Padrão: Arquitetura Microservice ou Microserviço

Contexto

Você está desenvolvendo uma aplicação cliente-servidor empresarial robusta. Ela precisa suportar uma variedade de diferentes clientes incluindo desktop, navegadores, navegadores moveis e aplicativos móveis nativos.

Além disso ela deve suportar expor uma api para consumo de parceiros/terceiros. Deve também. Deve também integrar com outras aplicações tanto via web services ou mensageria. A aplicação gerencia requisições HTTP, executando lógica de negocio, acessando bancos de dados, entregando mensagens para outros sistemas e retornando dcumentos html/json/xml como resposta. Existem componentes logicos correspondendo a diferentes areas funcionais da aplicação.

O problema

Qual é a arquitetura de desenvolvimento da aplicação?

A motivação

* Existem um time de desenvolvedores trabalhando na aplicação
* Novos membros do time precisam rapidamente tornar-se produtivos
* A aplicação precisa ser facil de entender e modificar
* Você quer praticar entrega contínua na aplicação
* Você precisa executar multiplas copias da aplicação em multiplas maquinas para satisfazer os requisitos de escalabilidade e disponibilidade
* Você quer obter vantagem das tecnologias emergentes (frameworks, linguagens de programação e etc)

Solução

Definir uma arquitetura que estrutura a aplicação como um conjunto de serviçoes colaborativos com baixo acoplamento. Esta abordagem corresponde ao eixo Y cubo de escala ( [Scale Cube](https://microservices.io/articles/scalecube.html)). Cada serviço implementa um conjunto de funções estritamente relacionadas.

Por exemplo, uma aplicação pode consistir de serviços como serviço de gerenciamento de pedidos, serviço de gerenciamento de clientes, e etc.

Serviços comunicam uns com os outros usando tanto protocolos sincronos como HTTP/REST quanto como protocolos assincronos como o AMQP. Serviços podem ser desenvolvidos e distribuídos de forma independente um do outro. Cada serviço possui seu próprio banco de dados de forma a se tornar desacoplado de outros serviços. A consistencia dos dados entre serviços é mantida usando o padrão Saga([Saga pattern](https://microservices.io/patterns/data/saga.html))

Exemplos

Aplicação de comercio eletrônico fictícia

Vamos imaginar que você está construindo uma aplicação de comercio eletrônico que coleta pedidos de clientes, verifica o estoque e crédito disponível, e prepara o pedido para entrega. A aplicação consiste de vários componentes incluindo o FrenteDeLojaUI, que implementa uma interface de usuário, juntamente com serviços de back-end para verficação do crédito, manutenção do estoque e envio dos pedidos.

A aplicação consiste de um conjunto de serviços



Mostre-me o codigo

Por favor me veja a [aplicacao de exemplo de chris richardson](http://eventuate.io/exampleapps.html) . Estes exemplos estão no [github.com](http://github.com/) e ilustram varios aspectos da arquitetura de microserviços.

Contexto final

Benefícios

Esta solição tem um numero de benefícios:

* Permite a entrega contínua e o desenvolvimento de grandes e complexas aplicações.
  + Melhor capacidade de teste - serviços são menores e mais rápidos de testar
  + Melhor distribuição - serviços podem ser distribuidos e implantados de forma independente.
  + Permite a você organizar o esforço de desenvolvimento por meio de múltiplos times auto motivados. Cada um é dono e responsável por um ou mais serviços individuais. Cada time pode desenvolver, implantar e escalar seus serviços de forma independente de todos os outros times.
* Cada micro-serviço é relativamente pequeno
  + Fácil para o desenvolvedor entender
  + A interface de desenvolvimento fica mais rápida e o desenvolvedor torna-se mais produtivo
  + A aplicação inicia mais rápido, que faz o desenvolvedor mais produtivo, e aumenta a velocidade da implantação (deploy)
* Isolamento de falha melhorada. Por exemplo, se houver um vazamento de memória em um serviço então somente esse serviço será afetado. Os outros serviços continuarão a gerenciar suas requisições. Em comparação, um componente apresentando comportamento adverso em uma arquitetura monolitica pode deixar indisponível o sistema inteiro.
* Elimina qualquer comprometimento de longo prazo com a pilha de tecnologias adotada. Quando desenvolver um novo serviço você pode aplicar uma nova pilha de tecnologias. De forma similar, quando fizer muitas mudanças em um serviço existente voce pode reescrever na nova tecnologia.

