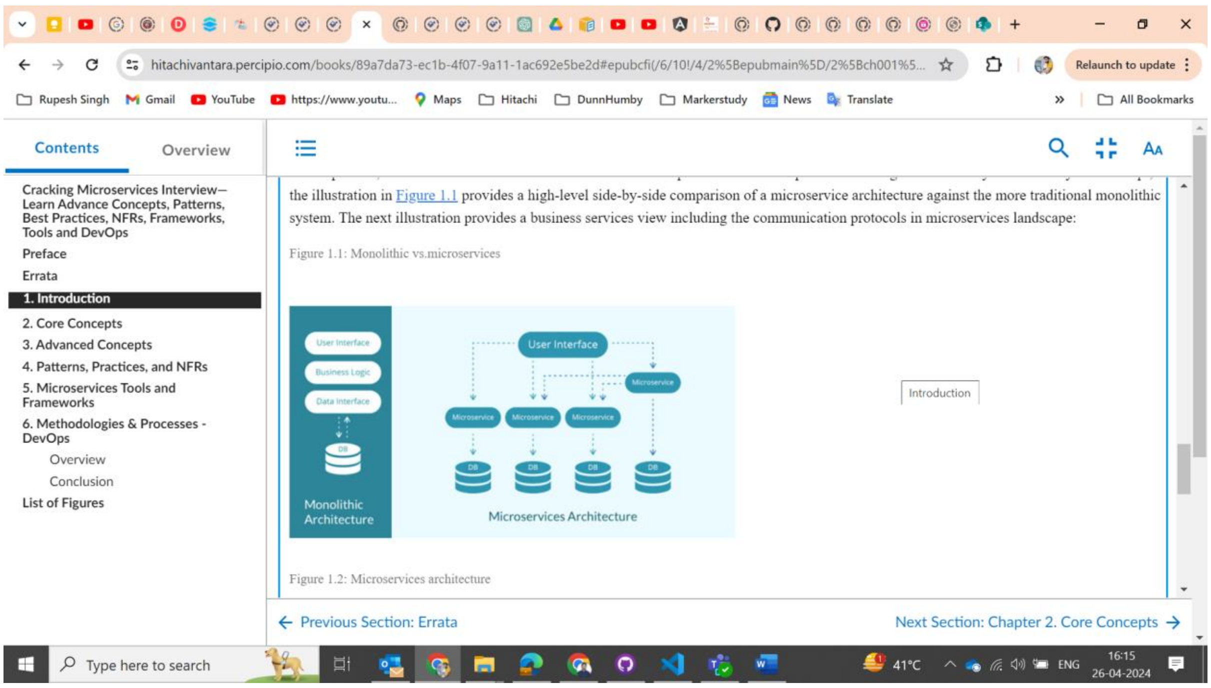
he following are the benefits of microservices:

- **Separate components:** The primary benefit of the microservice architecture is its loosely coupled components. These components can easily be developed, replaced and scaled individually.
- **Increased availability and resilience:** Microservices improve fault isolation. As complex applications are broken into separate service components and deployed on multiple servers, failing of one of the services or modules will not impact the entire application. A single service fault can easily be replaced with another service (simple to build resilience around the small set of services) increasing the application's availability.
- **Easy to change technology stack:** With microservices, software development teams can try a new stack on specific service to avail larger benefits at the application level. There is no long-term commitment to one particular stack as there are no dependency concerns. For example, recommendation micro-services can use python due to its machine-learning libraries against which event-processing micro-services may use Java due to the multithreading properties of JVM.
- **Easy to understand even in distributed environment:** Understanding how an application is developed is important when there is a change of hand in development teams. In a distributed development project when some of the team members are geographically dispersed, microservice architecture make it easier for dev teams to understand the entire functionality of a service as it is not built into one single package.
- **Organized around business capabilities:** Microservices are not organized around technical capabilities of a particular product, but rather business capabilities. As the end goal is user experience and customer satisfaction, the teams leveraging microservices are not divided into UI teams, database teams and so on. In fact, there are cross-functional teams that work towards fulfillment of one single functionality. Here's a diagrammatic representation for quick understanding.
- **Re-usability of services:** As microservices are organized around business capabilities and not a specific project or product requirement, they are project agnostic. This enables technology teams to reuse services and reduce costs.
- **Decentralized data management:** Large scale and complex enterprise applications are normally three-tier. Martin Fowler, in his microservices article, describes that microservices let each service manage its own database, either different instances of the same database technology or entirely different database systems. As he mentioned, this approach is called Polyglot Persistence.
- **Easy to deploy:** While technology teams have to deploy an entire application again because of small change in the code, with microservices this deployment

becomes easy. The scope of deployment is smaller and only the service that has a problem needs to be deployed again.

