proxy is responsible for handling features such as traffic routing, circuit breaking, and security. Istio also provides a control plane for managing and configuring the service mesh.

Another way to join microservices together is **through API gateways**, which act as a front-end interface for all incoming requests to a microservices system. The API gateway is responsible for routing requests to the appropriate microservice and for handling cross-cutting concerns such as authentication, rate limiting, and caching.

Finally, microservices can also be joined together through direct service-to-service communication using REST or **gRPC APIs**, message queues such as Kafka or RabbitMQ, or other communication protocols such as GraphQL or WebSocket.

gRPC

RPC (gRPC Remote Procedure Call) is a modern, high-performance, open-source framework for building scalable, distributed systems. It was developed by Google and is now a Cloud Native Computing Foundation (CNCF) project. gRPC **enables client and server applications to communicate transparently, using remote procedure calls** (RPCs), making it easier to build connected systems.

gRPC **uses Protocol Buffers as its interface definition language** (IDL) and provides support for many programming languages, including C++, Java, Python, Go, Ruby, and many more. gRPC also **supports streaming responses**, which allows a server to send a continuous stream of data back to a client, rather than just a single response. This makes it ideal for building real-time applications, such as chat applications, or for streaming large amounts of data, such as video or audio streams.

gRPC also provides **support for bi-directional streaming**, which allows both the client and server to send streams of data to each other at the same time. This makes it possible to build more complex applications, such as collaborative editing applications or multi-player games.

Overall, gRPC is a powerful framework that makes it easier to build distributed systems that are fast, reliable, and scalable.

gRPC and REST APIs

REST and gRPC are two popular types of APIs used for building distributed systems. Here are some of the key differences between the two:

1. **Protocol:** REST (Representational State Transfer) uses HTTP/1.1 or HTTP/2 as its underlying protocol, while gRPC uses the newer HTTP/2 protocol.
2. **Payload format:** REST APIs typically use JSON or XML as the payload format, while gRPC **uses Protocol Buffers (protobufs)** by default. Protocol Buffers are a more compact, efficient, and faster way to serialize data than JSON or XML.
3. **Request and response types:** REST APIs use HTTP methods such as GET, POST, PUT, and DELETE to perform operations, **while gRPC APIs use the concept of Remote Procedure Calls (RPCs)** to define methods and operations. gRPC APIs use strongly typed requests and responses, while REST APIs use loosely typed messages.
4. **API contract:** REST APIs have a loose contract, meaning that the client and server are not tightly coupled and can evolve independently. gRPC APIs, on the other hand, have a strict contract defined by the protobufs schema. This means that any change to the API requires a new version of the protobufs schema.
5. **Streaming**: gRPC supports bidirectional streaming, where the client and server can send multiple messages in a single RPC call. REST APIs only support unidirectional streaming.
6. **Language support:** REST APIs are widely supported in many programming languages, while gRPC is primarily supported by Google and is best suited for building microservices in the Go and Java programming languages.

Overall, both REST and gRPC have their own strengths and weaknesses, and the choice between them depends on the specific requirements of the project. REST is a good choice for building simple, lightweight APIs that can be easily consumed by many different clients. gRPC is a good choice for building high-performance, scalable, and efficient APIs for microservices architectures.