Desvantagens

Esta solução tem um numero de desvantagens:

* Desenvolvedores precisam lidar com complexidade adicional de criar um sistema distribuído.
  + Maioria das ferramentas de desenvolvimento são orientadas a construir aplicações monolíticas e não fornecem suporte explicito para desenvolvimento de aplicações distribuídas.
  + Teste integrado é mais dificil
  + Desenvolvedores precisam impementar o mecanismo de comunicação entre os serviços "inter-service".
  + Implementar casos de uso que expandem para múltiplos serviços sem usar transação distribuída é difícil.
  + Implementar casos de uso que espandem para multiples serviços requerem uma cuidadosa coordenação entre os times
* Deployment complexity. In production, there is also the operational complexity of deploying and managing a system comprised of many different service types.
* Increased memory consumption. The microservice architecture replaces N monolithic application instances with NxM services instances. If each service runs in its own JVM (or equivalent), which is usually necessary to isolate the instances, then there is the overhead of M times as many JVM runtimes. Moreover, if each service runs on its own VM (e.g. EC2 instance), as is the case at Netflix, the overhead is even higher.

Issues

There are many issues that you must address.

When to use the microservice architecture?

One challenge with using this approach is deciding when it makes sense to use it. When developing the first version of an application, you often do not have the problems that this approach solves. Moreover, using an elaborate, distributed architecture will slow down development. This can be a major problem for startups whose biggest challenge is often how to rapidly evolve the business model and accompanying application. Using Y-axis splits might make it much more difficult to iterate rapidly. Later on, however, when the challenge is how to scale and you need to use functional decomposition, the tangled dependencies might make it difficult to decompose your monolithic application into a set of services.

How to decompose the application into services?

Another challenge is deciding how to partition the system into microservices. This is very much an art, but there are a number of strategies that can help:

* [Decompose by business capability](https://microservices.io/patterns/decomposition/decompose-by-business-capability.html) and define services corresponding to business capabilities.
* [Decompose by domain-driven design subdomain](https://microservices.io/patterns/decomposition/decompose-by-subdomain.html).
* Decompose by verb or use case and define services that are responsible for particular actions. e.g. a Shipping Service that’s responsible for shipping complete orders.
* Decompose by by nouns or resources by defining a service that is responsible for all operations on entities/resources of a given type. e.g. an Account Service that is responsible for managing user accounts.

Ideally, each service should have only a small set of responsibilities. (Uncle) Bob Martin talks about designing classes using the [Single Responsibility Principle (SRP)](http://www.objectmentor.com/resources/articles/srp.pdf). The SRP defines a responsibility of a class as a reason to change, and states that a class should only have one reason to change. It make sense to apply the SRP to service design as well.

Another analogy that helps with service design is the design of Unix utilities. Unix provides a large number of utilities such as grep, cat and find. Each utility does exactly one thing, often exceptionally well, and can be combined with other utilities using a shell script to perform complex tasks.

How to maintain data consistency?

In order to ensure loose coupling, each service has its own database. Maintaining data consistency between services is a challenge because 2 phase-commit/distributed transactions is not an option for many applications. An application must instead use the [Saga pattern](https://microservices.io/patterns/data/saga.html). A service publishes an event when its data changes. Other services consume that event and update their data. There are several ways of reliably updating data and publishing events including [Event Sourcing](https://microservices.io/patterns/data/event-sourcing.html) and [Transaction Log Tailing](https://microservices.io/patterns/data/transaction-log-tailing.html).

How to implement queries?

Another challenge is implementing queries that need to retrieve data owned by multiple services.

* The [API Composition](https://microservices.io/patterns/data/api-composition.html) and [Command Query Responsibility Segregation (CQRS)](https://microservices.io/patterns/data/cqrs.html) patterns.

Related patterns

There are many patterns related to the microservices pattern. The [Monolithic architecture](https://microservices.io/patterns/monolithic.html) is an alternative to the microservice architecture. The other patterns address issues that you will encounter when applying the microservice architecture.