The following important aspects enable success with a microservices-based system:

- Monitoring and healthchecks of the services and infrastructure.
- Scalable infrastructure for the entire landscape via cloud layer, containers, and orchestrators.
- **Security at multiple levels:** Authentication, authorization, key management, secure communication, cloud layer, application, networks, and more.
- Rapid application delivery of microservices with different teams focusing on different microservices.
- DevOps and CI/CD practices and infrastructure.



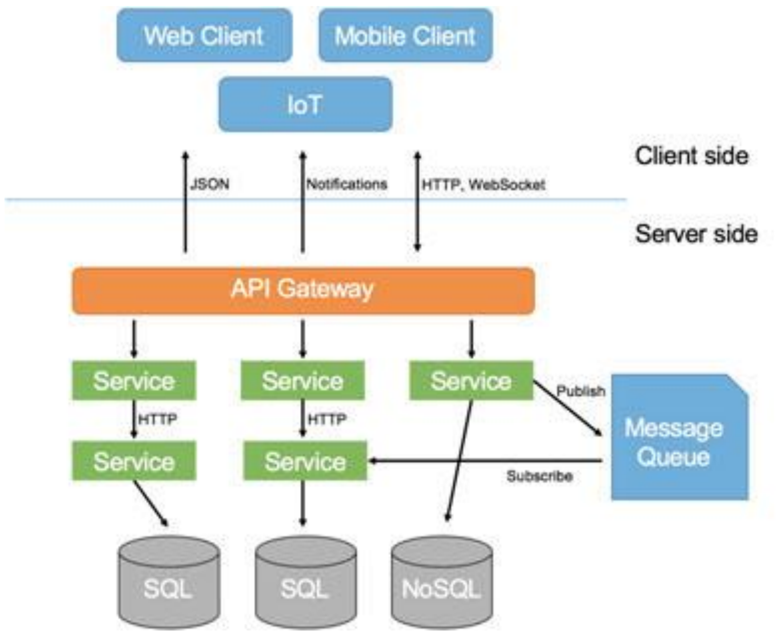
the illustration in Figure 1.1 provides a high-level side-by-side comparison of a microservice architecture against the more traditional monolithic system. The next illustration provides a business services view including the communication protocols in microservices landscape:

Figure 1.1: Monolithic vs.microservices

Figure 1.2: Microservices architecture

← Previous Section: Errata

Next Section: Chapter 2. Core Concepts →



What are microservices?

Microservices is a variation of **service-oriented architecture (SOA)** style that organizes a business application as a collection of loosely coupled services. In a microservices architecture paradigm, services are fine-grained and leverage light-weight protocols such as REST over SOAP.

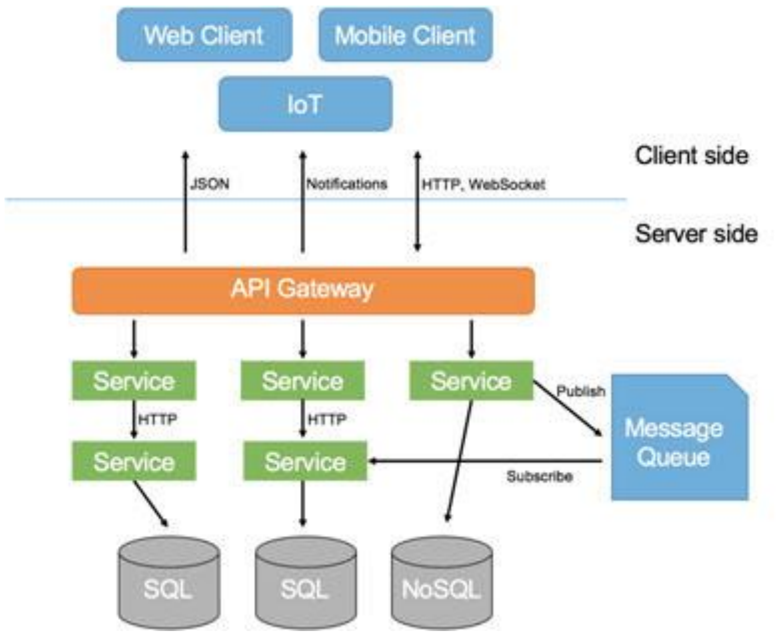
1. The following are the similarities of microservices and SOA services:

    - Microservices are reusable and can be invoked from desktop applications, web applications, and mobile applications, or by other microservices.
    - Microservices are loosely coupled and interact with each other through lightweight mechanism; for example, REST, JSON, XML, and more
    - WADL, JSON Schema, or Swagger is leveraged for describing the microservices.

