Pattern: Microservice Architecture

Context

You are developing a server-side enterprise application. It must support a variety of different clients including desktop browsers, mobile browsers and native mobile applications. The application might also expose an API for 3rd parties to consume. It might also integrate with other applications via either web services or a message broker. The application handles requests (HTTP requests and messages) by executing business logic; accessing a database; exchanging messages with other systems; and returning a HTML/JSON/XML response. There are logical components corresponding to different functional areas of the application.

Problem

What's the application's deployment architecture?

Forces

- There is a team of developers working on the application
- New team members must quickly become productive
- The application must be easy to understand and modify
- You want to practice continuous deployment of the application
- You must run multiple instances of the application on multiple machines in order to satisfy scalability and availability requirements
- You want to take advantage of emerging technologies (frameworks, programming languages, etc)

Solution

Define an architecture that structures the application as a set of loosely coupled, collaborating services. This approach corresponds to the Y-axis of the Scale Cube (/articles/scalecube.html). Each service is:

- Highly maintainable and testable - enables rapid and frequent development and deployment
- Loosely coupled with other services - enables a team to work independently the majority of time on their service(s) without being impacted by changes to other services and without affecting other services
- Independently deployable - enables a team to deploy their service without having to coordinate with other teams
- Capable of being developed by a small team - essential for high productivity by avoiding the high communication head of large teams

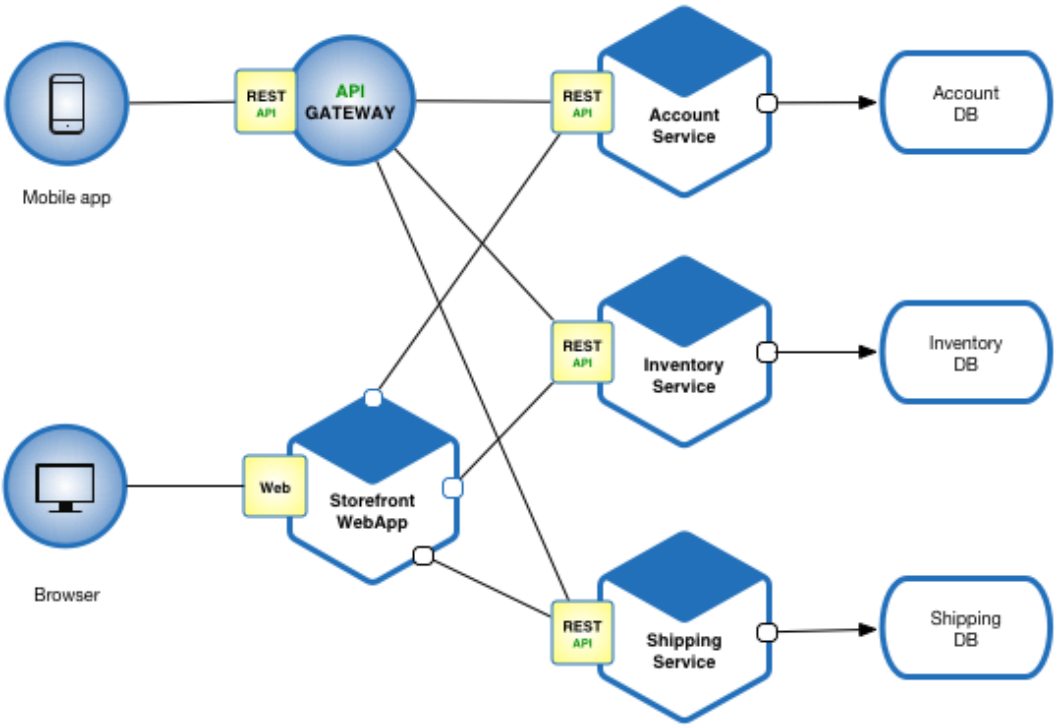
Services communicate using either synchronous protocols such as HTTP/REST or asynchronous protocols such as AMQP. Services can be developed and deployed independently of one another. Each service has its own database (data/database-per-service.html) in order to be decoupled from other services. Data consistency between services is maintained using the Saga pattern (data/saga.html)

To learn more about the nature of a service, please read this article (http://chrisrichardson.net/post/microservices/general/2019/02/16/whats-a-service-part-1.html).

