NOKIA

Anatomy of a microservice

White paper

About the NFV Insight Series

The application of virtualization and cloud principles, such as network functions virtualization (NFV) and software defined networks (SDN), represent a major shift for the communications and networking industry. Until recently, this approach appeared to be unworkable because of stringent performance, availability, reliability, and security requirements of communication networks. Leading communications service providers (CSPs) now implement Network optimized Cloud architectures to gain competitive advantage through increased automation and responsiveness, as well as by delivering an enhanced customer experience, while reducing operational costs. This series of briefs and white papers addresses some of the key technical and business challenges faced by service providers as they move to the cloud.

NOKIA

Contents

NOKIA

Introduction

Today's modern software architectures and design principles are built on lessons learned and experiences in the software industry at large. Some of these are expressed and adopted as popular approaches to microservices, DevOps, the Twelve-Factor App, Agile development, and others.

As Martin Fowler and others have pointed out,[1] the software industry has been defining and working with the microservices concept for some time now. While there is no formal or agreed upon definition of what constitutes a microservice, a common set of characteristics exhibited by microservices-based architectures can nonetheless be identified. Some of the most interesting characteristics include:

- Componentization that enables the replacement of pieces independent of one another
- Organization based on business capabilities (functions) instead of technology
- Smart endpoints and dumb pipes
- Decentralized data management with one database for each service rather than one database for a whole product
- Infrastructure automation with mandatory continuous delivery.

Based on this list, one can deduce that other domains, such as DevOps and Agile development,[2] play a pivotal role in realizing these capabilities. Also, these can be seen as intertwined or, in some cases, as being prerequisites of other capabilities. For example, does an agile organization structure enable the continuous delivery of software changes?

The principles and concepts of a modern software application are also rooted in popular manifestos, such as the Twelve-Factor App.[3] These provide a framework for organizing the components and requirements of an application as service.

These concepts can be broad and applicable to many software architectures that might not claim to be microservices and/or DevOps based. In some cases, they could even be misconstrued or misinterpreted by consumers of these ideas.

From the outset, it is important to note that this paper is neither exhaustive nor complete. Rather it has been written to generate interest in microservices and in seeing how microservices concepts improve application development, simplify the functional composition of a product, add agility to software deployment and management, as well as further refine the principles of a microservices architecture.