* Decomposition patterns
  + [Decompose by business capability](https://microservices.io/patterns/decomposition/decompose-by-business-capability.html)
  + [Decompose by subdomain](https://microservices.io/patterns/decomposition/decompose-by-subdomain.html)
* The [Database per Service pattern](https://microservices.io/patterns/data/database-per-service.html) describes how each service has its own database in order to ensure loose coupling.
* The [API Gateway pattern](https://microservices.io/patterns/apigateway.html) defines how clients access the services in a microservice architecture.
* The [Client-side Discovery](https://microservices.io/patterns/client-side-discovery.html) and [Server-side Discovery](https://microservices.io/patterns/server-side-discovery.html) patterns are used to route requests for a client to an available service instance in a microservice architecture.
* The Messaging and Remote Procedure Invocation patterns are two different ways that services can communicate.
* The [Single Service per Host](https://microservices.io/patterns/deployment/single-service-per-host.html) and [Multiple Services per Host](https://microservices.io/patterns/deployment/multiple-services-per-host.html) patterns are two different deployment strategies.
* Cross-cutting concerns patterns: [Microservice chassis pattern](https://microservices.io/patterns/microservice-chassis.html) and [Externalized configuration](https://microservices.io/patterns/externalized-configuration.html)
* Testing patterns: [Service Component Test](https://microservices.io/patterns/testing/service-component-test.html) and [Service Integration Contract Test](https://microservices.io/patterns/testing/service-integration-contract-test.html)
* [Circuit Breaker](https://microservices.io/patterns/reliability/circuit-breaker.html)
* [Access Token](https://microservices.io/patterns/security/access-token.html)
* Observability patterns:
  + [Log aggregation](https://microservices.io/patterns/observability/application-logging.html)
  + [Application metrics](https://microservices.io/patterns/observability/application-metrics.html)
  + [Audit logging](https://microservices.io/patterns/observability/audit-logging.html)
  + [Distributed tracing](https://microservices.io/patterns/observability/distributed-tracing.html)
  + [Exception tracking](https://microservices.io/patterns/observability/exception-tracking.html)
  + [Health check API](https://microservices.io/patterns/observability/health-check-api.html)
  + [Log deployments and changes](https://microservices.io/patterns/observability/log-deployments-and-changes.html)
* UI patterns:
  + [Server-side page fragment composition](https://microservices.io/patterns/ui/server-side-page-fragment-composition.html)
  + [Client-side UI composition](https://microservices.io/patterns/ui/client-side-ui-composition.html)

Known uses

Most large scale web sites including [Netflix](http://techblog.netflix.com/), [Amazon](http://highscalability.com/blog/2007/9/18/amazon-architecture.html) and [eBay](http://www.addsimplicity.com/downloads/eBaySDForum2006-11-29.pdf) have evolved from a monolithic architecture to a microservice architecture.

Netflix, which is a very popular video streaming service that’s responsible for up to 30% of Internet traffic, has a large scale, service-oriented architecture. They handle over a billion calls per day to their video streaming API from over 800 different kinds of devices. Each API call fans out to an average of six calls to backend services.

[Amazon.com](http://amazon.com/) originally had a two-tier architecture. In order to scale they migrated to a service-oriented architecture consisting of hundreds of backend services. Several applications call these services including the applications that implement the [Amazon.com](http://amazon.com/) website and the web service API. The [Amazon.com](http://amazon.com/) website application calls 100-150 services to get the data that used to build a web page.

The auction site [ebay.com](http://ebay.com/) also evolved from a monolithic architecture to a service-oriented architecture. The application tier consists of multiple independent applications. Each application implements the business logic for a specific function area such as buying or selling. Each application uses X-axis splits and some applications such as search use Z-axis splits. [Ebay.com](http://ebay.com/) also applies a combination of X-, Y- and Z-style scaling to the database tier.

There are [numerous other examples](https://microservices.io/articles/whoisusingmicroservices.html) of companies using the microservice architecture.

Examples

Chris Richardson has [examples](http://eventuate.io/exampleapps.html) of microservices-based applications.

See also

See my [Code Freeze 2018 keynote](https://microservices.io/microservices/news/2018/02/20/no-such-thing-as-a-microservice.html), which provides a good introduction to the microservice architecture.