Microservices

Microservices architecture is an approach to software development that structures an application as a collection of loosely coupled and independently deployable services

In a microservices architecture, the application is divided into several small services, each responsible for a well-defined task or functionality. These services communicate with each other through lightweight protocols like HTTP or messaging systems.

Each microservice is built around a specific business capability and can be developed using different technologies, programming languages, and frameworks, depending on its requirements.

Some key characteristics of microservices include:

1. Independently deployable: Each microservice can be developed, tested, and deployed independently of other services. This enables teams to work on different services concurrently, allowing for faster development and deployment cycles.

2. Loose coupling: Microservices communicate with each other through well-defined interfaces, usually via APIs. This loose coupling ensures that changes made to one service do not directly impact other services, promoting flexibility and resilience.

3. Scalability: Microservices can be individually scaled based on demand. This means that resources can be allocated to specific services that require more processing power or have higher traffic, optimizing resource utilization.

4. Resilience: Since microservices are decoupled, failures in one service do not cascade to the entire application. This enables fault isolation and improves the overall resilience of the system.

5. Technology diversity: Microservices architecture allows for the use of different technologies and frameworks for each service. Teams can choose the best-suited technology stack for a specific service's requirements, making it easier to adopt new technologies and experiment.

6. Continuous delivery: Microservices lend themselves well to continuous integration and continuous delivery (CI/CD) practices. Each service can be tested and deployed independently, allowing for faster iterations and more frequent releases.

While microservices offer several benefits, they also introduce challenges such as managing inter-service communication, data consistency, and deployment complexity.

Proper design, use of service orchestration or choreography patterns, and implementing robust monitoring and management systems are essential for successful microservices adoption.

Overall, microservices architecture enables the development of complex applications by breaking them down into smaller, manageable, and independent services that can be developed, deployed, and scaled independently.

It provides flexibility, scalability, and promotes faster development cycles, making it a popular choice for modern software architectures.

Microservices Draw Backs:

While microservices architecture offers numerous advantages, it's important to consider some of the potential challenges and disadvantages associated with this approach:

1. Increased complexity: Microservices introduce a higher level of complexity compared to monolithic architectures. With multiple services communicating and interacting with each other, the overall system complexity increases, requiring additional efforts for design, development, testing, deployment, and monitoring.

2. Distributed system challenges: As the system is distributed across multiple services, communication between services becomes crucial. Implementing reliable and efficient inter-service communication mechanisms, such as APIs or message queues, requires careful design and can introduce latency and overhead.

3. Operational overhead: Managing and monitoring a distributed system with multiple services can be more challenging compared to a monolithic architecture. Tasks such as deployment, service discovery, load balancing, and monitoring need to be addressed effectively. This may require additional tooling, infrastructure, and expertise.

4. Data consistency: Maintaining data consistency across multiple services can be complex. Each service typically has its own database, and ensuring data consistency and synchronization between services requires careful coordination and implementation of appropriate strategies such as event-driven architectures or distributed transactions.

5. Testing complexities: Testing microservices involves not only testing individual services but also verifying their interactions and integration. Creating comprehensive test environments, managing test data, and coordinating end-to-end testing can be more challenging in a distributed system.

6. Versioning and compatibility: As services are developed and deployed independently, managing versioning and compatibility between services becomes crucial. Changes in one service may require updates or adaptations in other dependent services. Careful versioning strategies and backward compatibility mechanisms should be in place to avoid service disruptions and compatibility issues.

7. Increased resource consumption: While microservices allow for scalability at the service level, it may result in increased resource consumption due to duplication of common functionalities across multiple services. Each service requires its own resources, including CPU, memory, and network, which may lead to higher infrastructure costs.

8. Organizational and communication challenges: Microservices often require teams to be organized around specific services, which can lead to challenges in communication, coordination, and knowledge sharing across teams. Ensuring effective collaboration and maintaining a shared understanding of the entire system can be more difficult.

SOA VS Microservices:

	Microservices	**SOA**
Architecture	Designed to host services which can function independently	Designed to share resources across services
Component sharing	Typically does not involve component sharing	Frequently involves component sharing
Granularity	Fine-grained services	Larger, more modular services
Data storage	Each service can have an independent data storage	Involves sharing data storage between services
Governance	Requires collaboration between teams	Common governance protocols across teams
Size and scope	Better for smaller and web-based applications	Better for large scale integrations
Communication	Communicates through an API layer	Communicates through an ESB
Coupling and cohesion	Relies on bounded context for coupling	Relies on sharing resources
Remote services	Uses REST and JMS	Uses protocols like SOAP and AMQP
Deployment	Quick and easy deployment	Less flexibility in deployment

Different interaction patterns between microservices.:

When it comes to microservices architecture, there are several interaction patterns that can be employed for communication between microservices. Here are some common interaction patterns:

1. Synchronous HTTP/REST: In this pattern, microservices communicate with each other over the HTTP protocol using RESTful APIs. One microservice makes an HTTP request to another microservice and waits for a response. This pattern is widely used and allows for easy integration between microservices.

