Book

on

Micro Services Architecture

Chapter 1

Monolithic Architecture

When 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.

1.0 Expectations of the Application Deployment Architecture

In the modern application development context following are the expectations:

•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)

1.1 Conventional Solution

Build an application with a monolithic architecture. For example:

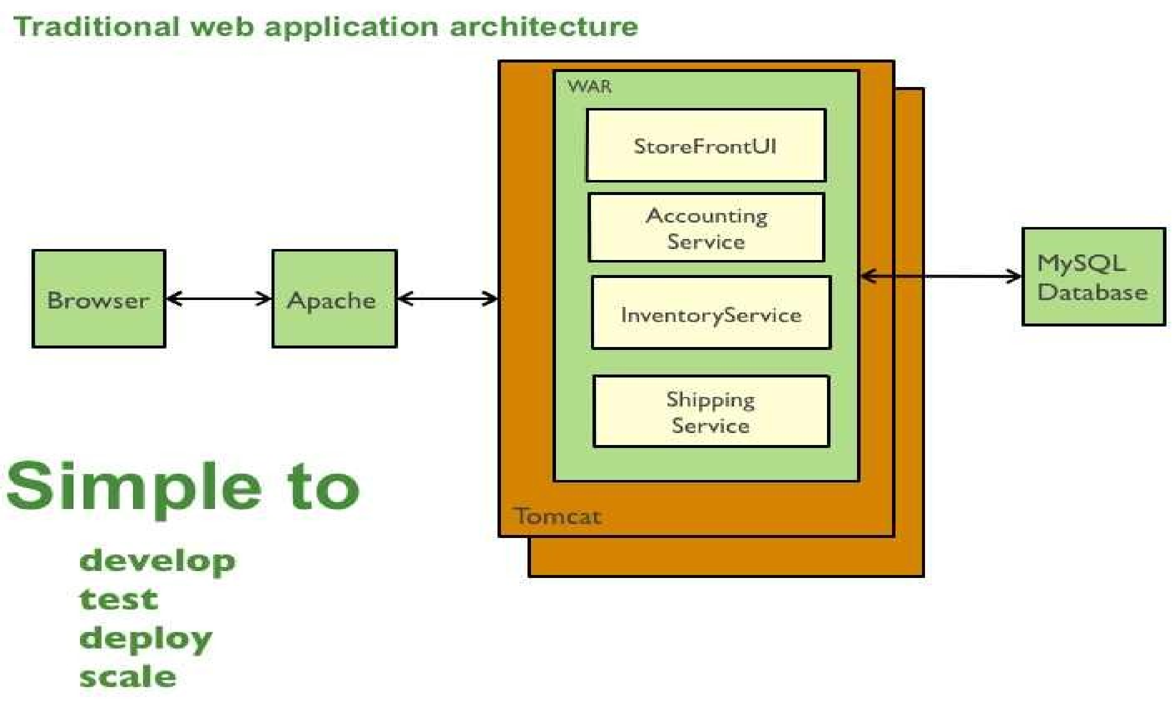
•a single Java WAR file.

•a single directory hierarchy of Rails or NodeJS code

Example

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 is deployed as a single monolithic application. For example, a Java web application consists of a single WAR file that runs on a web container such as Tomcat. A Rails application consists of a single directory hierarchy deployed using either, for example, Phusion Passenger on Apache/Nginx or JRuby on Tomcat. You can run multiple instances of the application behind a load balancer in order to scale and improve availability.



1.3 Benefits of Monolithic Architecture :

This solution has a number of benefits:

- Simple to develop - the goal of current development tools and IDEs is to support the development of monolithic applications
- Simple to deploy - you simply need to deploy the WAR file (or directory hierarchy) on the appropriate runtime
- Simple to scale - you can scale the application by running multiple copies of the application behind a load balancer

1.4 Limitations of Monolithic Architecture

However, once the application becomes large and the team grows in size, this approach has a number of drawbacks that become increasingly significant:

- The large monolithic code base intimidates developers, especially ones who are new to the team. The application can be difficult to understand and modify. As a result, development typically slows down. Also, because there are not hard module boundaries, modularity breaks down over time. Moreover, because it can be difficult to understand how to correctly implement a change the quality of the code declines over time. It's a downwards spiral.
- Overloaded IDE - the larger the code base the slower the IDE and the less productive developers are.
- Overloaded web container - the larger the application the longer it takes to start up. This had have a huge impact on developer productivity because of time wasted waiting for the container to start. It also impacts deployment too.

•Continuous deployment is difficult - a large monolithic application is also an obstacle to frequent deployments. In order to update one component you have to redeploy the entire application. This will interrupt background tasks (e.g. Quartz jobs in a Java application), regardless of whether they are impacted by the change, and possibly cause problems. There is also the chance that components that haven’t been updated will fail to start correctly. As a result, the risk associated with redeployment increases, which discourages frequent updates. This is especially a problem for user interface developers, since they usually need to iterative rapidly and redeploy frequently.

•Scaling the application can be difficult - a monolithic architecture is that it can only scale in one dimension. On the one hand, it can scale with an increasing transaction volume by running more copies of the application. Some clouds can even adjust the number of instances dynamically based on load. But on the other hand, this architecture can’t scale with an increasing data volume. Each copy of application instance will access all of the data, which makes caching less effective and increases memory consumption and I/O traffic. Also, different application components have different resource requirements - one might be CPU intensive while another might memory intensive. With a monolithic architecture we cannot scale each component independently

•Obstacle to scaling development - A monolithic application is also an obstacle to scaling development. Once the application gets to a certain size its useful to divide up the engineering organization into teams that focus on specific functional areas. For example, we might want to have the UI team, accounting team, inventory team, etc. The trouble with a monolithic application is that it prevents the teams from working independently. The teams must coordinate their development efforts and redeployments. It is much more difficult for a team to make a change and update production.

•Requires a long-term commitment to a technology stack - a monolithic architecture forces you to be married to the technology stack (and in some cases, to a particular version of that technology) you chose at the start of development . With a monolithic application, can be difficult to incrementally adopt a newer technology. For example, let’s imagine that you chose the JVM. You have some language choices since as well as Java you can use other JVM languages that inter-operate nicely with Java such as Groovy and Scala. But components written in non-JVM languages do not have a place within your monolithic architecture. Also, if your application uses a platform framework that subsequently becomes obsolete then it can be challenging to incrementally migrate the application to a newer and better framework. It’s possible that in order to adopt a newer platform framework you have to rewrite the entire application, which is a risky undertaking.