    The following are the difference between microservices and SOA services:

    - Microservices typically don't rely on enterprise products like **Enterprise Service Bus (ESB)**.
    - There are no physical infrastructure dependencies with microservices. They are usually deployed in Docker containers, which encapsulate both the code and required libraries.

2. **Describe the microservices architecture**.



The microservices reference architecture consists of the following:

- **Client:** The client can be a web application, desktop application, mobile application, or even another remote microservice.

- **API Gateway:** The API gateway handles client requests and forwards them to the appropriate service.

- **Microservices:** These are small, independent, and loosely coupled microservices that are created based on the requirements of specific business domain. These services will typically have their own database tier.

- **Service discovery:** This enables service lookup to find the service endpoints in the microservices landscape, which typically consists of service registration and discovery.

- **Management:** This aspect enables application and infrastructure monitoring for the microservices applications and helps identify failures and issues.

- **Identity provider:** This enables the authentication and authorization of end users. This authenticates user or client identities and issues security tokens. This may leverage capabilities like OAuth and AD

- **Content delivery networks:** This is a distributed network of proxy servers and their data centers.

- **Static content:** This houses all the content of the system.

- **Remote service:** This enables the remote access information that resides on a network of IT devices.

This architecture facilitates the avoidance of huge application implementation for a large complex system. The microservices architecture enables loose coupling between various collaborating procedures and it also has the ability to run in an autonomous manner under various types of situations.

How will you monitor multiple microservices for different health indicators?

The Spring Boot actuator is a good tool to monitor microservices metrics and counters for individual microservices. But if you have multiple microservices, it's difficult to monitor each individually. For this, we can use open source tools like Prometheus, Nagios, and ELK Search.

Monitoring is an approach of gathering, storing, analyzing, and reporting data. Monitoring systems produce valuable data that can be leveraged to efficiently monitor and manage microservices application and also enhance service performance. Failure and performance data can be analyzed for patterns in failures, which can be correlated with events to gain critical insights.

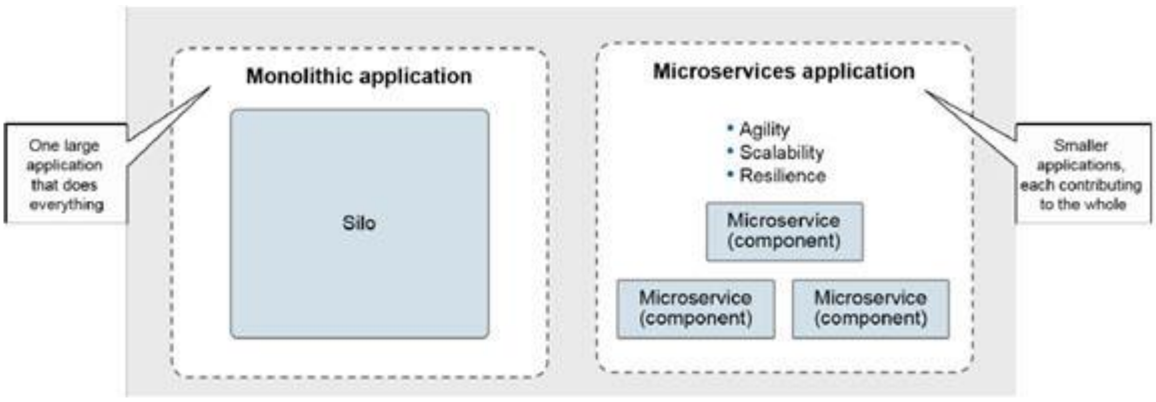
Monitoring technology and tools are divided in two broad categories: libraries and platforms. Some tools include both, providing a library for instrumentation and platform for collection and visualization of monitoring data.

Monitoring libraries are embedded or integrated into microservices application during the development phase. Most popular frameworks such as Java, .NET, Go, and others include resources for writing to data streams. These resources are leveraged for logging and monitoring. Paid and open source third-party libraries are available to enable metrics reporting dashboards. Examples include open-source libraries like **AppMetrics** for .NET and **SPF4J** for Java.

