SOA & MicroServices

- What are the main benefits of SOA?

Service-oriented architecture (SOA) is the organization of a system's components according to the services they provide. The provider of the service and the requester are loosely coupled within the architecture, allowing the services to operate independently of the platform on which they'll be used. It's an approach to designing software in which components are arranged according to business processes.

- SOA helps create greater alignment between IT and line of business while generating more flexibility – IT flexibility to support greater business flexibility. Your business processes are changing faster and faster and global competition requires the flexibility that SOA can provide.
- SOA can help you get better reuse out of your existing IT investments as well as the new services you're developing today. SOA makes integration of your IT investments easier by making use of well-defined interfaces between services. SOA also provides an architectural model for integrating business partners', customers' and suppliers' services into an enterprise's business processes. This reduces cost and improves customer satisfaction.

- **How can you achieve loose coupling in SOA**

One strategy for achieving loose coupling is to use the service interface (the WSDL for a SOAP Web Service) to limit this dependency, hiding the service implementation from the consumer. Loose coupling can be addressed by encapsulating the service functionalities in a manner that limits the impact of changes to the implementation on the service interface.

However, at some point you will need to change the interface and manage versioning without impacting service consumers, in addition to managing multiple security constraints, multiple transports, and other considerations.

- **Are web services and SOA the same?**

There are some key differences between Web services and SOA. Web services define a web technology that can be used to build applications that can send /receive messages using SOAP over HTTP. However, SOA is an architectural model for implementing loosely coupled service based applications. Web services can be used to implement SOA applications. Even though web service approach to SOA has become very popular, it is only a single method of implementing SOA. SOA can be implemented using any other service-based technology (e.g. CORBA and REST).

SOA & MicroServices

- **What is a reusable service?**

It is an autonomous, reusable, discoverable, stateless functionality that has the necessary granularity, and can be part of a composite application or a composite service.A reusable service should be identified with a business activity described by the service specifications (design-time contract).

A service's constraints, including security, QoS, SLA, usage policies, may be defined by multiple run-time contracts, multiple interfaces (the WSDL for a SOAP Web Service), and multiple implementations (the code).

A reusable service should be governed at the enterprise level throughout its entire lifecycle, from design-time through run-time. Its reuse should be promoted through a prescriptive process, and that reuse should be measured.

- **What are the disadvantages of SOA?**

Disadvantages of Service oriented architecture

Extra overload:

In SOA, all inputs are validated before it is sent to the service. If you are using multiple services then it will overload your system with extra computation.

High cost:

SOA is costly in terms of human resource, development, and technology.

High bandwidth server:

As some web service sends and receives messages and information frequently so it easily reaches a million requests per day. So it involves a high-speed server with a lot of data bandwidth to run a web service.

- **What is ESB and where does it fit in?**

An enterprise service bus (ESB for short) refers to software architecture that allows for the integration of enterprise applications and services, such as middleware infrastructure platforms.

SOA & MicroServices

The best way to think of an ESB is to compare it to a router. It provides the connections between applications that need to communicate with one another. Businesses use ESBs in enterprise application integration.

At the beginning of the 21st century, service-oriented architecture (SOA) became popular. In fact, SOA became the backbone of many an organization's architectural strategies. It was realized in the form of web services, which have become a significant component in implementing software solutions today.

While the interactions between services can be implemented with point-to-point architecture, such an approach creates needless complexity. It is monolithic, not reusable, and does not offer business functionality.

- **In SOA do we need to build a system from scratch?**

No. If you need to integrate or make an existing system as a business service, you just need to create loosely coupled wrappers which will wrap your custom systems and expose the systems functionality in a generic fashion to the external world.

- **What is the most important skill needed to adopt SOA ?technical or cultural?**

Surely cultural. SOA does require people to think of business and technology differently. Instead of thinking of technology first (e.g., If we implement this system, what kinds of things can we do with it?), practitioners must first think in terms of business functions, or services (e.g., My company does these business functions, so how can I set up my IT system to do those things for me most efficiently?).

It is expected that adoption of SOA will change business IT departments, creating service-oriented (instead of technology-oriented) IT organizations.

- **List down the advantages of Microservices Architecture.**

Benefits of Microservices Architecture

Let's take a detailed look at the benefits of microservice architecture.

Isolation

SOA & MicroServices

Microservices are profitable due to their isolation and resilience. If one of the components fail, developers have the option to use another service and the application will continue to run independently. This way, engineers can build and deploy services without the need to change the whole app.

