BROUGHT TO YOU IN PARTNERSHIP WITH

Getting Started With Microservices

CONTENTS

- Introduction
- What Are Microservices?
- Why a Microservices Architecture?
- Benefits
- Design Patterns for Microservices
- Operational Requirements for Microservices
- The Future, Serverless Computing, and FaaS
- Conclusion

WRITTEN BY ANDY HAMPSHIRE
GLOBAL ARCHITECT AT TIBCO

Introduction

The term "microservices" describes a software architectural style that gives modern developers a way to design highly scalable, flexible applications by decomposing the application into discrete services that implement specific business functions. These services, often referred to as "loosely coupled," can then be built, deployed, and scaled independently.

The "microservices" style is linked to other trends that make this a practical approach. Things like containerization, Agile methods, DevOps culture, cloud services, and the widespread adoption — both culturally and technically — of continuous integration and continuous delivery/deployment (CI/CD) methods across the industry are making it possible to build truly modular, large-scale, service-optimized systems for both internal and commercial use.

This Refcard aims to introduce the reader to microservices and to define their key characteristics and benefits.

What Are Microservices?

Microservices are business functions that are "loosely coupled," which can then be built, deployed, and scaled independently.

Each service communicates with other services through standardized application programming interfaces (APIs), enabling the services to be written in different languages, to use different technologies and even different infrastructure. The concept differs completely from systems built as monolithic structures, where services were inextricably interlinked and could only be deployed and scaled together. However, microservices do share common goals with EAI and SOA architectures.

As each individual service has limited functionality, it is much smaller in size and complexity. The term "microservice" comes from this discrete functionality design, not from its physical size.

Microservices Made Easy

TRY PROJECT FLOGO

An ultra-light, Go-based, open source ecosystem

flogo.io

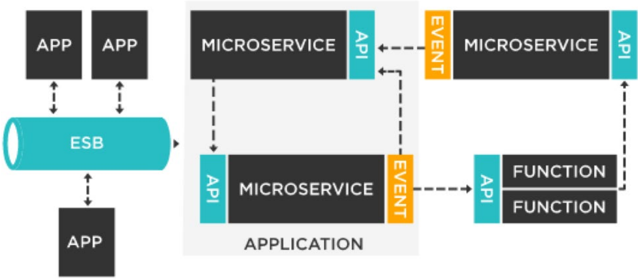

Figure 1: Evolution of Software Application Architectures

Why a Microservices Architecture?

Microservices architectures have risen in popularity because their modular characteristics lead to flexibility, scalability, and reduced development effort. Deployment flexibility, in addition to the rise of cloud-native serverless and function-as-a-service deployment options (such as AWS Lambda and Microsoft Azure Cloud Functions), have created the perfect environment for microservices to flourish in today's IT landscape. These deployment platforms enable microservices and functions to be scaled from inactivity to high volume and back again. Cloud-based services also allow businesses to pay only for the compute capacity they use.

As businesses are continuously looking to be more agile, reduce bottlenecks, and improve application delivery times, microservices architecture continues to rise in popularity. Some of the advantages of using microservices include:

- Application components can be built in different programming languages.
- Each service can be independently deployed, updated, replaced, and scaled.
- Each service is responsible for a single part of the overall functionality and executes it well.
- Use of cloud-native function-as-a-service deployment options is possible.
- "Smart endpoints and dumb pipes" — each microservice owns its domain logic and communicates with others through simple protocols.

Benefits of Microservices

INDEPENDENT SCALING

Each microservice can scale independently. As each instance of a microservice is fully independent, you have the option to deploy multiple instances of a service. This could be on the same hardware, on different machines, on cloud-based infrastructure, or any combination of these. With cloud (both private and public), the ability to scale based on demand means that the service is always able to provide the desired SLAs.

INDEPENDENT UPGRADES

Each service is deployed independently of any other services. Changes local to a service can be easily made by a developer without requiring coordination with other teams. For example, new business requirements and bug fixes can be implemented through updates to the underlying implementation. Where new versions introduce changed APIs, new versions of the service can be built and deployed alongside and service users can migrate to the new version as desired.

EASY(ER) MAINTENANCE

As code in a microservice is limited in functional scope, it should be easier to maintain — its impact on the rest of the code base is far more limited, and its codebase ultimately smaller and easier to understand. Integration testing only needs to be performed at the API level to ensure backward compatibility to other code, so testing can be focused on the business functionality. Anecdotally ,this results in systems that are far better tested, as tests are better and more comprehensively targeted.

TECHNOLOGY INDEPENDENCE

Developers are now free to pick the language and tools that are best suited for their service. As interoperability is at the API level, developers are free to innovate within the confines of their service. It also means that any future rewrite of the service can utilize newer technologies, as opposed to being tied to past decisions — thus taking advantage of technological advances in the future.

FAULT AND RESOURCE ISOLATION

With any large application, finding the cause of a code issue like a memory leak or an unclosed database connection can be hard. But with microservices, this can be easier to manage, as periodic restarts of a misbehaving service only affect the users of that service. A smaller, simpler code base often makes finding errors and issues quicker and easier. This improved fault isolation also limits how much of an application a failure can affect, but with critical components, it is still important to understand the cascading impact of failures. However, as faults are isolated to a single service, they can be ultimately resolved independently and quickly.