Chapter -2

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.

2.1 Expectations of New Development Architecture :

- 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)

2.2 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. 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

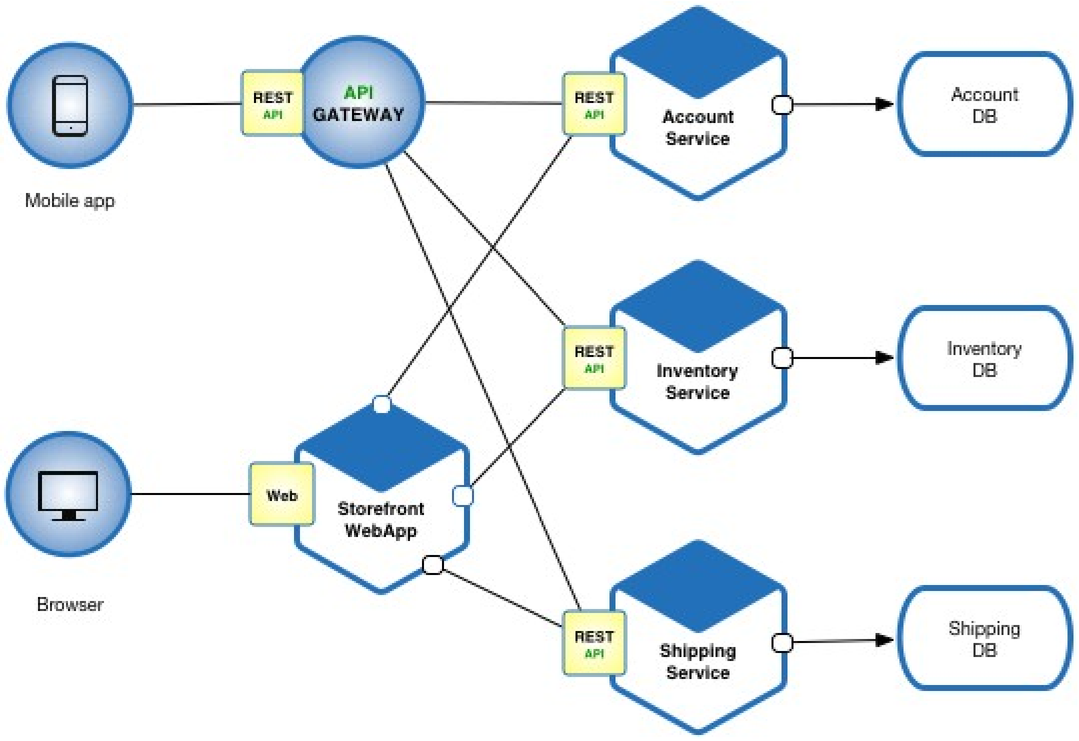
Services communicate using either synchronous protocols such as HTTP/REST or asynchronous protocols such as AMQP. Services can be developed and deployed independently of one another. Each service has its own database in order to be decoupled from other services. Data consistency between services is maintained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

To learn more about the nature of a service, please read this article.

2.3 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.



2.4 Benefits of Micro Services Architecture

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.

2.5 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.

2.6 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.

2.7 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 and define services corresponding to business capabilities.
- Decompose by domain-driven design subdomain.
- 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). 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.

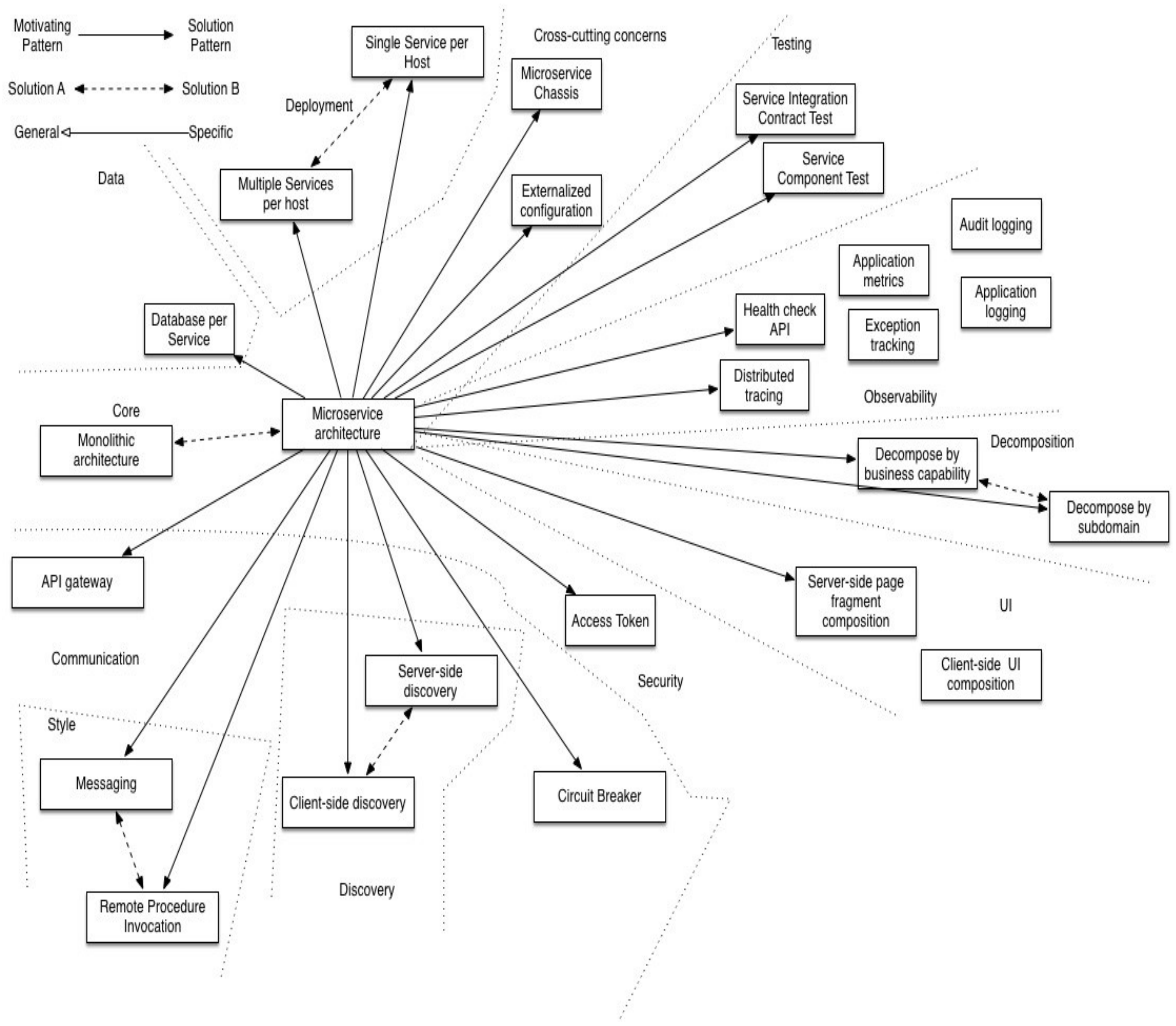
Another analogy that helps with service design is the design of Unix utilities. Unix provides a large number of utilities such as grep, cat and find. Each utility does exactly one thing, often exceptionally well, and can be combined with other utilities using a shell script to perform complex tasks.