Examples

Fictitious e-commerce application

Let's imagine that you are building an e-commerce application that takes orders from customers, verifies inventory and available credit, and ships them. The application consists of several components including the StoreFrontUI, which implements the user interface, along with some backend services for checking credit, maintaining inventory and shipping orders. The application consists of a set of services.



Show me the code

Please see the example applications developed by Chris Richardson (http://eventuate.io/exampleapps.html). These examples on Github illustrate various aspects of the microservice architecture.

Resulting context

Benefits

This solution has a number of benefits:

- Enables the continuous delivery and deployment of large, complex applications.
    - Improved maintainability - each service is relatively small and so is easier to understand and change
    - Better testability - services are smaller and faster to test
    - Better deployability - services can be deployed independently
    - It enables you to organize the development effort around multiple, autonomous teams. Each (so called two pizza) team owns and is responsible for one or more services. Each team can develop, test, deploy and scale their services independently of all of the other teams.
- Each microservice is relatively small:
    - Easier for a developer to understand
    - The IDE is faster making developers more productive
    - The application starts faster, which makes developers more productive, and speeds up deployments
- Improved fault isolation. For example, if there is a memory leak in one service then only that service will be affected. The other services will continue to handle requests. In comparison, one misbehaving component of a monolithic architecture can bring down the entire system.
- Eliminates any long-term commitment to a technology stack. When developing a new service you can pick a new technology stack. Similarly, when making major changes to an existing service you can rewrite it using a new technology stack.

Drawbacks

This solution has a number of drawbacks:

- Developers must deal with the additional complexity of creating a distributed system:
    - Developers must implement the inter-service communication mechanism and deal with partial failure
    - Implementing requests that span multiple services is more difficult
    - Testing the interactions between services is more difficult
    - Implementing requests that span multiple services requires careful coordination between the teams
    - Developer tools/IDEs are oriented on building monolithic applications and don't provide explicit support for developing distributed applications.
- Deployment complexity. In production, there is also the operational complexity of deploying and managing a system comprised of many different services.
- 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 (/patterns/decomposition/decompose-by-business-capability.html) and define services corresponding to business capabilities.
- Decompose by domain-driven design subdomain (/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 (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 (data/event-sourcing.html) and Transaction Log Tailing (data/transaction-log-tailing.html).