Scalability

With the architecture of microservices based on small components, it's easier for development teams to scale up or down following the requirements of a specific element. Isolation allows apps to run correctly when massive changes are happening. Microservices prove to be the perfect approach for companies working with various platforms and devices.

Productivity

Part of functionality associated with microservice architecture is the ability to easily understand when compared to an entire monolithic app. If you plan to expand your development team, microservices are a better pick.

Flexibility

The microservice approach lets developers choose the right tools for the right task. They can build each server utilizing a language or framework they need without affecting the communication between microservices.

Faster project development

Microservices work independently, so you don't have to change the codebase in order to modify the features. You can change one component, test, and then deploy it individually. In turn, you will deliver the app faster.

Evolutionary

Microservice architecture is a perfect choice for developers unable to predict the kinds of devices the app is going to run on. Developers can provide fast and controlled upgrades through not slowing down or stopping the apps.

While microservices offer the opportunity for better productivity and tool selection, cons include teams using different coding languages, frameworks, libraries. This can paralyze the team if they're not ready for such independence. But if you're working on a large-scale complex app, microservice architecture is the perfect choice.

- **What are the best practices to design Microservices?**

The following are some of the best practices related to microservices which you may consider following while doing application implementation based on microservices-styled architecture:

SOA & MicroServices

- **Model Services based on Domain-driven Design (DDD)**: Services should be modeled around the business domain. Check out some of the following great reads in relation to **domain-driven design**.
    - Bounded context (Martin Fowler)
    - Domain-driven design for services architecture
    - This video on dDomain-driven design by Eric Evans:

- **Consider separating data storage**: Data should be made private to each of the microservices. Microservice becomes the **owner of its data**. **Any access to data** owned by a specific service should **only happen through APIs**. Failing to do so would allow multiple services to access the database owned by a specific service leading to **coupling between services**. The architecture pattern such as CQRS (Command and Query Responsibility Segregation) comes handy in taking care of data which required to be read by different kinds of users.
- **Build separate teams for different microservices**: Teams should be divided based on microservices with one team working on one microservice. This consists of product manager, and DevOps staff (development, QA, and Ops staff). Recall that microservices shine when they could help organizations in building cloud-native applications which could be released to cloud frequently with very less lead time.
- **Design domain-driven APIs**: APIs should be designed keeping the business domain in mind. Also, implementation details should not be made part of API design.
- **Design cohesive services**: Consider grouping the functions requiring to change together as a single unit rather than separate services. Not doing so would lead to a lot of inter-service communications representing the hard-coupling.
- **Consider separating services for cross-cutting concerns**: One should consider designing separate services for cross-cutting concerns such as authentication and authorization.
- **Automate enough for independent deployment**: Nicely designed micro-services should be able to be deployed independently. And, build and release automation would enhance the deployment process thereby leading to quicker releases and shorter overall lead time. This would help build microservices truly cloud-native in nature with microservices wrapped in containers and deployed to any environment including cloud in an easy manner. Good **DevOps** practice followed organization-wide would help achieve this objective.
- **Failure isolation**: Microservices-based architecture should consider adopting isolation of failure with independent microservices. Architecture principles and design patterns such as some of the following would help achieve the same:
    - **Circuit-breaker** design pattern
    - Asynchronous communication
    - Loose coupling
    - Event-driven architecture
    - Stateless design
    - Self-contained services
    - Timeouts

SOA & MicroServices

- **How does Microservice Architecture work?**

Today, cloud microservices break down the SOA strategy further as a collection of granular functional services. Collections of microservices combine into large macroservices, providing even greater ability to quickly update the code of a single function in an overall service or larger end-user application. A microservice attempts to address a single concern, such as a data search, logging function, or web service function. This approach increases flexibility—for example, updating the code of a single function without having to refactor or even redeploy the rest of the microservices architecture. The failure points are more independent of each other creating creating a more stable overall application architecture.

This approach also creates an opportunity for microservices to become self-healing. For example, suppose that a microservice in one cluster contains three subfunctions. If one of those subfunctions fails, the is being repaired. With orchestration tools such as Kubernetes, self-healing can occur without human intervention; it occurs behind the scenes, is automatic, and is transparent to the end user.

Microservices architectures have come into use along with Docker containers—a packaging and deployment construct. VM images have been used as the deployment mechanism of choice for many years. But containers are even more efficient than VMs, allowing the code (and required code libraries) to be deployed on any Linux system (or any OS that supports Docker containers). Containers are the perfect deployment vector for microservices. They can be launched in seconds, so they can be redeployed rapidly after failure or migration, and they can scale quickly to meet demands. Because containers are native to Linux, commodity hardware can be applied to vast farms of microservices in any data center, private cloud, or hybrid multicloud.

