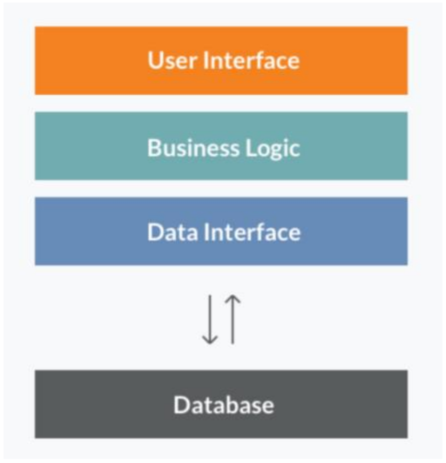
Abstract: Microservices are a recent development in the software industry. Debates about microservices vs. monoliths are inescapable in the new microservices movement. Heavy applications can be made more affordably and with real advantages like scalability and flexibility thanks to the microservice design. Microservices are typically used in one form or another by tech behemoths like Netflix, Amazon, and Oracle. Contrarily, because it poses hazards to conventional software delivery processes, the monolithic approach is losing value. In this paper, each design will be individually examined before moving on to the ultimate comparison of microservices vs monoliths.

1. Monolithic architecture

1.1. Definition & Architecture

Monolithic is the typical structure for software applications, which is an all-in-one architecture in which all components of the software operate as a single unit. Monoliths are best thought of as a non-distributed solution. It has three components:

- Client-side user interface
- Business logic
- Data interface



Monollithic architecture

A single database is used by all three sections. This model's software is based on a single code base. The server-side application usually handles all the HTTP requests and executes the business logic. In monoliths, the server-side logic, the UI logic, the batch jobs, etc., are all bundled in a single EAR (Enterprise Archive), WAR (Web Archive), or a JAR(Java Archive).

To make a change to this sort of application requires updating the entire stack by accessing the code base and building and deploying an updated version of the service-side interface. This makes updates restrictive and time-consuming.

Early in a project's life cycle, monoliths can be useful for code management, cognitive overhead, and deployment. This allows the entire project to be released at the same time.

1.2. Design principles

1.2.1. DRY

DRY is stands for “Don’t Repeat Yourself”
This indicates that there shouldn't be any duplicate code or design elements and that you should aim to retain the functionality of a system's behavior in a single piece of code. We will use the same ideas when developing software because we are seeking for same principles for system design. When a code or system only exists in one location, it is simpler to maintain because any changes to the code or system only need to be made in that one location.

1.2.2. KISS

KISS is stands for “Keep It Simple, Stupid”

Making your code or system simple is what this principle advises. Avoid adding complexity that is not necessary. It's simpler to maintain and comprehend a basic code. This idea can be used in both the implementation and design of the code. Use names for apis and methods that make sense and correspond to their duties. You should also remove redundant code and unneeded features.

The responsibilities of your applications and those from the system layers should be kept apart as well. Simplicity should be a primary design goal and superfluous complexity should be avoided because most systems function best when kept simple rather than having complex designs.

1.2.3. YAGNI

YAGNI is stands for “You Ain’t Gonna Need It”.

Making the code or system simple is what this principle advises. Avoid adding complexity that is not necessary. It's simpler to maintain and comprehend a basic code. This idea can be used in both the implementation and design of the code. Use names for APIs and methods that make sense and correspond to their duties. Redundant code and unneeded features should also be removed.

The responsibilities of the applications and those from the system layers should be kept apart as well. Simplicity should be a primary design goal and superfluous complexity should be avoided because most systems function best when kept simple rather than having complex designs.

1.3. Advantages

Depending on a variety of criteria, organizations can choose monolithic or microservices architecture. The key benefit of employing a monolithic architecture is the speed with which an application can be developed due to the simplicity of having a single code base.

The following are some of the benefits of a monolithic architecture:

- Easy development: Since the application is built with one code base, it is more simple to develop at the beginning of the project. As long as the monolithic approach is a standard way of building applications, any engineering team has the right knowledge and capabilities to develop a monolithic application.

- Easy testing and debugging: End-to-end testing can be completed faster with a monolithic architecture because it is a single, centralized entity. For example, testers can implement end-to-end testing by simply launching the application and testing the UI with Selenium.

- Easy deployment: The monolithic architecture only requires a single file or directory to set it in motion makes deployment easier. When it comes to monolithic applications, develpers just have to deal with one file or directory deployment. It's easier to deploy because only one file is used. In comparison to microservices design, network latency and security issues are comparatively minor. So, when developing a small application, monolithic design is one of the greatest architectures that can be applied.