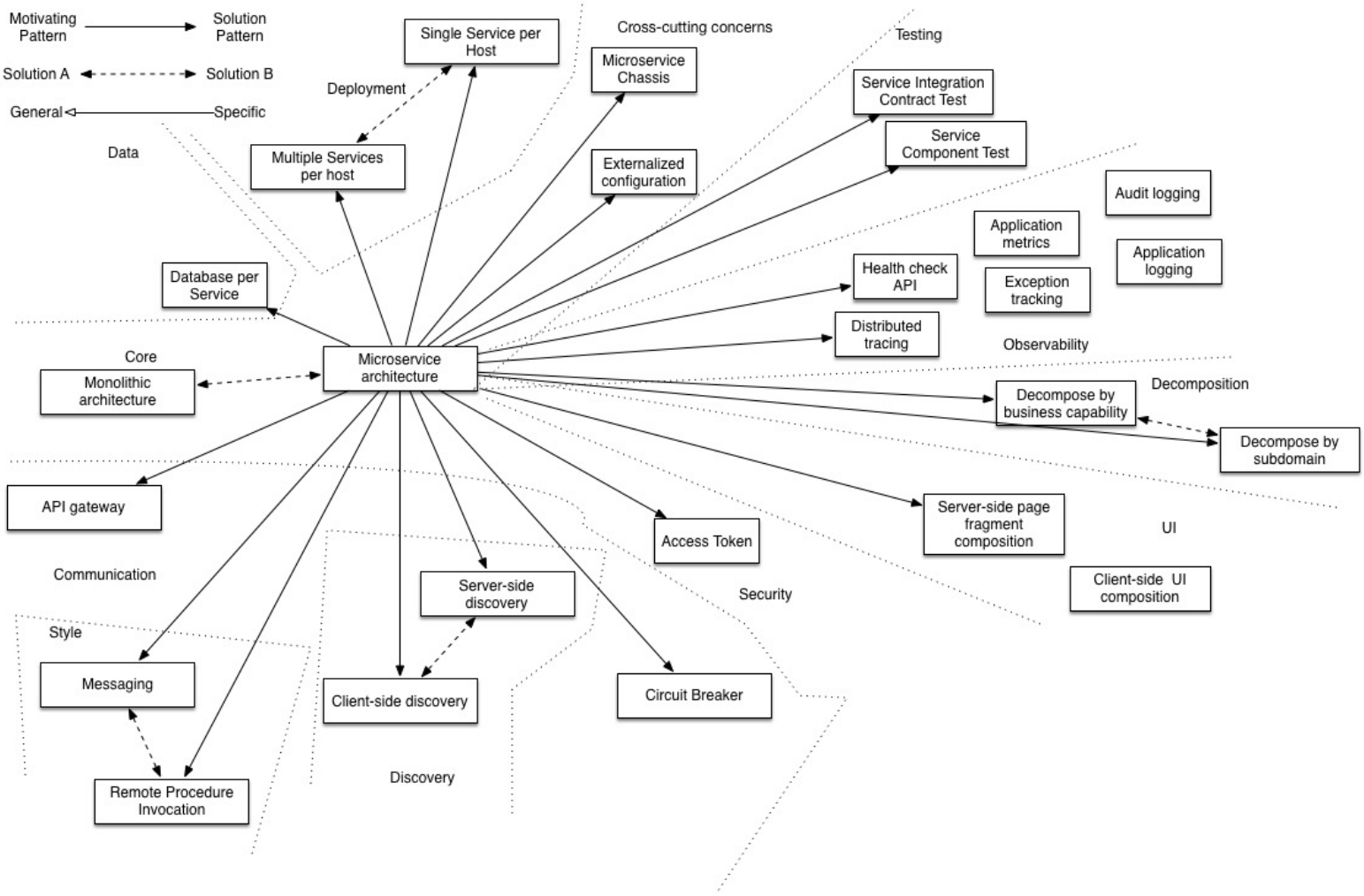
How to implement queries?

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

- The API Composition (data/api-composition.html) and Command Query Responsibility Segregation (CQRS) (data/cqrs.html) patterns.

Related patterns

There are many patterns related to the microservices pattern. The Monolithic architecture (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 (/patterns/decomposition/decompose-by-business-capability.html)
  - Decompose by subdomain (/patterns/decomposition/decompose-by-subdomain.html)
- The Database per Service pattern (data/database-per-service.html) describes how each service has its own database in order to ensure loose coupling.
- The API Gateway pattern (apigateway.html) defines how clients access the services in a microservice architecture.
- The Client-side Discovery (client-side-discovery.html) and Server-side Discovery (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 (deployment/single-service-per-host.html) and Multiple Services per Host (deployment/multiple-services-per-host.html) patterns are two different deployment strategies.
- Cross-cutting concerns patterns: Microservice chassis pattern (/patterns/microservice-chassis.html) and Externalized configuration (/patterns/externalized-configuration.html)

- Testing patterns: Service Component Test (testing/service-component-test.html) and Service Integration Contract Test (testing/service-integration-contract-test.html)
- Circuit Breaker (/patterns/reliability/circuit-breaker.html)
- Access Token (/patterns/security/access-token.html)
- Observability patterns:
    - Log aggregation (/patterns/observability/application-logging.html)
    - Application metrics (/patterns/observability/application-metrics.html)
    - Audit logging (/patterns/observability/audit-logging.html)
    - Distributed tracing (/patterns/observability/distributed-tracing.html)
    - Exception tracking (/patterns/observability/exception-tracking.html)
    - Health check API (/patterns/observability/health-check-api.html)
    - Log deployments and changes (/patterns/observability/log-deployments-and-changes.html)
- UI patterns:
    - Server-side page fragment composition (/patterns/ui/server-side-page-fragment-composition.html)
    - Client-side UI composition (/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 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 website and the web service API. The Amazon.com website application calls 100-150 services to get the data that used to build a web page.

The auction site 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 also applies a combination of X-, Y- and Z-style scaling to the database tier.

There are numerous other examples (../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 (/microservices/news/2018/02/20/no-such-thing-as-a-microservice.html), which provides a good introduction to the microservice architecture.