Design Patterns for Microservices

DEFINING/DECOMPOSING SERVICES

The first big challenge in getting started with microservices is how to identify the services. Developers often think about services as technical services, much like they would when building a functional library. But microservices are much more about solving a business problem than just a technical one.

So, as we start to think about things in terms of business services, it is often tempting to think along object-oriented lines, and start

to create services like *Customer* or *Order*. But this doesn't go far enough. These OO-like services are too much like "God" services, and they need to be broken down into domains more closely aligned to business-led requirements. Getting the balance right is key, but difficult. Techniques like domain-driven design can help you get the balance right for your environment.

EVENT-DRIVEN ARCHITECTURE

Microservices architectures are renowned for being eventually consistent, given that there are multiple datastores that store state within the architecture. Individual microservices can themselves be strongly consistent, but the system as a whole may exhibit eventual consistency in parts. To account for the eventual consistency property of a microservices architecture, you should consider the use of an event-driven architecture where data changes in one microservice are propagated to interested microservices via events.

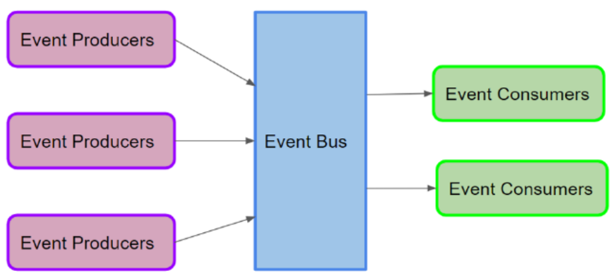

Figure 2: Typical Event Architecture

A pub-sub messaging architecture may be employed to realize the event-driven architecture. One microservice may publish events as they occur in its context, and the events would be communicated to all interested microservices, which can proceed to update their own state. Events are a means to transfer state from one service to another, so that all parts of the application reach an eventually consistent state.

DATABASE DESIGN

Unlike a monolithic application, which can be designed to use a single database, microservices should be designed to use a separate logical database for each microservice. Sharing databases is discouraged and is an anti-pattern, as it makes the database an integration point, stops services being truly independent and introduces a scaling bottleneck. The problems start because the structure of data, the granularity and scope of transactions, and the business requirements will almost certainly overlap.

To solve this conundrum, look at the options to decide what works best in your use case. A database per service scales well, but doesn't work well when multiple services need to share the same data. Sharing databases for related services is a compromise that is often considered, but has issues with scaling, autonomy, and independence of services. Other options include Command Query Responsibility Segregation (CQRS) and the Saga data pattern. Each has pros and cons that should be considered but are still better than the monolithic database.

API GATEWAYS TO CENTRALIZE ACCESS TO MICROSERVICES

All your microservices will be potentially used by disparate clients, ranging from mobile apps to other microservices. As these clients could be external, or in the case of a mobile client, from third-party applications or part of the same infrastructure, we have the issue of how to manage our interfaces. All services will have an API, but will all the APIs be implemented in the same way — for example, REST and JSON? For external clients, how are you going to manage access to the services? What about security?

This is where an API gateway comes in. It acts as a facade that centralizes the aforementioned concerns at the network perimeter, where the API gateway would respond to client requests over a protocol native to the calling client. The gateway can manage security demands outside the architecture, where clients can identify themselves to the API gateway through a token-authentication scheme like OAuth. It can also manage things like data format changes (XML<->JSON translation) for clients and services that are unable to handle multiple formats.

The gateway also needs to be aware of operational considerations for the services it is proxying, especially where the services are running in a service mesh. Working well with service discovery, service replication, and service migration becomes essential when services can dynamically scale.

Operational Requirements for Microservices

Microservices are not the silver bullet that will solve all architectural or infrastructure problems in your existing applications. Moving to microservices may help, but that could also be just a byproduct of refactoring your application and rewriting code to a new platform. True success requires significant investment in understanding the new technologies and taking advantage of them.

SERVICE REPLICATION

Each service needs to be able to replicate, to manage loads, and to be resilient. There should be a standard mechanism by which services can easily scale based on metadata. A container-optimized application platform such as Kubernetes, CloudFoundry, Amazon EKS, or Red Hat's OpenShift, can simplify this process by defining scaling and recovery rules.

SERVICE DISCOVERY

In a microservice world, multiple services are typically distributed in a container application platform. Service infrastructure is provided

by containers and virtual images, both for on-premise and hybrid-/cloud-orientated deployments. The services may scale up and down based on certain predefined rules, so the exact location of a service may not be known until the service is deployed and ready to be used.

The dynamic nature of a service's endpoint address is handled by service registration and discovery. Each service registers with a broker and provides more details about itself (including the endpoint address). Other consumer services then query the broker to find out the location of a service and invoke it.

There are several ways to register and query services, such as ZooKeeper, etcd, Consul, Kubernetes, Netflix Eureka, and others. Kubernetes, in particular, makes service discoverability very easy, as it assigns a virtual IP address to groups of like resources and manages the mapping of DNS entries to those grouped resources.