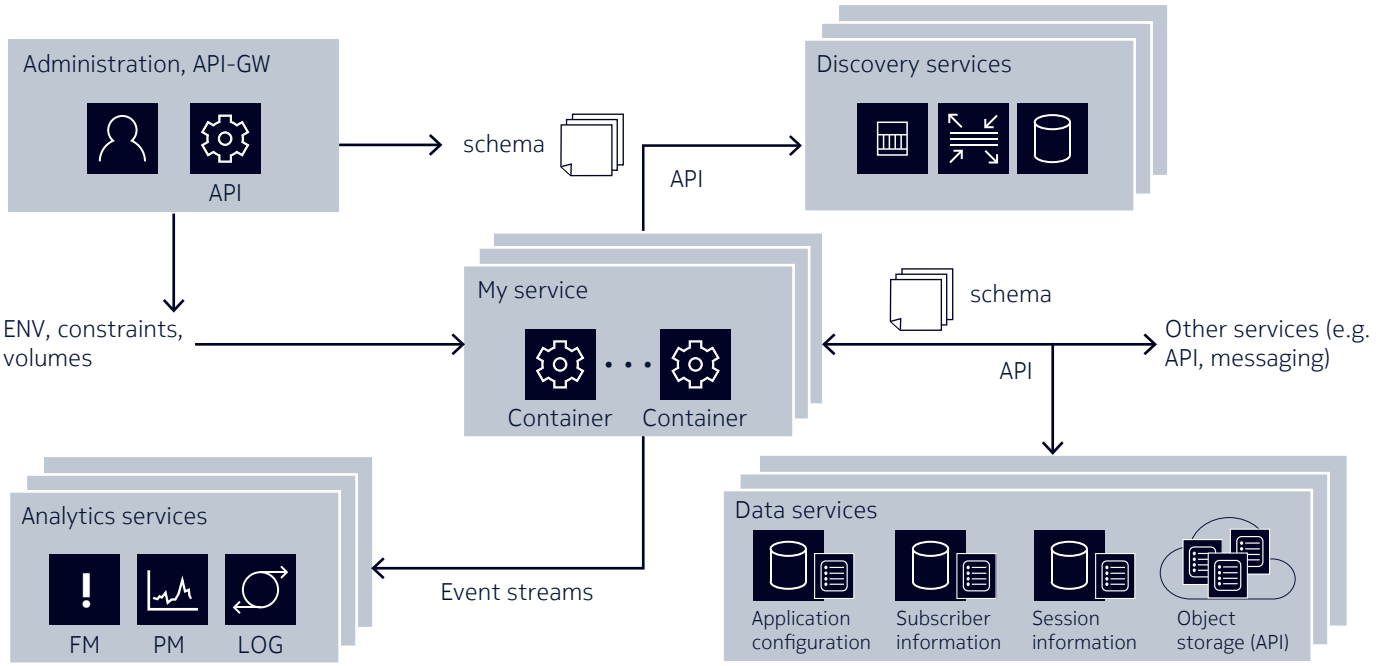
Many of the concepts and principles across these domains can be applied to modern software that both meets the needs of customers and exceeds their expectations. To explore these ideas, this paper presents the anatomy of a microservice.

1 Microservices by James Lewis, Martin Fowler. https://martinfowler.com/articles/microservices.html
2 Manifesto for Agile Software Development. http://agilemanifesto.org/
3 The Twelve-Factor App. https://12factor.net/

Microservice anatomy

To begin with, what does it mean for a software component to be part of a microservices-based architecture? Figure 1 illustrates the software components essential to the function or capability that the microservice represents. The “My Service” component shows the minimal and modular software components required to “build one thing and do it well.”[4]

Figure 1. Microservice Anatomy



In addition to the software component’s core function is the responsibility to provide a contract for the interfaces — for example, the discovery service API and the data schema — along with each of the services provided by the infrastructure. These services include the:

- Administration and API gateway: An API-first approach, as well as a large collection of services requires a unified and controlled access point application service exposed for consumption.
- Discovery services: These services provide the publish/subscribe mechanisms for registration and discovery of distributed services within the cluster.
- Analytics services: These services pertain to all telemetry data offloaded from the microservice for processing based on a consistent set of policies.
- Data services: These services consist of the data layer for the persistence and availability of state data required by the microservice.

These infrastructure services can themselves be implemented as microservices. Each of these auxiliary services, domains, and interfaces is briefly described in the following sections.

4 Unix Philosophy - https://en.wikipedia.org/wiki/Unix_philosophy

NOKIA

A microservice instance

A microservice or service instance is built to do one thing, and only one thing, well. This is the basis of disassembling an application into its functional pieces.

To offer the dynamic scalability required in a microservices architecture, the service must be stateless and dateless.. To meet the offered workload, it is expected that a microservice must be able to easily expand and contract in order to meet the number of running instances directed by an orchestrator. For its part, the scaling of the microservice is assisted by a traffic distributor or load balancer.

The service must also offer a clear definition of its interfaces in addition to providing a lasting contract for the other services and management functions with which the interfaces interact. The notion of "APIs first" is one important aspect of these interfaces. Here the premise is that, for a greenfield service, the use cases should be built and the interfaces described for the application function before the backend services are composed and implemented. In the case of a brownfield service, for example, a number of patterns or best practices are available in order to evolve to a microservice. This can be done by first building the APIs to expose the targeted service. The microservice services also help to insulate and decouple the service's consumer or user from the internal design and implementation changes. The aim is that the service, as defined by the contract APIs, can be refactored and evolve over time.