- **Prometheus:** Prometheus is an open source monitoring tool originally created by **SoundCloud** for application-level monitoring. This is widely used to store and query *time-series data*, which is data that describes actions over time. Prometheus is often combined with other tools, especially **Grafana**, to visualize the time series data and to provide dashboards. Prometheous is a pull-based monitoring tool. It contains metrics at given intervals, displays them and can also trigger alerts.
- **ELK Stack:** ELK Stack is a collection of three open-source products such as Elasticsearch, Logstash, and Kibana and can be leveraged for business logging. They are developed, managed, and maintained by Elastic. E stands for

ElasticSearch, leveraged for storing logs, L stands for LogStash, leveraged for shipping as well as processing and storing logs, and K stands for Kibana, which is a visualization tool. ELK is created to enable users to take data from any source, in any format, and to search, analyze, and visualize that data in real time. ELK provides centralized logging which is useful when attempting to identify problems with applications and servers. This allows you to search the application logs in a single place. This also helps to find issues that occur in multiple servers by connecting their logs during a specific time frame.

- **Nagios:** The Nagios monitoring tool operates within IT infrastructures to monitor applications, services, servers, network devices, and other critical components. It offers report, dashboards and alerting so that administrators or the ops team can determine the problems occurrence and determine the root cause to resolve the issue.
-
-
- **What is a monolithic architecture?**
- The monolithic architecture style is like a large container in which all the software components of an application are integrated into one large package. In contrast, the objective of the microservice architecture style is to completely decouple application components from one another such that they can be created, deployed, scaled, and maintained independently. The following diagram demonstrates the monolithic and microservices architecture:



1. It's an evolution of application architecture, SOA and publishing APIs.
    - **SOA:** Focuses on reuse, technical integration challenges, and technical APIs.

- **Microservices architecture:** Focuses on functional decomposition, business capabilities, and business APIs.

2. **What are the main differences between microservices and monolithic architecture?**

Microservices architecture	Monolithic architecture
Build as a suite of small services.	Build as a single logical executable.
Requirement changes can be applied to each service independently.	Requirement changes involve building and deploying a new version of the entire application.
Each service can be scaled independently.	Entire application has to be scaled in case a bottleneck is identified in one part.
Each service can be developed in different programming languages.	Typically, the entire application is developed in one programming language or framework.
Smaller code base is easier to maintain and manage.	Large code base is intimidating to a new development team.
Simple deployment as each service can be deployed individually with minimum downtime.	Complex deployment with maintenance windows and scheduled downtime.
Microservices architecture is loosely coupled.	Monolithic architecture is primarily tightly coupled.
Changes done in a single data model does not affect other microservices.	Any changes in the data model affect the entire database

1. **What are the benefits of microservices over traditional monolithic architecture?**

   The microservices architecture style facilitates continuous delivery and continuous deployment, that is, DevOps pipeline. Microservices are small in size and can be quickly created, deployed, and scaled. This is an excellent fit for the agile development paradigm as the business users don't have to wait to see the full product. In the Microservice architecture, individual services can be built in different languages like Java and Scala. Also, different microservices may be written leveraging different versions of the same language such as Java 8 and 9.

   Microservices are autonomous components; hence, microservice can be independently scaled in the microservices application landscape. For example, in an airline ticket company, the ratio between flight ticket searches and bookings is 50:1; hence, the search microservice can be scaled without impacting the ticketing microservice. This reduces the cost as the organization doesn't have to scale the entire application to meet the performance requirements. By leveraging microservices, we can easily change the technology or version of a specific microservice rather than having to impact the entire application. Microservices are DevOps friendly and the changes can be continuously integrated with a DevOps pipeline to increase speed of delivery. Microservices architecture can be leveraged for developing large complex distributed application that can be scaled efficiently.

A monolith application is created as one unit; it is usually composed of three tiers: a database (an RDBMS), a server-side business tier (war deployed on Tomcat/Websphere), and a UI interface tier (JSP). Whenever there is a need to add/update functionality in a Monolith application, the development team needs to change at least one of these three components and deploy the new version to production. The entire system is tightly coupled, has limitations in choosing technology stack, and has low cohesion. When there is a need to scale a monolith, one has to deploy the same version of the monolith on multiple servers by deploying the big war/ear file multiple times. Everything is contained in a single executable file.

On the other hand, the microservices architecture style is made up of smaller autonomous services, which are divided based on business capabilities that communicate with each other in leveraging asynchronous protocols using a lightweight framework. Most of the above issues are resolved in the microservices paradigm. Please refer to the above question for additional benefits between monolith vs. microservice.