2. Asynchronous Messaging: In this pattern, microservices communicate through message brokers or event buses. Instead of direct communication, a microservice publishes events or messages to a broker, and other microservices subscribe to those events and react accordingly. This pattern enables loose coupling between microservices and allows for scalability and fault tolerance.

3. Message Queues: Microservices can also communicate using message queues, where one microservice places messages or commands into a queue, and another microservice retrieves and processes them. This pattern ensures reliable message delivery and can handle load imbalances between microservices.

4. Publish/Subscribe: This pattern is similar to asynchronous messaging but with a focus on broadcasting events to multiple subscribers. Microservices can publish events to a topic, and any interested microservice can subscribe to that topic and receive the events. This pattern enables a decoupled and scalable architecture.

5. Remote Procedure Invocation (RPC): Microservices can use RPC frameworks to invoke procedures or methods on remote microservices as if they were local. The communication can be synchronous or asynchronous, and the RPC framework handles the underlying messaging and serialization. This pattern simplifies the remote communication between microservices.

6. Database Integration: Microservices can interact with shared databases or use the database-per-service pattern. In the former, multiple microservices access the same database, which can lead to coupling and dependency challenges. In the latter, each microservice has its own private database, and communication is done through explicit APIs. This pattern provides better isolation but requires careful management of data consistency.

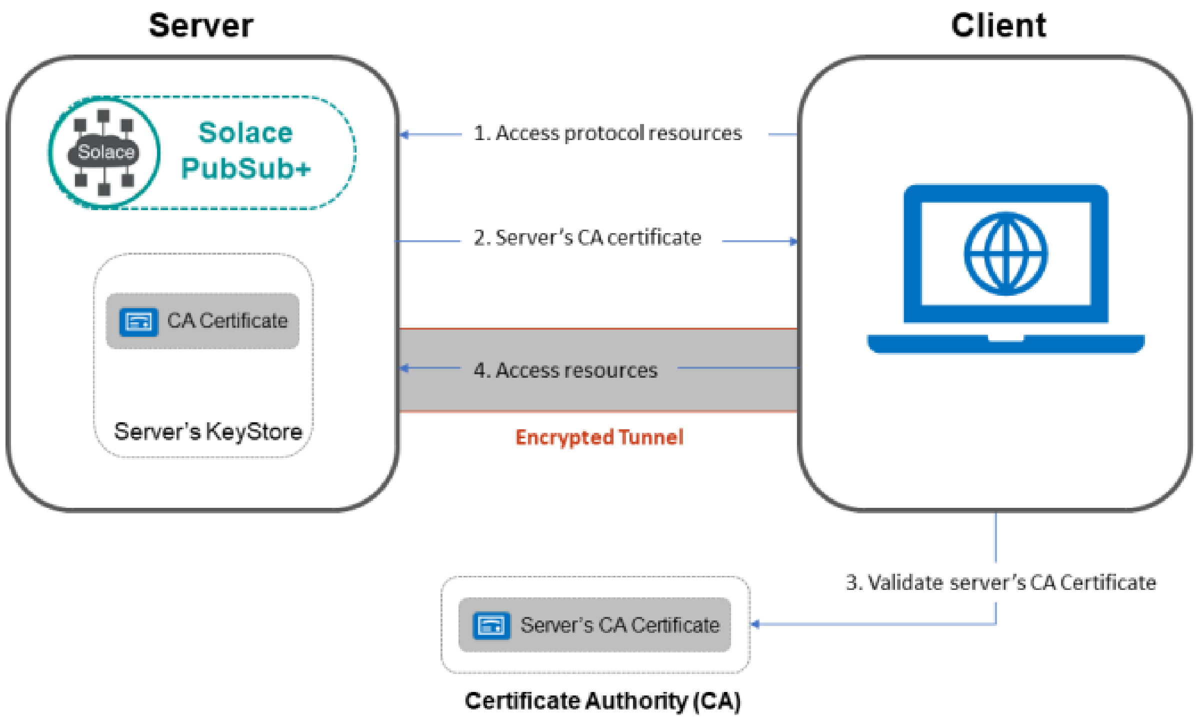
7. Caching: Microservices can employ caching techniques to improve performance and reduce latency. A microservice can cache frequently accessed data and respond to subsequent requests without involving other microservices. This pattern helps to optimize system performance.