In most cases, a single type of microservice does not offer a complete service function solution. The service function needs to be built from a collection of services. Composing and managing the service function can be quite complex; it could involve 100s or even 1000s of clustered microservice instances. In the case of Kubernetes, for example, it is tasked with providing the orchestration and management function for microservices clusters. As such, it enables cloud native application builders to construct highly available Kubernetes clusters. These provide the computing layer abstraction, as well as the resources for managing the services functions' lifecycle. It is built around a set of constructs and simple principles that describe the needs of an application using a manifest file. Additional projects, such as Helm, are making it easier to describe, package, and configure cloud native applications to consume configured Kubernetes resources in a hardware-agnostic manner.[5]

Analytics services

As is the case with any well-designed software, a microservice generates a variety of operations-related telemetry data. This information includes logs, events, performance data, usage data, and health checks. To monitor and control the microservice, this data is used by analytics tools. In some cases, the data can result in an anti-pattern for a microservice that stores this data in a running container. This is due to the transient state of containers, which come and go (keeping them stateless is one of the primary goals) and because the hosting environment may not be under the application's complete control. Providing a microservice with the capability to output event streams, makes possible the application of a rich set of data analytics. In many cases, the event streams, such as stdout and stderr can be transformed, interpreted, and correlated with other tiers, such as the infrastructure and the container service, as well as related services and/or applications.

5 Many other competing technologies, such as Swarm, DCOS, and Rancher are available for composing and managing microservices at scale, but Kubernetes is favored predominately by the cloud native ecosystem.

In a native Linux hosting environment, standard tools, including journald, syslog, and rsyslog provide seamless integration with popular open source logging and analytics tools. For the most part, a micro service need only adapt to feeding these tools. In the case of Docker, a logging driver is used to provide support. Independent of the hosting infrastructure, it is also possible to run a dedicated logging container that serves as a logging funnel for multiple application containers. In this use case, the application is completely independent of the hosting infrastructure and controls how and where the data is forwarded. This allows for flexibility in how event streams can be de-coupled from analytics services, using a messaging bus (e.g., Kafka) to achieve higher performance.

Data services

In some cases, dataless and stateless operation are required for the application of other microservices design principles. In these instances, the data service is what provides a persistent or recoverable state. The microservice remains obligated to describe its database schema and its required service level agreement (SLA). For example: How often does the database need to be accessed? Or, what is the maximum delay for a database transaction?

One key principle of microservices is that data tied to a particular service should be kept private and not shared among other services. However, the adoption of a dedicated database server for each service is neither realistic nor practical — the licensing alone would make it cost prohibitive.

To effectively meet particular use cases, as well as assure a sufficient degree of optimization in order to be stateful, data services tend to demand a different composition of resources. In many large-scale cloud infrastructures, such as AWS, these capabilities are provided *as-a-service. Regardless of how data services are made available, user applications need only establish the contract and consume the services, using the API(s) offered.

Discovery services

Unlike traditional applications and services deployed on fixed locations (i.e., physical hardware), dynamic microservices running across a cluster of machines, such as virtual machines and containers, make it difficult to track the network location of a service instance.

By serving as the registry of network information for service instances, the discovery service plays a key role. The registry is usually deployed as a highly available service with up-to-date service information. It can exist as-a-service or, in some deployments, such as Kubernetes, can be baked-in as part of the infrastructure. In all cases, the dynamically assigned network location of a service instance is maintained.

Several patterns allow a microservice to take advantage of this service based on its needs:[6]

- Client-side service discovery – With this pattern, the client is responsible for querying the discovery service directly to get the network information of the available service instances.
- Server-side service discovery – In this case, client requests are directed towards a router, also known as a load balancer, running at a known network location. The server(s) and router interact directly with the registry to maintain an up-to-date view of the service instance(s).
- Self and third-party registration patterns are also options for service instances to register with the discovery service.