1.4. Disadvantages

Monolithic architecture has certain potential drawbacks, despite its advantages. The all-in-one structure, which is the monolith's defining feature, causes several problems. They are as follows:

- Hard to modify: Because the monolith is built on a single code base, all modifications must be implemented across the entire architecture. This covers the application's scaling. Scaling merely one segment is impossible. The application as a whole must scale at the same time. Moreover, these changes brings costly side effects, and also this makes the overall development process much longer. New developing features become time-consuming and expensive to implement.

- Barriers to new technology: Applying a new technology to a monolithic program is highly difficult because the entire application must be redeveloped. If a team has been developing the same monolithic program for a long time, updating the entire code base with the latest technologies can be unpleasant. The majority of their time is spent managing legacy apps rather than developing new or unique solutions. As a result, introducing new technologies and frameworks is not an option.

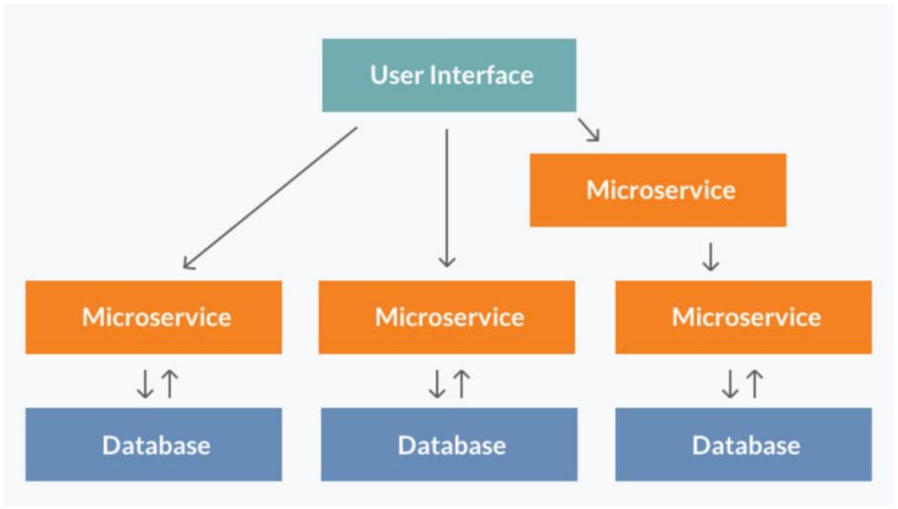
- Complexity: Because there is only one code base, the program becomes increasingly complex as it grows and evolves. Users cannot limit those changes to a specific segment or portion of the application; they must be coordinated across the entire application.

2. Microservices architecture

The drawbacks of monolithic mentioned above were the main reasons that led to the evolution of microservices architecture.

2.1. Definition

Microservices architecture, or simply microservices, are an architectural and organizational approach to software development in which software is made up of small independent services that communicate using APIs. Microservices are distributed.



Microservices architecture

Microservices is an architectural method based on a sequence of independently deployable services. These services each have their own business logic and database that serves a distinct purpose. Microservices architectures make applications easier to scale and faster to develop, enabling innovation and accelerating delivery for new features. Microservices split up major business and domain-specific issues into discrete, self-contained code bases. Microservices don't diminish complexity; instead, they make it visible and manageable by breaking down activities into tiny processes that work independently but contribute to the larger whole.

2.2. Design principles

Microservices have successfully replaced the problems with conventional business application development as an evolutionary type of software engineering. Its design pattern allows businesses to use an agile approach, integrate CI/CD using DevOps techniques, and quicken growth in a cutthroat market.
But the road to such an ideal transition is filled with challenges. There are innumerable examples of failed microservices implementations, often due to poor implementation rather than a lack of understanding on the part of the engineers. Use of a technological stack that developers require but do not comprehend or are unable to integrate and work together, a lack of autonomy, improperly designed software architecture, or adhering to the same monolithic application architecture principles are a few examples.