Two way SSL Handshaking:

One-way SSL / Server Certificate Authentication:

In one-way SSL authentication (Server Certificate Authentication), only the client validates the server; the server does not verify the client application.

When implementing one-way SSL authentication, the server application shares its public certificate with the client.



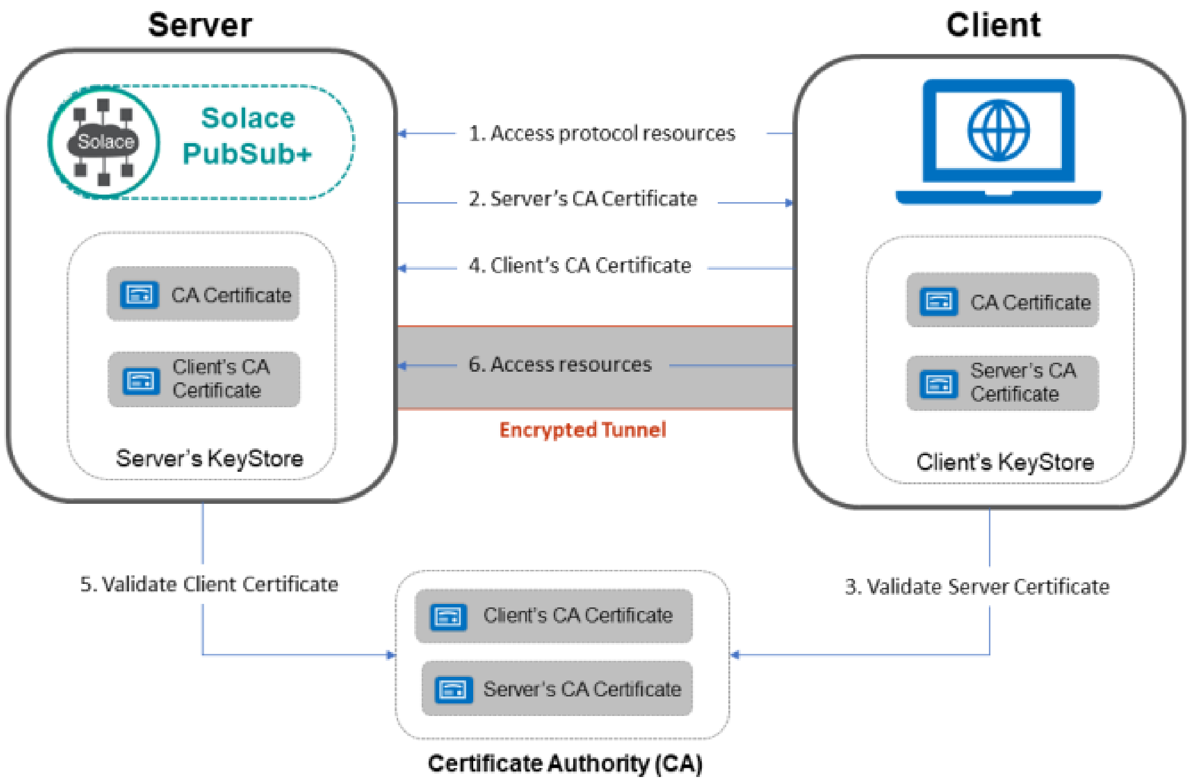
Command:

```
keytool -genkey -keyalg RSA -alias selfsigned -keystore keystore.jks -storepass password -
validity 360 -keysize 2048
```

Two way SSL:

In two-way SSL authentication, the client application verifies the identity of the server application, and then the server application verifies the identity of the client application. Both parties share their public certificates, and then validation is performed.

Two-way SSL authentication works with a mutual handshake by exchanging the certificates.


Server
Client
Solace
Solace PubSub+
1. Access protocol resources
2. Server's CA Certificate
4. Client's CA Certificate
CA Certificate
Client's CA Certificate
6. Access resources
Encrypted Tunnel
CA Certificate
Server's CA Certificate
Server's KeyStore
Client's KeyStore
5. Validate Client Certificate
3. Validate Server Certificate
Client's CA Certificate
Server's CA Certificate
Certificate Authority (CA)