Microservices Architecture Introduction

The microservices architecture is one of the most rapidly expanding architectural paradigms in commercial computing today. Since its introduction in a white paper by James Lewis and Martin Fowler it has since become a de-facto standard for developing large-scale commercial applications.

But despite the clear descriptions of the principles of this architecture in the Lewis/Fowler whitepaper, and also in later works such as Newman's that elaborate on the architecture, many development teams still struggle with basic practical questions about how to implement systems using the microservices architecture. In particular, they struggle with questions of how the microservices are invoked from a client application, how different client application styles affect their microservices implementation, and, most important, how to build their microservices efficiently.

It is that last point that is perhaps the most troublesome in practical microservices implementations. A benefit and a drawback of Services Oriented Architectures (including the microservices approach) is that services are often (but not necessarily) implemented as simple HTTP endpoints – but the simplicity and power of HTTP is sometimes outweighed by the latency inherent in process-to-process communication using HTTP. Overcoming that latency requires forethought and planning in your architecture in order to reduce unnecessary overhead.

Another issue with microservices architectures is that while they make it possible for developers to have more choices – for instance, by allowing for both Polyglot programming and Polyglot persistence – in fact the set of choices facing developers is too large – with too few guidelines in place for helping navigate. The result is that developers find it hard to decide where to start, and find few good examples of template applications of different styles that use microservices effectively.

In the Microservices application architecture style each application is composed of many discrete, network-connected components, termed microservices. The microservices architectural style can be seen as an evolution of the SOA (Services Oriented Architecture) architectural style. The key differences between the two are that while applications built using SOA services tended to become focused on technical integration issues, and the level of the services implemented were often very fine-grained technical APIs, the microservices approach instead stay focused on implementing clear business capabilities through larger-grained business APIs. An applicable rule in this case is that any microservice API should be directly related to a business entity.

But aside from the service design questions, perhaps the biggest difference is one of deployment style. For many years, applications have been packaged in a monolithic fashion – that is a team of developers would construct one large application that does everything required for a business need. Once built, that application would then be deployed multiple times across a farm of application servers. By contrast, in the microservices architectural style, several smaller applications that each implement only part of the whole are built and packaged independently.

When you look at the implications of packaging services independently, you see that five simple rules drive the implementation of applications built using the microservices architecture. They are:

- Deploy applications as sets of small, independent services – A single network- accessible service is the smallest deployable unit for a microservices application. Each service should run in its own process. This rule is sometimes stated as "one service per container", where "container" could mean a Docker container or any other lightweight deployment mechanism such as a Cloud Foundry runtime, or even a Tomcat or WebSphere Liberty server.
- Optimize services for a single function – In a traditional monolithic SOA approach a single application runtime would perform multiple business functions. In a microservices approach, there should be one and only one business function per service. This makes each service smaller and simpler to write and maintain. Robert Martin calls this the "Single Responsibility Principle".
- Communicate via REST API and message brokers – One of the drawbacks of the SOA approach was that there was an explosion of standards and options for implementing SOA services. The microservices approach instead tries to strictly limit the types of network connectivity that a service can implement to achieve maximum simplicity. Likewise, another rule for microservices is to avoid the tight coupling introduced by implicit communication through a database – all communication from service to service must be through the service API or at least must use an explicit communication pattern such as the Claim Check Pattern from Hohpe.
- Apply Per-service CI/CD – When building a large application comprised of many services, you soon realize that different services evolve at different rates. Letting each service have its own unique Continuous Integration/Continuous Delivery pipeline allows that evolution to proceed at is own natural pace – unlike in the monolithic approach where different aspects of the system were forced to all be released at the speed of the slowest- moving part of the system.

Apply Per-service HA/clustering decisions – Another realization when building large systems is that when it comes to clustering, not one size fits all. The monolithic approach of scaling all the services in the monolith at the same level often led to overutilization of some servers and underutilization of others – or even worse, starvation of some services by others when they monopolized all of the available shared resources such as thread pools. The reality is that in a large system, not all services need to scale and can be deployed in a minimum number of servers to conserve resources. Others require scaling up to very large numbers.

The power of the combination of these points (each of which will be referenced in the patterns below) and the benefits obtained from following them is the primary reason why the microservices architecture has become so popular. In this section, we will help address a few of those issues by introducing a set of simple patterns that can help developers understand how to apply microservices in a variety of different situations.

Microservices Architecture is the root pattern of this section. It explains why a microservices architecture aids in decoupling.

Domain Microservice introduces the most basic view of a microservice as implementing a concept from your business domain.

Adapter Microservice provides a mechanism to bring external systems into a *Microservices Architecture* without unduly creating direct dependencies on the external system.

Legacy Adapter Microservice embodies the common concept in enterprise systems that microservices can still be useful as a transitional step in the refactoring of back-end systems by providing a modern API to legacy data.