2.7 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. 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 and Transaction Log Tailing.

Related patterns

There are many patterns related to the microservices pattern. The Monolithic architecture 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
    - Decompose by subdomain
- The Database per Service pattern describes how each service has its own database in order to ensure loose coupling.
- The API Gateway pattern defines how clients access the services in a microservice architecture.
- The Client-side Discovery and Server-side Discovery 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 and Multiple Services per Host patterns are two different deployment strategies.
- Cross-cutting concerns patterns: Microservice chassis pattern and Externalized configuration
- Testing patterns: Service Component Test and Service Integration Contract Test

- Circuit Breaker
- Access Token
- Observability patterns:
    - Log aggregation
    - Application metrics
    - Audit logging
    - Distributed tracing
    - Exception tracking
    - Health check API
    - Log deployments and changes
- UI patterns:
    - Server-side page fragment composition
    - Client-side UI composition

2.8 Known uses

Most large scale web sites including Netflix, Amazon and eBay 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.

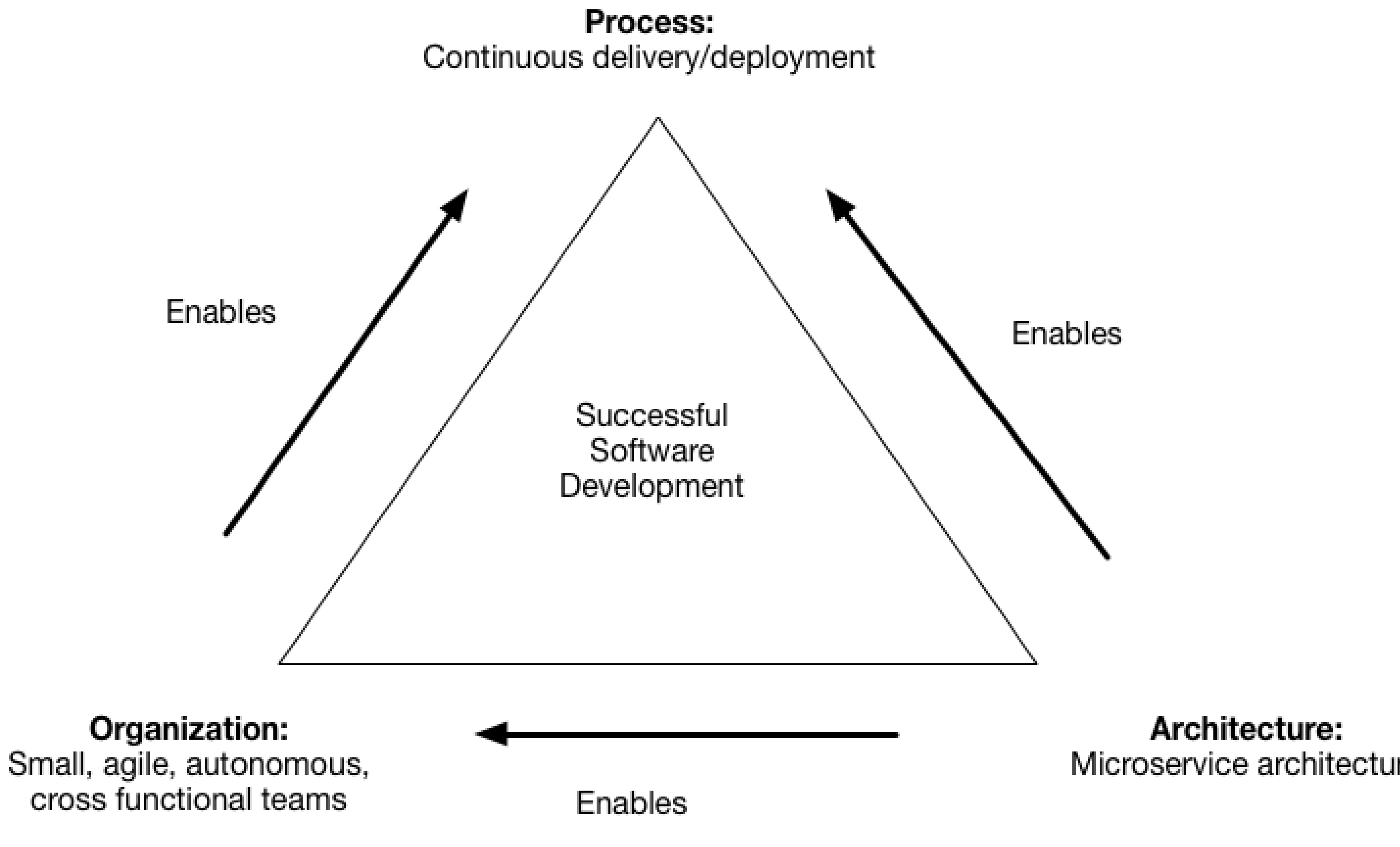
The auction site ebay.com also evolved from a monolithic architecture to a service-oriented architecture. The application tier consists of multiple independent applications. Each application implements the business logic for a specific function area such as buying or selling. Each application uses X-axis 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.