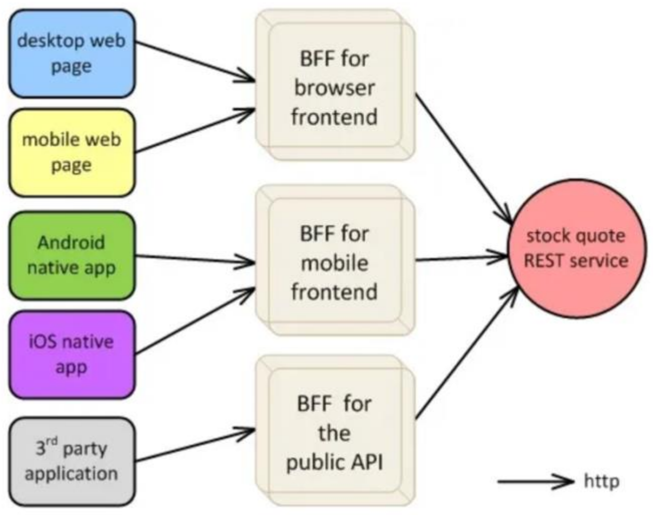
The use of microservices design principles enables programmers to make important choices regarding their architecture, frameworks, and tools for navigating the complex technology landscape. Here are the following set of core principles for microservices architecture design – IDEALS:

- **I**nterface segregation
- **D**eployability
- **E**vent-driven
- **A**vailability over consistency
- **L**oose coupling
- **S**ingle responsibility

2.2.1. Interface segregation

There should be distinct interfaces catering to the individual demands of each sort of client rather than a single interface with all potential methods clients might need. The goal of interface segregation for microservices is that each type of frontend sees the service contract that best suits its needs. For example, a mobile native app wants to call endpoints that respond with a short JSON representation of the data; the same system has a web application that uses the full JSON representation; there's also an old desktop application that calls the same service and requires a full representation but in XML. Different clients may also use different protocols. For example, external clients want to use HTTP to call a gRPC service.

Segregating the interface so that each type of client sees the service interface that it requires rather than attempting to impose the same service contract on all types of service clients. It can be done by using an API gateway is a well-known substitute. It is capable of transforming message formats, message structures, protocol bridging, message routing, and many other things. The Backend for Frontends (BFF) pattern is a well-liked substitute. In this situation, we have an API gateway for each type of client, in other words, a separate BFF for each customer.



BFF pattern

2.2.2. Deployability

Almost throughout the whole history of software, the design effort has been concentrated on design choices relating to the organization of implementation units (modules) and the interaction of runtime components. In the software space, architecture tactics, design patterns, and other design strategies have offered guidelines for organizing software elements in layers, avoiding excessive dependencies, designating particular roles or concerns to particular types of components, and making other design decisions. There are crucial design choices for microservice developers that go beyond the software components.

Deployability, a field of technology and development decisions, is now crucial to the success of microservices. The primary cause is the much higher number of deployment units brought on by microservices. Altogether, deployability involves:

- Deployability entails setting up the runtime infrastructure, which includes persistence, security, networking, pods, clusters, and containers.
- Shifting microservices across different runtime environments or scaling them
- Speeding up the commit, build, test, and deploy procedure.
- Minimizing downtime for replacing the current version.
- Synchronizing software program version changes.
- Monitoring the health of the microservices to quickly identify and remedy faults.

Automation is the key to effective deployability. Automation requires using tools and technology effectively, and this area has undergone the most change overall since the introduction of microservices. Microservice developers should therefore be on the lookout for new tool and platform directions while constantly evaluating the advantages and disadvantages of each new option. Here is a list of strategies and technologies that should be considered in order to improve deployability:

- Containerization and container orchestration
- Service mesh
- API gateway
- Serverless architecture
- Monitoring tools
- Log consolidation tools
- Tracing tools
- DevOps
- Blue-green deployment and canary releasing
- Infrastructure as Code (IaC)
- Continuous delivery
- Externalized configuration

2.2.3. Event-driven

Since they are more likely to satisfy the scalability and performance needs of current software systems, event-driven microservices entail aiming for modeling event-driven microservices. Since message senders and receivers are separate entities with no knowledge of one another, this type of design also encourages loose coupling. Because of the design's capacity to handle brief microservice outages, which allow for later processing of the queued messages, reliability is also increased.

However, event-driven microservices, sometimes referred to as reactive microservices, can be difficult to implement. Asynchronous and parallel processing is engaged, and it may be necessary to use synchronization points and correlation IDs. The design must take errors and lost communications into consideration. Correction events and techniques for rolling back data changes are frequently required. Additionally, the user experience for user-facing transactions supported by an event-driven architecture should be carefully designed to keep the end-user updated on developments and problems.

