**What is a Microservice?**

There are numerous definitions of the microservice architecture. Some take the name too literally and claim that a service should be tiny—for example, 100 LOC. Others claim that a service should only take two weeks to develop. Adrian Cockcroft, formerly of Netflix, defines a microservice architecture as a service-oriented architecture composed of loosely coupled elements that have bounded contexts.

**The scale cube defines three separate ways to scale an application: X-axis scaling load balances requests across multiple, identical instances; Z-axis scaling routes requests based on an attribute of the request; Y-axis functionally decomposes an application into services.**

Shape, rectangle

Description automatically generated

**X-axis scaling load balances requests across multiple instances:** You run multiple instances of the application behind a load balancer. The load balancer distributes requests among the *N* identical instances of the application. This is a great way of improving the capacity and availability of an application.

**Diagram

Description automatically generated**

**Z-axis scaling routes requests based on an attribute of the request:** The router in front of the instances uses a request attribute to route it to the appropriate instance. An application might, for example, route requests using userId.

**Diagram

Description automatically generated**

In this example, each application instance is responsible for a subset of users. The router uses the userId specified by the request Authorization header to select one of the *N* identical instances of the application. Z-axis scaling is a great way to scale an application to handle increasing transaction and data volumes.

**Y-axis scaling functionally decomposes an application into services:** X- and Z-axis scaling improve the application’s capacity and availability. But neither approach solves the problem of increasing development and application complexity.To solve those, you need to apply *Y-axis* scaling, or *functional decomposition* by splitting a monolithic application into a set of services.

Diagram

Description automatically generated

A *service* is a mini application that implements narrowly focused functionality, such as order management, customer management, and so on. A service is scaled using X-axis scaling, though some services may also use Z-axis scaling. For example, the Order service consists of a set of load-balanced service instances.

The high-level definition of microservice architecture (microservices) is an architectural style that functionally decomposes an application into a set of services. Note that this definition doesn’t say anything about size. Instead, what matters is that each service has a focused, cohesive set of responsibilities.

#### **Microservices as a form of modularity**

Applications must be decomposed into modules that are developed and understood by different people. In a monolithic application, modules are defined using a combination of programming language constructs (such as Java packages) and build artifacts (such as Java JAR files). Long-lived, monolithic applications usually degenerate into big balls of mud.

The microservice architecture uses services as the unit of modularity. A service has an API, which is an impermeable boundary that is difficult to violate. You can’t bypass the API and access an internal class as you can with a Java package. As a result, it’s much easier to preserve the modularity of the application over time. There are other benefits of using services as building blocks, including the ability to deploy and scale them independently.

#### **Each service has its own database**

A key characteristic of the microservice architecture is that the services are loosely coupled and communicate only via APIs. One way to achieve loose coupling is by each service having its own datastore. In the online store, for example, Order Service has a database that includes the ORDERS table, and Customer Service has its database, which includes the CUSTOMERS table.

At development time, developers can change a service’s schema without having to coordinate with developers working on other services. At runtime, the services are isolated from each other—for example, one service will never be blocked because another service holds a database lock.

**Don’t worry:** The requirement for each service to have its own database doesn’t mean it has its own database server. You don’t, for example, have to spend 10 times more on Oracle RDBMS licenses.

#### **The FTGO(Food to Go) microservice architecture:**

Diagram, polygon

Description automatically generated

The FTGO application’s business logic consists of numerous backend services. Each backend service has a REST API and its own private datastore. The backend services include the following:

Graphical user interface, text, application

Description automatically generated

#### **Comparing the microservice architecture and SOA**

#### Some critics of the microservice architecture claim it’s nothing new—it’s service-oriented architecture (SOA). At a very high level, there are some similarities. SOA and the microservice architecture are architectural styles that structure a system as a set of services.

Table

Description automatically generated

#### **Benefits of the microservice architecture**

Text

Description automatically generated

**The microservices-based FTGO application consists of a set of loosely coupled services. Each team develops, tests, and deploys their services independently.**

**Diagram

Description automatically generated**

The ability to do continuous delivery and deployment has several business benefits:

Graphical user interface, text, application, email

Description automatically generated

#### **Drawbacks of the microservice architecture**

A screenshot of a computer

Description automatically generated with medium confidence

**Finding the right services is challenging:** One challenge with using the microservice architecture is that there isn’t a concrete, well-defined algorithm for decomposing a system into services. As with much of software development, it’s something of an art. To make matters worse, if you decompose a system incorrectly, you’ll build a *distributed monolith*, a system consisting of coupled services that must be deployed together. A distributed monolith has the drawbacks of both the monolithic architecture and the microservice architecture.

##### Distributed systems are complex: Another issue with using the microservice architecture is that developers must deal with the additional complexity of creating a distributed system. Services must use an interprocess communication mechanism. This is more complex than a simple method call. Moreover, a service must be designed to handle partial failure and deal with the remote service either being unavailable or exhibiting high latency.

Implementing use cases that span multiple services requires the use of unfamiliar techniques. Each service has its own database, which makes it a challenge to implement transactions and queries that span services. A microservices-based application must use what are known as *sagas* to maintain data consistency across services. Microservices-based application can’t retrieve data from multiple services using simple queries. Instead, it must implement queries using either API composition or CQRS views.

IDEs and other development tools are focused on building monolithic applications and don’t provide explicit support for developing distributed applications. Writing automated tests that involve multiple services is challenging. These are all issues that are specific to the microservice architecture. Consequently, your organization’s developers must have sophisticated software development and delivery skills in order to successfully use microservices.

**Deploying features spanning multiple services needs careful coordination:** Another challenge with using the microservice architecture is that deploying features that span multiple services requires careful coordination between the various development teams. You have to create a rollout plan that orders service deployments based on the dependencies between services. That’s quite different than a monolithic architecture, where you can easily deploy updates to multiple components atomically.

**Deciding when to adopt is difficult:** Another issue with using the microservice architecture is deciding at what point during the lifecycle of the application you should use this architecture. When developing the first version of an application, you often don’t have the problems that this architecture solves. Moreover, using an elaborate, distributed architecture will slow down development. That can be a major dilemma for startups, where the biggest problem is usually how to rapidly evolve the business model and accompanying application. Using the microservice architecture makes it much more difficult to iterate rapidly. A startup should almost certainly begin with a monolithic application.

Later on, though, when the problem is how to handle complexity, that’s when it makes sense to functionally decompose the application into a set of microservices. You may find refactoring difficult because of tangled dependencies.