6 Microservices.io. http://microservices.io/patterns/

The publication of the location and availability information for a service makes it possible for other services or other instances of the same microservice to be autonomously discovered. This mechanism can also be used to share configuration data needed to make use of the service.

Administration and API gateway

As noted earlier, a microservice offers its functions using APIs. When a solution is composed from a collection of microservices – each having its own APIs – it can be advantageous to offer a higher-level solution API, which hides or abstracts the details of each microservice. As the number of microservices grows, it become a practical necessity to take a consistent approach to the processing of incoming API requests. Requests may need to be authenticated for user and role, logged for billing purposes, rate-limited to enforce external SLAs, cached to improve performance, transformed into other formats, and routed to the correct back-end micro-service. An API gateway can implement the higher-level abstraction, while providing common consistent API handling functions, thus relieving each microservice from having to do it.

Microservices should be self-contained and able to operate autonomously. However, an application may need to apply domain-specific logic on top of the relatively generic microservices that it uses. For example, this might need to be done when user input is being collected or data is being populated from an external source into the data services, which support the microservices (i.e., provisioning). In cases such as these, the operator may need to install rules, search patterns and thresholds into the analytics services in order to monitor system behavior. Indeed, such administrative functions may themselves be created as microservices, which are oriented towards operations functions. Some may even be transient — running periodically to perform a function, such as issuing a report — and then terminating once the function is complete. The APIs provided by these management functions allow the applications to provide the user or operator with a web-based user interface, or the ability to adapt to legacy interfaces where needed.

To distribute images and instantiate them, microservices are deployed using containers. This assures the consistency of development, testing, and deployment environments. The immutability of the containers enables the microservice components to be updated independently of one another and rapidly replaced in the event of a failure.

Life cycle management operations for microservices, including instantiation, upgrades, scaling, and healing, are delegated to an external orchestrator, such as Kubernetes. This enables a consistent approach toward the administration of microservices — even when they come from different sources. Tools, such as Helm, simplify the configuration and packaging of complex applications, which consist of many microservices, while capturing their deployment parameters at the same time.

Resource constraints

A microservice must have a holistic understanding of its resource needs with regard to CPU, memory, network bandwidth, as well as disk throughput. With this information, a microservice can make use of orchestration and container runtime environments to prescribe its needs for an optimal use of resources while maintaining the promised SLA.

The resource constraints implementation for microservices runtimes are mostly built on Linux control groups' (cgroups) capabilities. Cgroups allow a microservice to set resource limits (CPU time, memory, disk I/O, and network bandwidth) at a granular level that can be administered and enforced by the hosting system kernel.

By default, a container runtime system, such as Docker, does not enforce any resource constraints, allowing a container to consume as much of a given resource as the hosting system kernel allows.[7] By prescribing the explicit needs of a microservice, the builder of the cloud native application can be confident that the service will perform as expected and that there are no unintended consequences as a result of getting insufficient or too many resources.

Orchestrators, such as Kubernetes, allow a cloud native application builder to be prescriptive. The builder can request resource constraints, using a simple API, or the manifest file for the targeted container runtime. The resources requested with Kubernetes are used to schedule the Pod(s) in a cluster. As a result, the containers get the resources they need. Also, it is important to note that the capabilities of resource constraints are being continuously improved with Kubernetes, along with its support across cloud providers and container runtimes.

Deployment parameters

With regard to environment-specific parameters that a microservice needs to be aware of at runtime, it is preferable to keep these parameters limited to make things as simple as possible when starting a service. Safe defaults are always expected. Put somewhat differently, what's needed is a minimal set of override parameters to enable the microservice instance to begin within the deployed environment, find the additional services needed for configuration, as well as make itself discoverable. These parameters usually include security hardening, such as username and password, as well as TLS certificates.