Chapter – 3

Decomposition of Monolith

3.1 Decompose by business capability

You are developing a large, complex application and want to use the microservice architecture. The microservice architecture structures an application as a set of loosely coupled services. The goal of the microservice architecture is to accelerate software development by enabling continuous delivery/deployment.


Process:
Continuous delivery/deployment
Enables
Enables
Successful Software Development
Organization:
Small, agile, autonomous, cross functional teams
Enables
Architecture:
Microservice architectu

The microservice architecture does this in two ways:

1.Simplifies testing and enables components to deployed independently

2.Structures the engineering organization as a collection of small (6-10 members), autonomous teams, each of which is responsible for one or more services

These benefits are not automatically guaranteed. Instead, they can only be achieved by the careful functional decomposition of the application into services.

A service must be small enough to be developed by a small team and to be easily tested. A useful guideline from object-oriented design (OOD) is the Single Responsibility Principle (SRP). 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 and design services that are cohesive and implement a small set of strongly related functions.

The application also be decomposed in a way so that most new and changed requirements only affect a single service. That is because changes that affect multiple services requires coordination across multiple teams, which slows down development. Another useful principle from OOD is the Common Closure Principle (CCP), which states that classes that change for the same reason should be in the same package. Perhaps, for instance, two classes implement different aspects of the same business rule. The goal is that when that business rule changes developers, only need to change code in a small number - ideally only one - of packages. This kind of thinking makes sense when designing services since it will help ensure that each change should impact only one service.

3.1.1 Expectation from a Decomposition Model

- The architecture must be stable
- Services must be cohesive. A service should implement a small set of strongly related functions.
- Services must conform to the Common Closure Principle - things that change together should be packaged together - to ensure that each change affect only one service
- Services must be loosely coupled - each service as an API that encapsulates its implementation. The implementation can be changed without affecting clients
- A service should be testable
- Each service be small enough to be developed by a “two pizza” team, i.e. a team of 6-10 people
- Each team that owns one or more services must be autonomous. A team must be able to develop and deploy their services with minimal collaboration with other teams.

3.1.2. Solution

Define services corresponding to business capabilities. A business capability is a concept from business architecture modeling. It is something that a business does in order to generate value. A business capability often corresponds to a business object, e.g.

- Order Management is responsible for orders
- Customer Management is responsible for customers

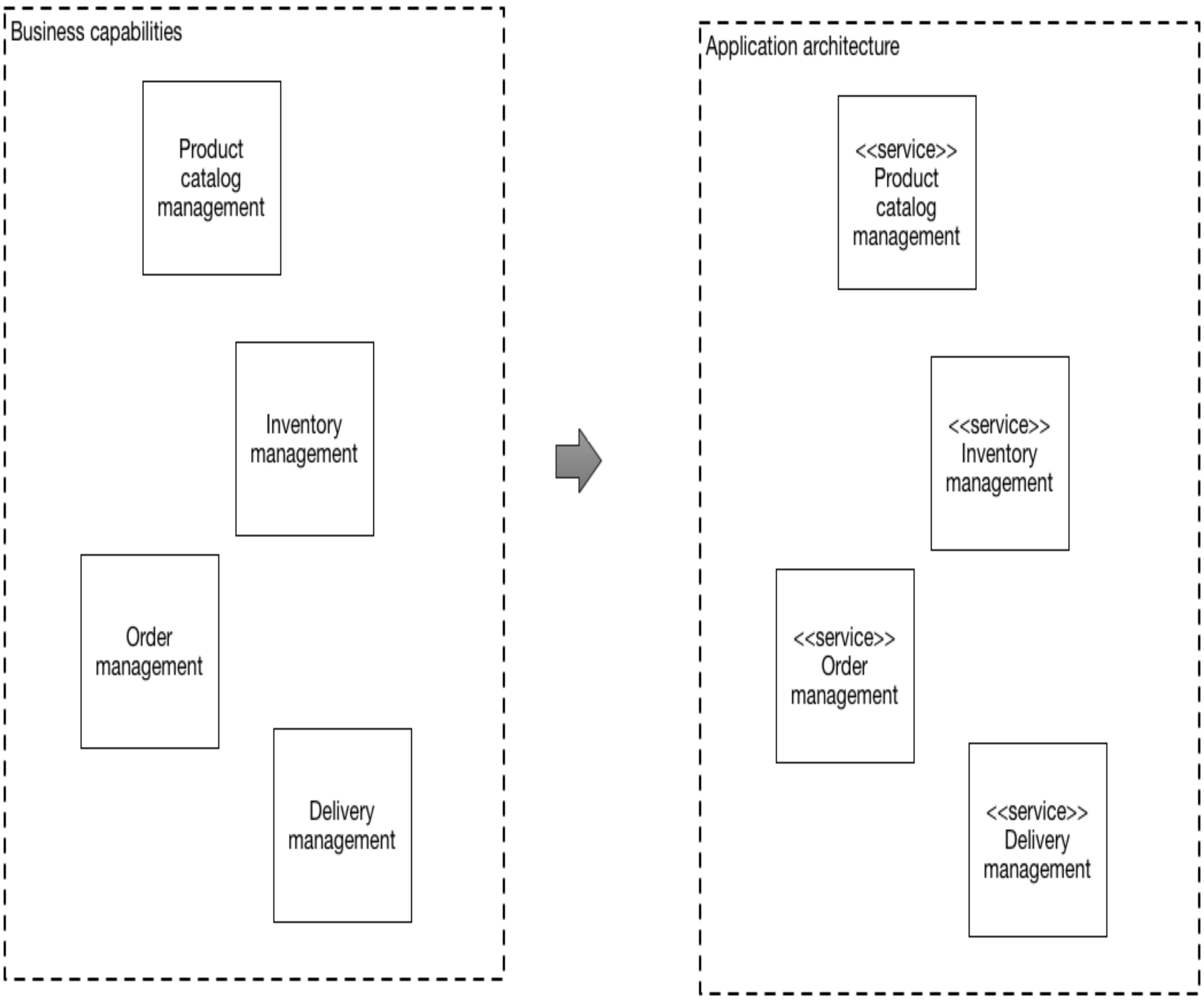
Business capabilities are often organized into a multi-level hierarchy. For example, an enterprise application might have top-level categories such as Product/Service development, Product/Service delivery, Demand generation, etc.