2.2.4. Availability over Consistency

The industry is making a significant effort to offer tools that allow high availability and, thus, accept eventual consistency. The explanation is straightforward: modern end users won't tolerate a lack of availability. Data replication is the main approach for microservices that allows the availability choice. Different design patterns may be used, such as:

- Service Data Replication pattern: When a microservice has to access data that belongs to other apps, this fundamental pattern is utilized (and API calls are not suitable to get the data). Data is duplicated and an easy access to the microservice will be provided. In order to make the replica and master data consistent, the solution also needs a data synchronization mechanism (such as an ETL tool or program, publish-subscribe messaging, or materialized views).

- Command Query Responsibility Segregation (CQRS) pattern: Seperate between the planning and execution of orders that alter data and those that only read data (queries). For enhanced performance and autonomy, CQRS often draws on Service Data Replication for the queries.

- Event Sourcing pattern: Instead of storing the current state of an object in the database, the sequence of append-only will be stored, immutable events that affected that object. Current state is obtained by replaying events, and we do so to provide a "query view" of the data. Thus, Event Sourcing typically builds upon a CQRS design.

2.2.5. Loose-Coupling

The degree of connection between two software pieces is referred to as coupling in software engineering. Afferent coupling in service-based systems refers to the relationship between service users and the service. We are aware that the service contract should be used for this engagement. Furthermore, the contract shouldn't be inextricably linked to implementation specifics or a particular technology. A distributed component known as a service is one that other programs can use. The service custodian may not always be aware of where every service user is (often the case for public API services). Contract amendments should therefore generally be avoided. The service contract is more likely to alter when the service logic or technology has to advance if it is firmly tied to those elements. Several strategies can be used and combined to promote (afferent and efferent) loose coupling. Examples of such strategies include:
- Point-to-point and publish-subscribe
- API gateway and BFF
- Contract-first design
- Hypermedia
- Facade and Adapter/Wrapper patterns
- Database per Microservice pattern

2.2.6. Single Responsibility

As a result of the tight coupling that comes from having several responsibilities in one class, designs become fragile, difficult to change, and prone to unexpected failure. The cohesion of services within a microservice might be seen to fall under the concept of single responsibility. The deployment unit must only include one service or a small number of coherent services, according to the microservice architecture design. A microservice may experience monolithic problems if it is overloaded with duties, or if there are too many services that aren't fully coherent. It becomes more difficult to evolve the technology stack and functionality of a bloated microservice. With numerous developers working on various moving elements that belong in the same deployment unit, continuous delivery also becomes difficult.

An approach that has become popular in the industry to drive the scope of microservices is to follow Domain-Driven Design precepts. In brief:

- A service (e.g., a REST service) can have the scope of a DDD aggregate.
- A microservice can have the scope of a DDD bounded context. Services within that microservice will correspond to the aggregates within that bounded context.
- For inter-microservice communication, domain events can be used when asynchronous messaging fits the requirements; API calls using some form of an anti-corruption layer when a request-response connector is more appropriate; or data replication with eventual consistency when a microservice needs a substantial amount of data from the other BC readily available.

2.3. Advantages

The software industry is a fast-paced, highly competitive sector, with new technology, methods, and tools being released on a daily basis. This makes it difficult for firms to figure out which solutions to implement, let alone when and how. Microservices have existed for more than a decade. Despite this, many businesses are unsure whether the framework will soon disappear or if they must embrace it to avoid becoming obsolete. The architectural framework was created to solve the scale, adaptability, and productivity limits of monolithic structures. Here are some key advantages of a microservices architecture:

- Agility: Microservices encourage the formation of small, self-contained teams that are responsible for their own services. Teams are empowered to work more independently and quickly within a small and well-understood environment. This shortens the length of the development cycle. Everything from data storage to communication is handled by each service. As a result, a single service can be developed, tested, and deployed by small teams. Because of the lower scope, onboarding new developers is a quicker and more straightforward process. New developers can get started right away because they don't need to know the function of each service or how the entire system is linked. This means that microservices can help businesses cut down onboarding costs and get new staff up and running faster.

- Flexible scaling: Each microservice can be scaled independently to meet demand for the application feature it supports. This allows teams to properly size infrastructure, appropriately estimate the cost of a feature, and ensure service availability in the event of a spike in demand.

- Easy deployment: Microservices allow for continuous integration and continuous delivery (CI/CD), making it simple to experiment with new ideas and roll back if something doesn't work. The low cost of failure allows for more experimentation, easier code updates, and faster time-to-market for new features.

- Technological freedom: Teams are allowed to use whatever technology they want to tackle their challenges. As a result, teams developing microservices can select the most appropriate tool for each task.