1. **How do microservices communicate with each other?**

   Microservices are integrated leveraging simple protocols like REST over HTTP. Other communication protocols are also leveraged for integration like AMQP, JMS, Kafka, and more. The communication mechanism is broadly divided into two categories: synchronous and asynchronous communication.

   - **Synchronous communication:** HTTP is a synchronous protocol. The client sends a request and waits for a response from the microservice. The important point here is that the protocol (HTTP/HTTPS) is synchronous and the client can only continue its task when it receives the HTTP response from the service. Ideally, one should minimize the number of synchronous calls between microservices because networks are brittle and they will introduce latency. Ribbon is a client-side load balancer that can be leveraged for efficient utilization of resources. A Hystrix circuit breaker can be leveraged to handle partial failures gracefully without a cascading effect on the entire microservices ecosystem. Distributed commits should be avoided at any cost; instead, one should opt for eventual consistency using asynchronous communication.

- **Asynchronous communication:** In this type of communication, the client does not wait for a response; instead, it just sends the message to the message broker. AMQP (like RabbitMQ) or Kafka can be used for asynchronous communication across microservices to achieve eventual consistency.

What are the cloud-native applications?

Cloud-Native Applications (CNA) is a paradigm of development that encourages the adoption of best practices in the area of continuous delivery and distributed software development. These applications are designed specifically for cloud architecture like Azure, AWS, or CloudFoundary. The following are the key aspects to create cloud-native applications:

- DevOps, continuous integration, continuous delivery, microservices, and containers are the key concepts in developing cloud-native applications.
- Spring Boot, Spring Cloud, Kubernetes, Docker, Maven, Jenkins, and Git are a few tools that help create cloud-native applications effortlessly.
- Microservices is an architectural style for building large complex distributed applications as a collection of small services. Each service is responsible for a specific business capability, runs in its own process, and communicates via HTTP REST API or messaging AMQP.
- **DevOps:** This is an extensive process framework that facilitates among other things collaboration between the development and IT operation teams with the goal of continuously delivering high-quality software as per customer requirements.
- **Continuous Integration and Continuous Delivery (CICD):** This ensures automated delivery of small low-risk updates constantly to production. This makes it possible to collect feedback faster.
- **Containers:** Containers like Dockers offer logical isolation to each microservice, thereby eliminating the issue of run on my machine forever. It's much efficient and faster compared to virtual machines.

1. **What are the benefits of the microservice architecture?**

   The following are the benefits of the microservice architecture:

- Microservice is developed independently by a smaller team of developers (normally two to five developers).
- Microservice is loosely coupled and meaning services are independent of each other, in terms of development and deployment.
- Microservice can be created leveraging different programming languages; for example, Java, C#, Python, and more.
- Microservice enables flexible and an easy way to automate the deployment with CI tools (for example, Jenkins, Hudson, Bamboo, and more.).
- Each microservice is focused around a specific domain or sub-domain which addresses a specific business requirement.
- Microservice is easy to understand, modify, and maintain for a development team as separation of code, small team, and focused work. This also ensures that the productivity of a new team member will be far better.
- Microservice enables taking advantage of emerging the latest methodologies and technologies (DevOps framework, programming practice, and more).
- Microservice has code for only business logic, No CSS, HTML, or other UI components.
- Microservice is easy to scale based on business demand. Individual services can scale as per business requirements; there is no need to scale all components together.
- Microservice can deploy on commodity hardware or low/medium configuration servers. The microservices architecture facilitates independent deployment and reduction in deployment time frame.
- Every microservice may have its own storage, but this is driven primarily by the business requirements; one can have a common database like MySQL or Oracle for all services.
- Fault isolation; for example, a process failure will not bring the whole system down. Even if one microservice in the application fails, the system still continues to function.

1. **What are the drawbacks of the microservice architecture?**

   The following are the drawbacks of the microservice architecture:

   - The microservice architecture brings a lot of operational overheads.

- DevOps skills are required for building microservices (`http://en.wikipedia.org/wiki/DevOps`).
- Duplication of effort.
- Distributed system is complicated to manage.
- Difficult to trace problems because of distributed deployment. In a distributed system, it's hard to debug and trace the issues.
- Complex to manage the whole application when the number of microservices increases.
- Difficult to achieve a strong consistency model across microservices
- ACID transactions will not span multiple processes.
- Greater requirement for an end-to-end testing.
- Required cultural changes across teams like Dev and Ops working together in the same organization.