Examples

The business capabilities of an online store include:

- Product catalog management
- Inventory management
- Order management
- Delivery management
- …

The corresponding microservice architecture would have services corresponding to each of these capabilities.


Business capabilities
Product catalog management
Inventory management
Order management
Delivery management
Application architecture
<<service>> Product catalog management
<<service>> Inventory management
<<service>> Order management
<<service>> Delivery management

This pattern has the following benefits:

- Stable architecture since the business capabilities are relatively stable
- Development teams are cross-functional, autonomous, and organized around delivering business value rather than technical features
- Services are cohesive and loosely coupled

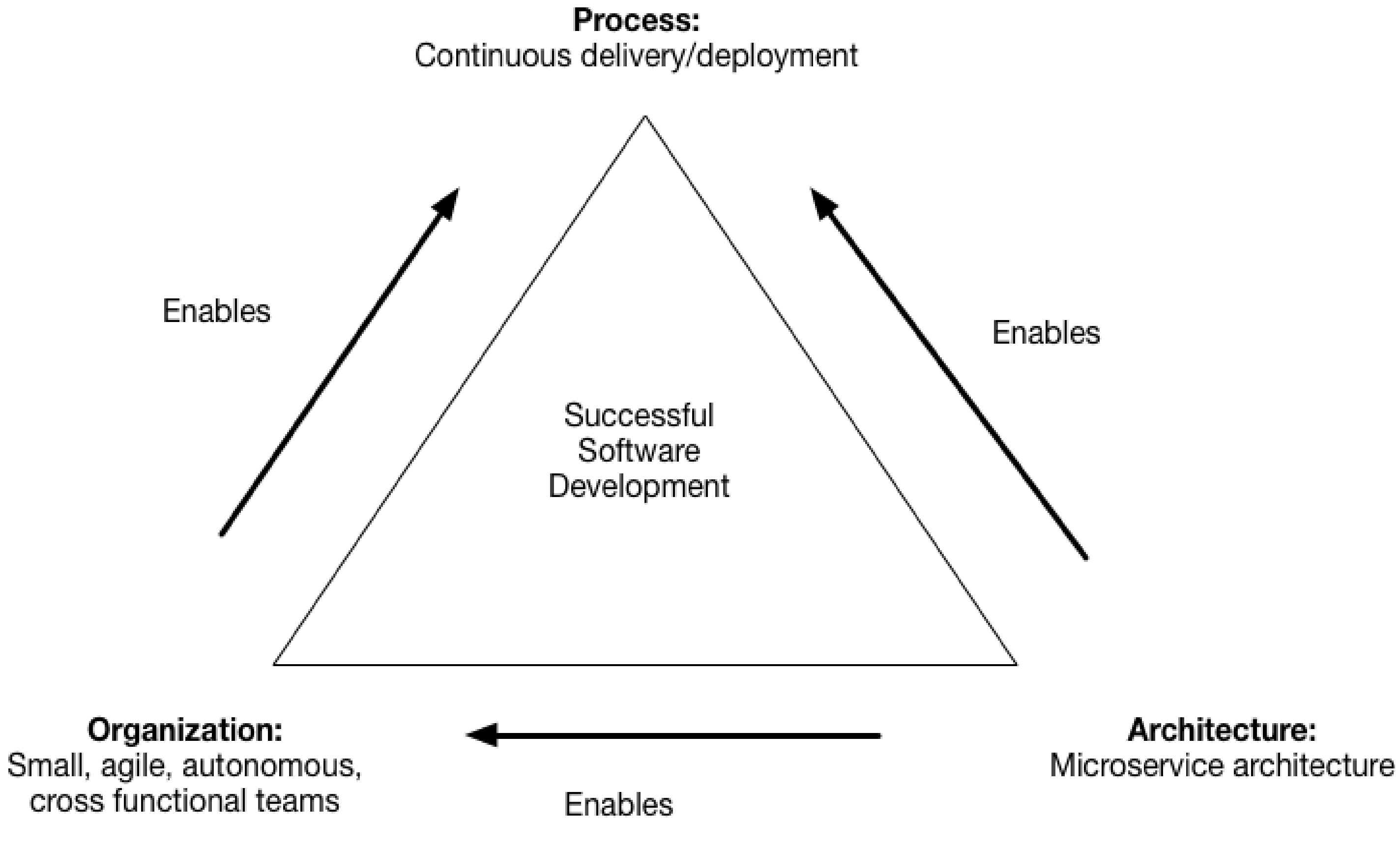
3.1.3 Issues to be Addressed

- **How to identify business capabilities?** Identifying business capabilities and hence services requires an understanding of the business. An organization’s business capabilities are identified by analyzing the organization’s purpose, structure, business processes, and areas of expertise. Bounded contexts are best identified using an iterative process. Good starting points for identifying business capabilities are:

- organization structure - different groups within an organization might correspond to business capabilities or business capability groups.
- high-level domain model - business capabilities often correspond to domain objects

3.2 Decompose by subdomain

You are developing a large, complex application and want to use the microservice architecture. The microservice architecture structures an application as a set of loosely coupled services. The goal of the microservice architecture is to accelerate software development by enabling continuous delivery/deployment.


Process:
Continuous delivery/deployment
Enables
Enables
Successful Software Development
Organization:
Small, agile, autonomous, cross functional teams
Enables
Architecture:
Microservice architecture

The microservice architecture does this in two ways:

1. Simplifies testing and enables components to deployed independently
2. Structures the engineering organization as a collection of small (6-10 members), autonomous teams, each of which is responsible for one or more services

These benefits are not automatically guaranteed. Instead, they can only be achieved by the careful functional decomposition of the application into services.

A service must be small enough to be developed by a small team and to be easily tested. A useful guideline from object-oriented design (OOD) is the Single Responsibility Principle (SRP). 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 and design services that are cohesive and implement a small set of strongly related functions.