- Reusability: Teams can use functions for many purposes since software is divided into discrete modules. A service created for one function can be used as a foundation for another feature. This allows an application to self-bootstrap since developers may add new features without having to write code from scratch.

- Resilience: The resistance to failure of an application is increased when it is service independent. If a single component fails in a monolithic design, the entire application can fail. Microservices enable apps to handle total service failure by reducing functionality rather than crashing the entire app.

2.4. Disadvantages

Given the current pace of business and the advent of new technologies that make microservices administration even easier, the list of benefits is growing faster than the list of negatives, but there are still some drawbacks. While microservices simplify most of the development process, there are a few situations where they might actually add to the complexity.

- Complexity: In order to get additional flexibility, a microservice architecture adds complexity. With a microservice architecture, the components are distributed throughout the network. As a result, connecting the microservices together to get the whole application up and operating takes more time. Connecting entails not just locating the microservice on the network but also granting permitted access to all of the microservices that make up the whole system. In the end, microservices do provide more flexibility in terms of maintenance and upgrade, but that flexibility comes at the expense of increasing complexity. To illustrate the point, here are some issues should be considered before applying microservices:
    - Maintaining several programming languages and frameworks is difficult.
    - Existing tools will almost certainly be incompatible with the new service requirements.
    - Each service necessitates unique testing and monitoring, which necessitates the use of automation technologies.
    - Because each service has its own database and transaction management system, maintaining data consistency is difficult.
    - There are various microservices patterns to select from, making it difficult to determine which one is best for your company.

- Costliness: The cost of a microservices architecture is another downside. The necessity for services to communicate with one another will result in a large number of remote calls. This can result in higher network latency and processing expenses than traditional designs. To avoid disruption, developers will need to account for this issue and create solutions that try to reduce the number of calls. Because each service is isolated and has its own CPU and runtime environment, microservices require extra resources. Because of the absence of consistency, more tools, servers, and APIs will be used. The demand for resources grows when each service uses its own programming language and technology stack.

- Security: Due to the massive increase in the volume of data shared between modules, microservices pose some significant security problems when compared to monolithic programs. More of the system are being exposed to the network by interacting with several small containers, which means more possible attackers. It's also worth mentioning that, because containers are highly reproducible, a flaw in one module can quickly escalate into a larger issue. Because source code is frequently reused across multiple applications, hackers have simple access to it. As a result, microservices can soon become a security problem without the proper tools and training.

3. When to use?

3.1. Monolithic

- Small team: Developers might not need to cope with the intricacy of the microservices architecture if they are a startup with a small workforce. There is no pressing need to start with microservices because a monolith can handle all of the company's requirements.

- Simple application: Monolithic architectures are better suited for small applications that do not need a lot of business logic, high scalability and flexibility.

- No microservices expertise: To function properly and provide business value, microservices need a high level of expertise. Without any prior technological experience, it is quite unlikely that starting a microservices program from scratch can be profitable.

- Rapid launch: A monolithic model is the greatest option if the application needs to be constructed and launched as quickly as feasible. It works well when the company’s initial spending goal is to test the business idea with minimal money.

3.2. Microservices

- Independent deployment: When a company has to alter functionality and deploy that feature without requiring the rest of the system to change, microservices are quite helpful. As a result, new functionality can be added to a microservices architecture without any downtime. In a SaaS-based company, where software goods must always be available for use, the requirement for zero downtime deployment would unquestionably be applicable.

- Data isolation: For example, PHI and PII information must be protected by many companies, including those in the healthcare and financial sectors. They must also follow particular compliance and data security standards. Data partitioning is made possible by isolating the data from the processing that affects it in a microservice design. This makes it simple to identify which services interact with PII and need further supervision, control, and auditing.

- Autonomy: Organizations may need to divide up responsibility among teams, allowing each team to make decisions and create software on its own. A microservices-based architecture provides a structure that encourages this kind of autonomy within the business when teams must work independently without close collaboration. A microservices architecture erects rigid barriers that call for the creation of this kind of team autonomy.

- Reduction of resources: One of the most important points that the company should carefully consider is the need to analyze the practicality of implementation before going completely develop a full-fledged microservice-based system. To put it another way, designing and integrating microservices is an expensive and time-consuming task, so it's crucial that the advantages of microservices outweigh the drawbacks. Many small to medium sized businesses have been unable to take use of the multiple advantages of a microservices-based app architecture because of this crucial "cost-benefit" factor. The cost of developing a microservices app often prevented smaller businesses from taking use of its many advantages.