SERVICE OBSERVABILITY

Some of the most important aspects of managing a microservices-based architecture are service monitoring, metrics, and logging. Through metrics, you can understand how an application runs in its normal state. You can then understand what's not normal, enabling you to take proactive action if, for example, a service is consuming unexpected resources. Elasticsearch, Fluentd, and Kibana can aggregate logs from different microservices, provide a consistent visualization, and make that data available to business users. When things go wrong, distributed tracing tools like Zipkin can give you a holistic view of the end-to-end process alongside performance metrics. This can help find and solve most runtime problems.

RESILIENCE

Software failures will occur, no matter how much or how well you test. This becomes all the more important when multiple microservices are deployed to different platforms. The key concern is not "how to avoid failure" but "how to deal with failure." It's important for services to automatically take corrective action to ensure user experience is not impacted. The Circuit Breaker pattern allows you to build in resiliency — Netflix's Hystrix is a good library that implements this pattern.

DEVOPS

Continuous integration and continuous delivery/deployment (CI/CD) are very important in order for microservices-based applications to succeed. These practices ensure that bugs are identified via automated testing and human factors are removed by automated deployment pipelines.

On the CI side, the pipeline will take code from a developer's checked-in source, build it, and deploy it to a testing environment where it is tested both in isolation and in relation to service users.

On the CD side, there are two options. First is "Delivery," where the tested code is sent to the production repository ready for final deployment. Second is "Deployment," where the code is deployed automatically to the production environments.

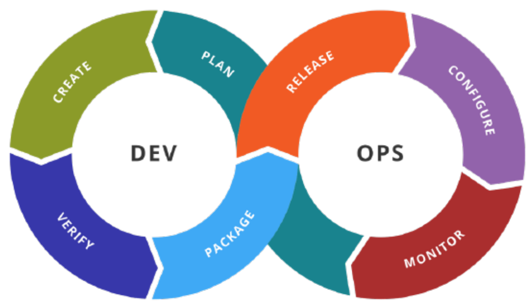

Figure 3 - The DevOps Cycle [1]

DEPLOYMENT

Being able to automate the Delivery/Deployment process is essential to a successful microservices program. But automated deployment through to production needs to be carefully managed. Regardless of whether deployment is fully automated or not, the Blue/Green deployment pattern is a good way of managing risk. In this scenario, existing services are running on the "green" system, and updated services are deployed to the "blue" system. The request routing (API gateway or service mesh) manages the switchover to the new version of the service, often in a phased manner to ensure that the service introduces no problems.

The Future, Serverless Computing, and FaaS

When people first start experimenting with microservices, they often default to using familiar techniques — for example, simple request/reply services based on RESTful APIs. The problem with this synchronous approach is that, as you have to wait for a response, the services become dependent on each other. If one service is running slower or doesn't respond, it means the service that called it will run slower or fail with a timeout. This coupling can mean losing some of the benefits of a microservices architecture, creating a more interdependent structure akin to a Request/Reply Service-Oriented Architecture (SOA) style.

If you design your services using an event-driven or Pub/Sub model, you can ensure that parts of your application continue to work, as opposed to the entire application becoming unresponsive. Take Netflix as an example. On occasion, you might notice that the "continue watching" button doesn't appear. This is because a specific service isn't available. However, that doesn't mean all of Netflix stops. Users can still browse for shows and watch previews, so other aspects of the Netflix service are still available, even though one service may not be.

Developers that fully embrace a microservices approach realize that true scalability is enabled with loose coupling and event-driven architecture. A service can be asynchronous, performing an action, broadcasting a message, and continuing on with its primary function without having to wait for a response from another service. This lends well to the adoption of Serverless and Function-as-a-Service (FaaS) platforms going forward, giving businesses easy access to "on-demand" capacity at a competitive price.

Conclusion

The microservices architectural style has well-known advantages. It can certainly help businesses evolve and innovate faster.

Consider the operational requirements of microservices carefully, in addition to the benefits, before moving to a microservices architecture — especially if you are refactoring an existing monolithic application. Better software engineering, organizational culture, and architecture will be enough to make your existing structure more agile, without having to jump to microservices. In this case, a natural migration may be a better way to proceed. But for new applications, microservices and FaaS are probably the best options available to us today.

Written by **Andy Hampshire**

Andy Hampshire has worked in the IT industry for over 30 years (been there, done that, and got quite a few t-shirts). Andy loves most forms of motorsport, so when Andy is not playing with (or working on) his cars and bikes he can often be found watching racing, at the moment mostly at Kart tracks supporting his son’s racing “career”. Away from work and cars, his idea of heaven is walking with his dog in the Surrey Hills where he lives, as far away from the rest of humanity as possible!

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects, and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, research guides, feature articles, source code, and more. "DZone is a developer’s dream," says PC Magazine.

Devada, Inc.
600 Park Offices Drive
Suite 150
Research Triangle Park, NC

888.678.0399 919.678.0300

Copyright © 2019 Devada, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.