Microservices have been intertwined with cloud-native architectures nearly from the beginning, so they have become indistinguishable from each other in many ways. Because microservices and containers are so abstracted, they can be run on any compatible OS (usually Linux). That OS can exist anywhere: in the public cloud, on premises, in a virtual hypervisor, or even on bare metal. As more development is done in the cloud, cloud-native architectures and practices have migrated back into on-premises data centers. Many organizations are constructing their local environments to share the same basic characteristics as the cloud, enabling a single development practice across any locations, or cloud-native anywhere. This cloud-native approach is made possible and necessary by the adoption of microservices architectures and container technologies.

- **What are the pros and cons of Microservice Architecture?**

SOA & MicroServices

Advantages of microservices

The advantages of microservices seem strong enough to have convinced some big enterprise players such as Amazon, Netflix, and eBay to adopt the methodology. Compared to more monolithic design structures, microservices offer:

- **Improved fault isolation**: Larger applications can remain mostly unaffected by the failure of a single module.
- **Eliminate vendor or technology lock-in**: Microservices provide the flexibility to try out a new technology stack on an individual service as needed. There won't be as many dependency concerns and rolling back changes becomes much easier. With less code in play, there is more flexibility.
- **Ease of understanding:** With added simplicity, developers can better understand the functionality of a service.
- **Smaller and faster deployments**: Smaller codebases and scope = quicker deployments, which also allow you to start to explore the benefits of Continuous Deployment.
- **Scalability**: Since your services are separate, you can more easily scale the most needed ones at the appropriate times, as opposed to the whole application. When done correctly, this can impact cost savings.

Disadvantages of microservices

Microservices may be a hot trend, but the architecture does have drawbacks. In general, the main negative of microservices is the complexity that any distributed system has.

Here's a list of some potential pain areas and other cons associated with microservices designs:

- **Communication between services is complex**: Since everything is now an independent service, you have to carefully handle requests traveling between your modules. In one such scenario, developers may be forced to write extra code to avoid disruption. Over time, complications will arise when remote calls experience latency.
- **More services equals more resources**: Multiple databases and transaction management can be painful.
- **Global testing is difficult**: Testing a microservices-based application can be cumbersome. In a monolithic approach, we would just need to launch our WAR on an application server and ensure its connectivity with the underlying database. With microservices, each dependent service needs to be confirmed before testing can occur.
- **Debugging problems can be harder**: Each service has its own set of logs to go through. Log, logs, and more logs.
- **Deployment challengers**: The product may need coordination among multiple services, which may not be as straightforward as deploying a WAR in a container.
- **Large vs small product companies**: Microservices are great for large companies, but can be slower to implement and too complicated for small companies who need to create and iterate quickly, and don't want to get bogged down in complex orchestration.

- **What is the difference between Monolithic, SOA and Microservices Architecture?**

SOA & MicroServices

	Microservice	SOA	Monolithic
Design	Services are built in small units and expressed formally with business-oriented APIs.	Services can range in size anywhere from small application services to very large enterprise services including much more business functionality.	Monolithic applications evolve into huge size, a situation where understanding the entirety of the application is difficult.
Usability	Services exposed with a standard protocol, such as a RESTful API, and consumed/reused by other services and applications.	Services exposed with a standard protocol, such as SOAP and consumed/reused by other services – leverage messaging middleware.	Limited re-use is realized across monolithic applications.
Scalability	Services exist as independent deployment artifacts and can be scaled independently of other services.	Dependencies between services and reusable sub-components can introduce scaling challenges.	Scaling monolithic applications can often be a challenge.
Agility	Smaller independent deployable units ease build/release management, thereby high operational agility.	Enhances components sharing that increases dependencies and limits management capabilities.	Difficult to achieve operational agility in the repeated deployment of monolithic application artifacts.
Development	Developing services discretely allows developers to use the appropriate development framework for the task at hand.	Reusable components and standard practices helps developers with implementation.	Monolithic applications are implemented using a single development stack (i.e., JEE or .NET), which can limit the availability of “the right tool for the job”.

- **What are the challenges you face while working Microservice Architectures?**

These are major challenges of microservices architecture and proposed solutions:

1. Data Synchronization (Consistency) — Event sourcing architecture can address this issue using the async messaging platform. The SAGA design pattern can address this challenge.