SaaS Adapter Microservice is a specialization of *Adapter Microservice* for a common case where the system being adapted is a third-party system.

Backend for Frontend introduces a way to mediate between the demands of a specific channel and the more general domain needs of a Business Microservice

Page Cache is a way of improving the performance of a *Backend For Frontend* when a microservice may return a large amount of data.

Polyglot Development is a way of not limiting your development teams to tools that may not be appropriate to a specific job at hand.

Circuit Breaker is a primary pattern for building applications built up of microservices that can avoid failure whenever downstream services fail.

Tolerant Reader is a way to design clients of microservices (including, importantly, upstream microservices) to be resiliant in the face of API changes in downstream microservices.

You can see how most of the patterns relate to each other in the diagram below:

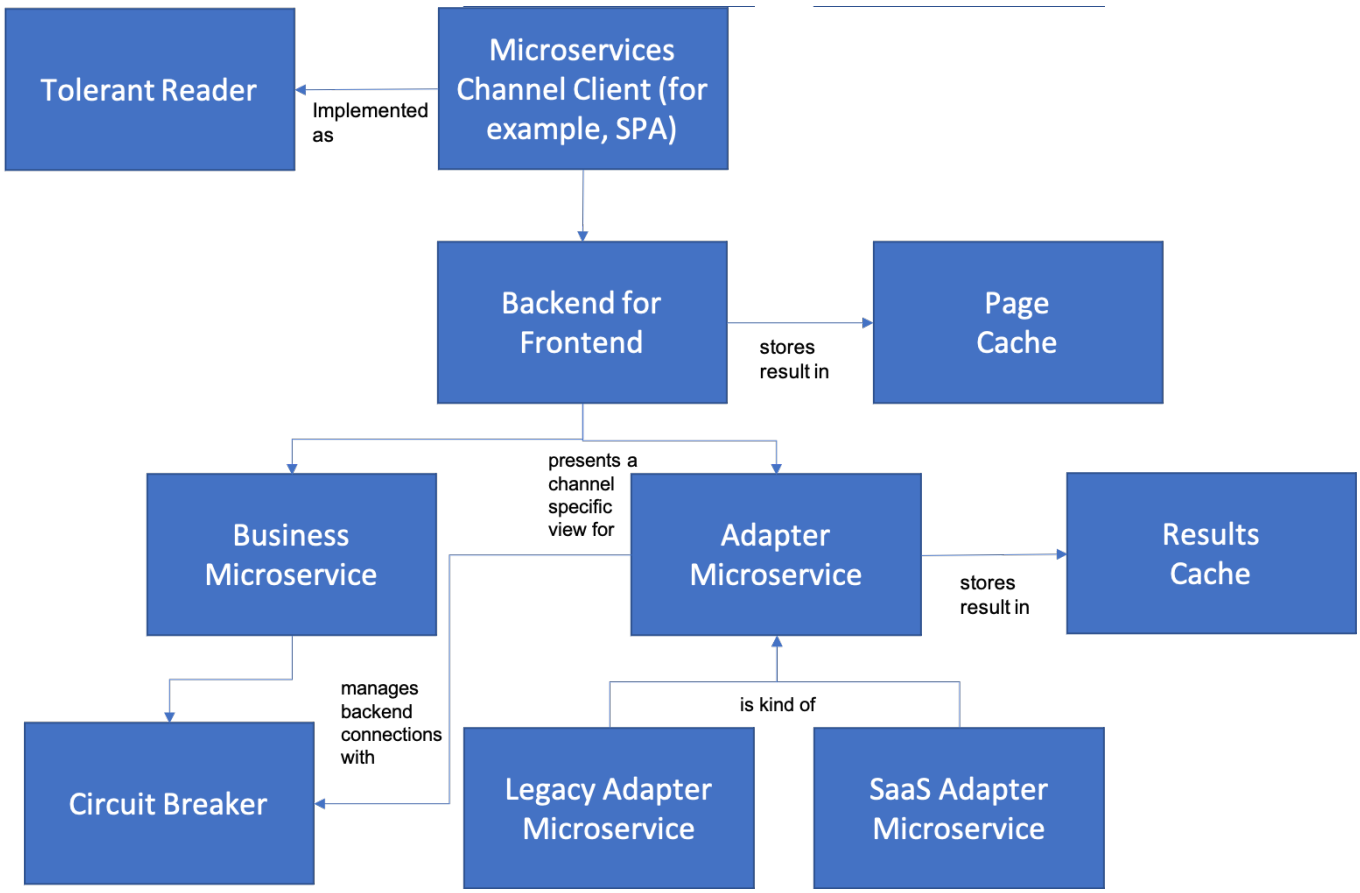

TABLE OF CONTENTS