The application also be decomposed in a way so that most new and changed requirements only affect a single service. That is because changes that affect multiple services requires coordination across multiple teams, which slows down development. Another useful principle from OOD is the Common Closure Principle (CCP), which states that classes that change for the same reason should be in the same package. Perhaps, for instance, two classes implement different aspects of the same business rule. The goal is that when that business rule changes developers, only need to change code in a small number - ideally only one - of packages. This kind of thinking makes sense when designing services since it will help ensure that each change should impact only one service.

3.2.1 Expectation from Decomposition Model

- The architecture must be stable
- Services must be cohesive. A service should implement a small set of strongly related functions.
- Services must conform to the Common Closure Principle - things that change together should be packaged together - to ensure that each change affect only one service
- Services must be loosely coupled - each service as an API that encapsulates its implementation. The implementation can be changed without affecting clients
- A service should be testable
- Each service be small enough to be developed by a “two pizza” team, i.e. a team of 6-10 people
- Each team that owns one or more services must be autonomous. A team must be able to develop and deploy their services with minimal collaboration with other teams.

3.2.2 Solution

Define services corresponding to Domain-Driven Design (DDD) subdomains. DDD refers to the application’s problem space - the business - as the domain. A domain is consists of multiple subdomains. Each subdomain corresponds to a different part of the business.

Subdomains can be classified as follows:

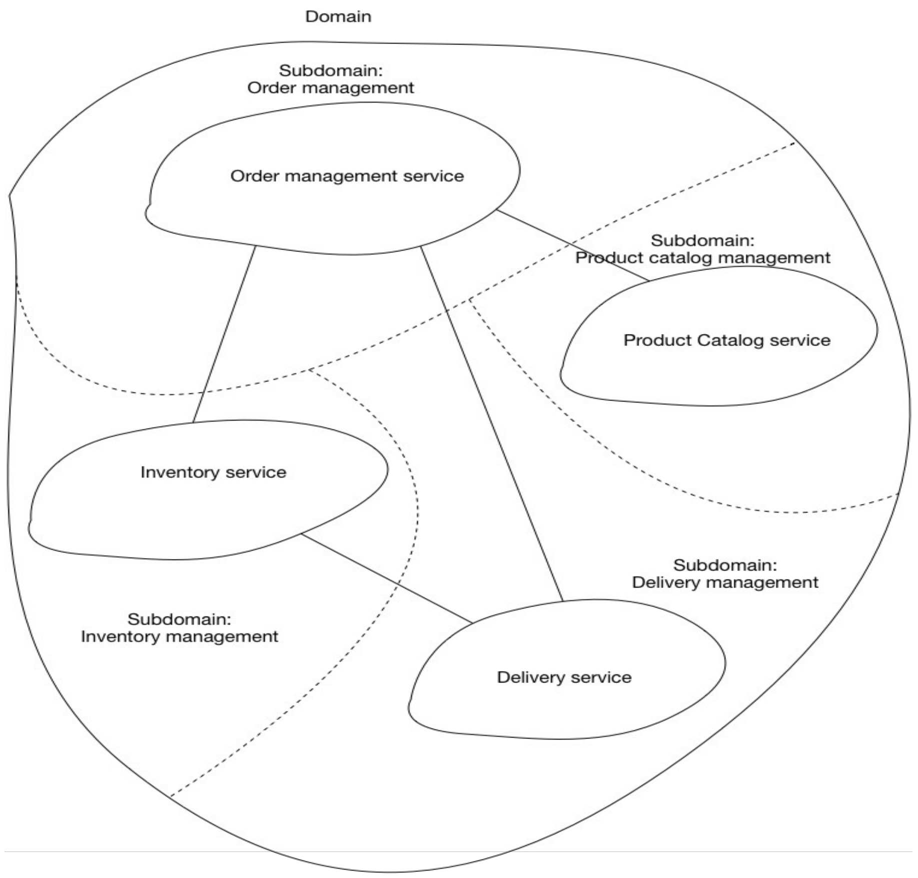
- Core - key differentiator for the business and the most valuable part of the application
- Supporting - related to what the business does but not a differentiator. These can be implemented in-house or outsourced.
- Generic - not specific to the business and are ideally implemented using off the shelf software

Examples

The subdomains of an online store include:

- Product catalog
- Inventory management
- Order management
- Delivery management
- …

The corresponding microservice architecture would have services corresponding to each of these subdomains.


Domain
Subdomain: Order management
Order management service
Subdomain: Product catalog management
Product Catalog service
Inventory service
Subdomain: Delivery management
Subdomain: Inventory management
Delivery service

3.2.3 Benefits

This pattern has the following benefits:

- Stable architecture since the subdomains are relatively stable
- Development teams are cross-functional, autonomous, and organized around delivering business value rather than technical features
- Services are cohesive and loosely coupled

3.2.4 Issues

There are the following issues to address:

- **How to identify the subdomains?** Identifying subdomains and hence services requires an understanding of the business. Like business capabilities, subdomains are identified by analyzing the business and its organizational structure and identifying the different areas of expertise. Subdomains are best identified using an iterative process. Good starting points for identifying subdomains are:
- organization structure - different groups within an organization might correspond to subdomains
- high-level domain model - subdomains often have a key domain object

3.3 Service per team

A high performance development organization consist of multiple teams. Each team is long-lived, small (typically 5-9 people), loosely coupled, autonomous, and cross-functional. Conway’s law says an architecture mirrors the communication structure of the organization that builds it. Consequently, an organization consisting of loosely coupled teams needs a loosely coupled architecture.

One such loosely coupled architecture is the microservice architecture. It’s an application style that structures an application as a loosely coupled set of services. Decompose by Subdomain and Decompose by business capability are patterns for identifying services and organizing them around business functionality. But what’s the relationship between services and teams?

One approach is a shared ownership model where multiple teams to work on each service as necessary. For example, each team might be responsible for implementing features that span multiple services. On the one hand, this approach aligns teams with the user experience. But on the other hand, it increases the amount of coordination needed between the teams. Also, the lack of code ownership increases the risk of poor code quality.

A better approach, which increases team autonomy and loose coupling, is a code/service ownership model. The team, which is responsible for a business function/capability owns a code base, which they deploy as one of more services. As a result, the team can freely develop, test, deploy and scale its services. They primarily interact with other teams in order to negotiate APIs.

A team should ideally own just one service since that’s sufficient to ensure team autonomy and loose coupling and each additional service adds complexity and overhead. A team should only deploy its code as multiple services if it solves a tangible problem, such as significantly reducing lead time or improving scalability or fault tolerance.