2. Security — An API Gateway can solve these challenges. There are many open source and enterprise APIs are available like Spring Cloud Gateway, Apigee, WSO2, Kong, Okta (2-step authentication) and public cloud offering from AWS, GCP and Azure etc. Custom solutions can also be developed for API security using JWT token, Spring Security, and Netflix OSS Zuul2.

3. Services Communication — There are the different way to communicate microservices –

a. Point to point using API Gateway
b. Messaging event driven platform using Kafka and RabbitMQ
c. Service Mesh

4. **Service Discovery** — This will be addressed by open source Istio Service Mesh, API Gateway, Netflix Eureka APIs. It can also be done using Netflix Eureka at the code level. However, doing it in with the orchestration layer will be better and can be managed by these tools rather doing and maintaining it through code and configuration.

5. Data Staleness — The database should be always updated to give recent data. The API will fetch data from the recent and updated database. A timestamp entry can also be added with each record in the database to check and verify the recent data. Caching can be used and customized with an acceptable eviction policy based on business requirements.

6. Distributed Logging, Cyclic Dependencies of Services and Debugging — There are multiple solutions for this. Externalized logging can be used by pushing log messages to an async messaging platform like Kafka, Google PubSub, ELK etc. Also, a good number of APM tools available like WaveFront, DataDog, App Dynamics, AWS CloudWatch etc.

It's difficult to identify issues between microservices when services are dependent on each other and they have a cyclic dependency. **Correlation ID** can be passed by the client in the header to REST APIs to track all the relevant logs across all the pods/Docker containers on all clusters.

7. Testing — This issue can be addressed with unit and integration testing by mocking microservices individually or integrated/dependent APIs which are not available for testing using WireMock, BDD, Cucumber, integration testing.

8. Monitoring & Performance — Monitoring can be done using open-source tools like Prometheus with Grafana APIs by creating gauges and matrices, GCP StackDriver, Kubernetes, Influx DB, combined with Grafana, Dynatrace, Amazon CloudWatch, VisualVM, jProfiler, YourToolKit, Graphite etc.

Tracing can be done by the latest Open tracing project or Uber's open source Jaeger. It will trace all microservices communication and show request/response, errors on its dashboard. Open tracing , Jaeger are good APIs to trace API logs Many enterprise offerings are also available like Tanzu TSM etc.

9. DevOps Support — Microservices deployment and support-related challenges can be addressed using state-of-the-art CI/CD DevOps tools like Jenkin, Concourse (supports Yaml), Spinnaker is good for multi-cloud deployment. PAAS K8 based solutions TKG, OpenShift.

10. Fault Tolerance — Istio Service Mesh or Spring Hystrix can be used to break the circuit if there is no response from the dependent microservices for the given SLA/ETA and provide a mechanism to re-try and graceful shutdown services without any data loss.

SOA & MicroServices

- **What are the characteristics of Microservices**

- **Service contract**: Similar to SOA, microservices are described through well-defined service contracts. In the microservices world, JSON and REST are universally accepted for service communication. In the case of JSON/REST, there are many techniques used to define service contracts. JSON Schema, WADL, Swagger, and RAML are a few examples.

- **Loose coupling**: Microservices are independent and loosely coupled. In most cases, microservices accept an event as input and respond with another event. Messaging, HTTP, and REST are commonly used for interaction between microservices. Message-based endpoints provide higher levels of decoupling.

- **Service abstraction**: In microservices, service abstraction is not just an abstraction of service realization, but it also provides a complete abstraction of all libraries and environment details, as discussed earlier.

- **Service reuse** : Microservices are course-grained reusable business services. These are accessed by mobile devices and desktop channels, other microservices, or even other systems.

- **Statelessness**: Well-designed microservices are stateless and share nothing with no shared state or conversational state maintained by the services. In case there is a requirement to maintain state, they are maintained in a database, perhaps in memory.

- **Services are discoverable** : Microservices are discoverable. In a typical microservices environment, microservices self-advertise their existence and make themselves available for discovery. When services die, they automatically take themselves out from the microservices ecosystem.

- **Service interoperability**: Services are interoperable as they use standard protocols and message exchange standards. Messaging, HTTP, and so on are used as transport mechanisms. REST/JSON is the most popular method for developing interoperable services in the microservices world. In cases where further optimization is required on communications, other protocols such as Protocol Buffers, Thrift, Avro, or Zero MQ could be used. However, the use of these protocols may limit the overall interoperability of the services.

- **Service composeability**: Microservices are composeable. Service composeability is achieved either through service orchestration or service choreography.