Consider the case of deploying a database service instance of MariaDB, using Docker's Compose tool:

```
version: '2'
services:
mydbcontainer:
image: mariadb
command: mysqld --innodb-buffer-pool-size=20M
volumes:
- "./mariadb:/var/lib/mysql:rw"
environment:
- "MYSQL_DATABASE=my_db_name"
- "MYSQL_USER=my_db_user"
- "MYSQL_PASSWORD=my_db_user_passwd"
- "MYSQL_ROOT_PASSWORD=my_db_root_passwd"
ports:
- "3306:3306"
```

7 Docker Documentation https://docs.docker.com

In this example, the deployment parameters are passed as environment variables or, in some cases, as optional arguments to the executable command. A similar scheme applies for other deployment tools, such as Kubernetes and Docker Swarm.

With Kubernetes, application builders can manage configuration parameters separate from the application code, using the ConfigMap API. The data blobs (key-value pairs) can be managed independently and made available for consumption during microservice deployment. Furthermore, secrets, such as passwords, SSH keys, and certificates are controlled separately from the configuration parameters to allow for proper safekeeping and best-practices security.

Service scalability

Scaling a microservice should be as easy as starting additional instances. The additional load comes from the load balancer, which discovers member availability within the replicated service instances pool. The fundamental premise is that the service is built to run with more than a single instance. For a new service, this may not be that difficult. However, for an existing one, the challenge is to adopt the previously mentioned principles in order to make the service nimble so that it can easily scale while being orchestrated by an external actor.

An orchestrator, such as Kubernetes, provides the tools to prescribe a service's scalability needs when deployed in a cluster. Within a Kubernetes deployment or Replica set resource, the number of pods or service instances can be manually or automatically scaled based on measured demand. For auto scaling capabilities, Kubernetes has implemented basic features and continues to improve these capabilities in order to meet more use cases. Noteworthy here is that the service scalability is no longer a self-implemented function; it's a capability of the underlying resource cluster.

Value to the market

With each of the previously described services, the concerns have been compartmentalized to allow for the refactoring of today's complex telecommunications software. Seen as a whole, each microservice is not a complete solution but rather a building block of a flexible, comprehensive, and dynamic solution. Understanding how each of these blocks fit into the larger ecosystem, while at the same time producing software components that enable application differentiation, is key to improving R&D's focus and efficiency, as well as reducing time-to-market for new features.

In the emergent operator's cloud ecosystem, these infrastructure services are becoming more prevalent and available for consumption by cloud native applications. Furthermore, making use of a common services infrastructure opens the possibility of reducing operational costs and simplifying deployments. On top of that, analytics and data services offer monetization opportunities and the exploration of patterns and trends in managed data services promises an additional means for network-wide optimization. Moreover, in contrast to each application replicating its own instance of these auxiliary services, each service would be capable of supporting an application at a cluster level.

Despite all this promise, it is important to remember that a comprehensive architecture and robust design must still be part of any transformation to a microservices-based architecture. There is no magic bullet when it comes to applying fundamental software development principles. Even so, it remains possible to learn from past challenges and the experiences of others in order to establish a firm foothold for capitalizing on the modern software as a service approach.

NOKIA

References

1. A Cloud-Native Vision for SBC
2. Network-optimized cloud architectures for session border controllers
2. Nokia AirGile cloud-native core eBook
3. Building a cloud-native core for a 5G world
4. FD.io: The Universal Data Plane

Acronyms

AAA	Authentication, Authorization, and Accounting
API	Application Programming Interface
CPU	Central Processing Unit
DRA	Diameter Routing Agent
ENV	Environment Variables
SBC	Session Border Controller
SLA	Service Level Agreement
TLS	Transport Layer Security
VM	Virtual Machine

Nokia Oyj
Karaportti 3
FI-02610 Espoo
Finland
Tel. +358 (0) 10 44 88 000

Product code: SR1802022125EN (February)