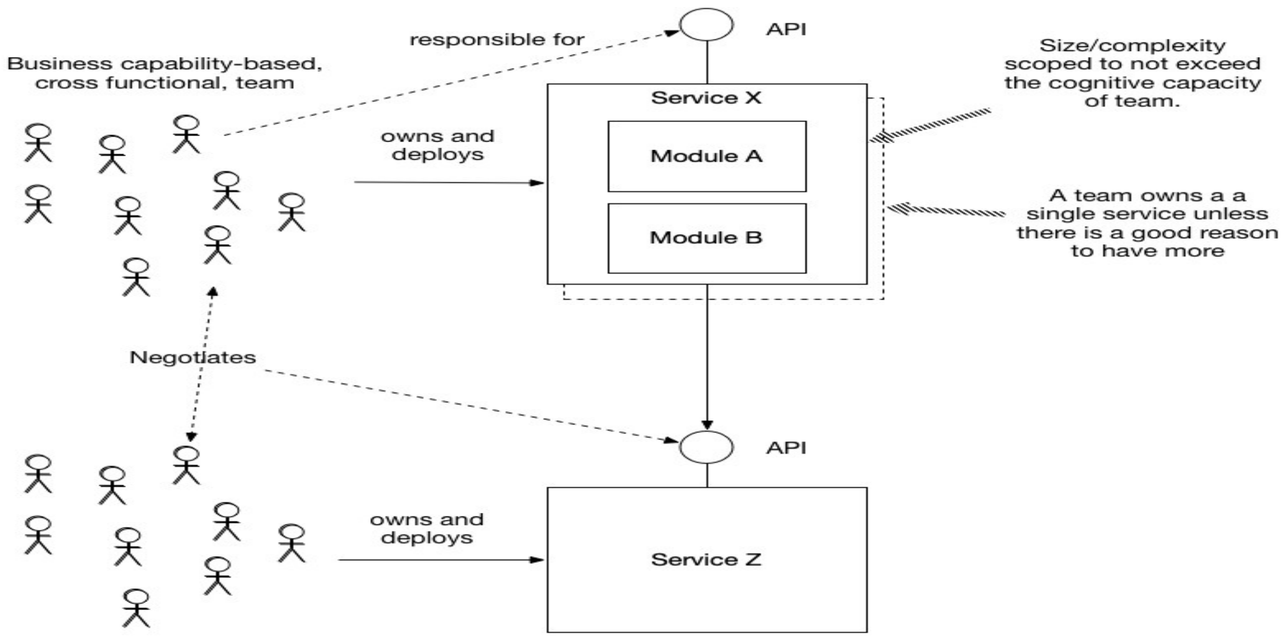
Since a team must be small, its cognitive capacity is limited. In order for the team to be productive, its code base should be scoped to not exceed the team's cognitive capacity. In other words, it must 'fit' in the team's heads. As a result, there is an upper bound on the size and/or complexity of a service.

3.3.1 Expectations from Decomposition Model

- A team should be small, e.g. 5-9 people
- A team should be autonomous and loosely coupled
- The size and complexity of the team's code base must not exceed the team's cognitive capacity.
- Finer-grained service decomposition improves -ilities including maintainability, testability, deployability
- Finer-grained service decomposition adds complexity

3.3.2 Solution

Each service is owned by a team, which has sole responsibility for making changes. Ideally each team has only one service:


Business capability-based, cross functional, team
responsible for
API
Size/complexity scoped to not exceed the cognitive capacity of team.
Service X
Module A
Module B
owns and deploys
A team owns a a single service unless there is a good reason to have more
Negotiates
API
owns and deploys
Service Z

Each team is responsible for one or more business functions (e.g. business capabilities). A team owns (has sole responsibility for changing) a code base consisting of one or more modules. Its code base is sized so as to not exceed the cognitive capacity of team. The team deploys its code as one or more services. A team should have exactly one service unless there is a proven need to have multiple services.

3.3.3 Benefits

This pattern has the following benefits:

- Enables each team to be autonomous and work with minimal coordination with other teams
- Enables the teams to be loosely coupled

- Achieves team autonomy and loose coupling with the minimum number of services
- Improves code quality due to long term code ownership

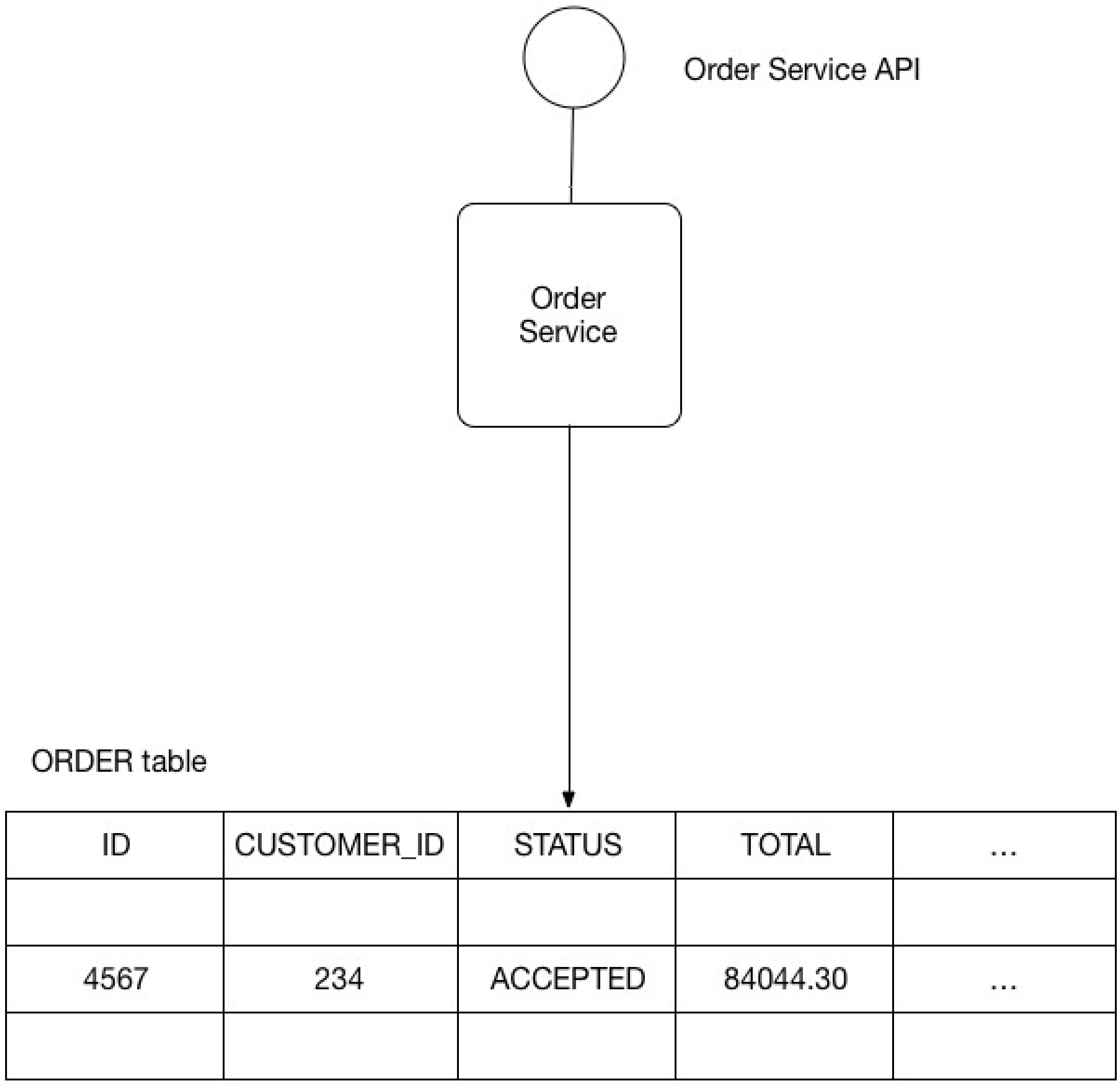
This pattern has the following drawbacks:

- Teams are not necessarily aligned with end user features
- Implementing features that span services is more complicated and requires teams to collaborate

Chapter-4
Database Models in Micro Services Architecture

4.1 Database per Service

Let’s imagine you are developing an online store application using the Microservice architecture pattern. Most services need to persist data in some kind of database. For example, the Order Service stores information about orders and the Customer Service stores information about customers.



ID	CUSTOMER_ID	STATUS	TOTAL	...
4567	234	ACCEPTED	84044.30	...

4.1.1 Expectation from the Model

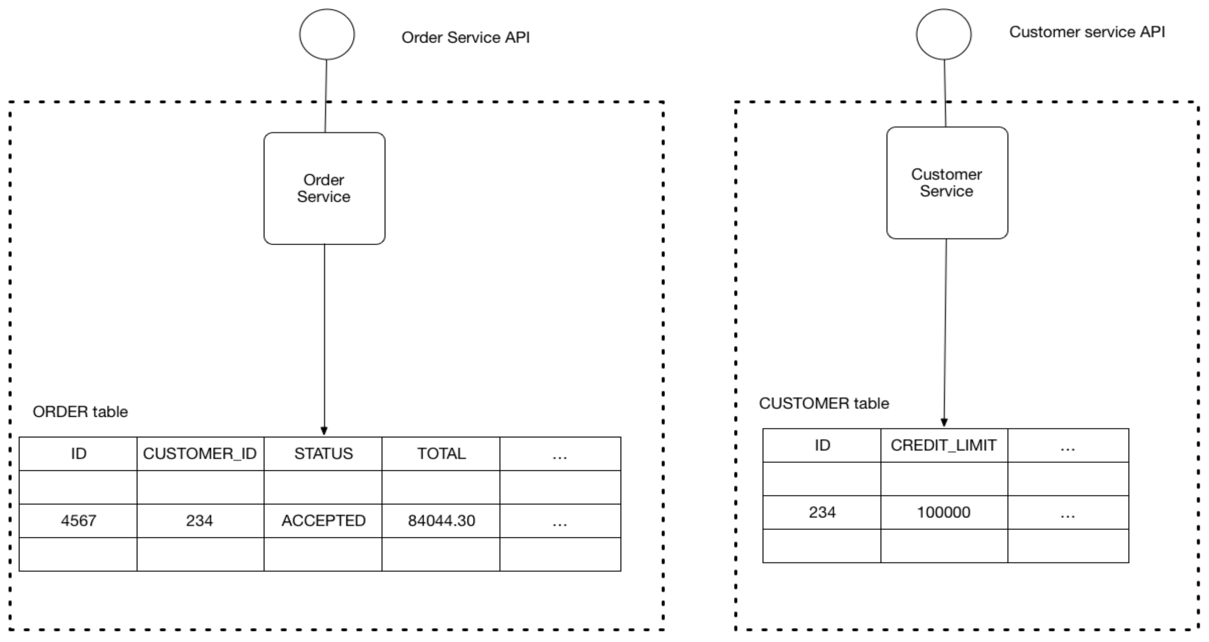
- Services must be loosely coupled so that they can be developed, deployed and scaled independently

- Some business transactions must enforce invariants that span multiple services. For example, the `Place Order` use case must verify that a new Order will not exceed the customer's credit limit. Other business transactions, must update data owned by multiple services.
- Some business transactions need to query data that is owned by multiple services. For example, the `View Available Credit` use must query the Customer to find the `creditLimit` and Orders to calculate the total amount of the open orders.
- Some queries must join data that is owned by multiple services. For example, finding customers in a particular region and their recent orders requires a join between customers and orders.
- Databases must sometimes be replicated and sharded in order to scale. See the Scale Cube.
- Different services have different data storage requirements. For some services, a relational database is the best choice. Other services might need a NoSQL database such as MongoDB, which is good at storing complex, unstructured data, or Neo4J, which is designed to efficiently store and query graph data.

4.1.2 Solution

Keep each microservice's persistent data private to that service and accessible only via its API. A service's transactions only involve its database.

The following diagram shows the structure of this pattern.



ORDER table

ID	CUSTOMER_ID	STATUS	TOTAL	...
4567	234	ACCEPTED	84044.30	...

CUSTOMER table

ID	CREDIT_LIMIT	...
234	100000	...

The service's database is effectively part of the implementation of that service. It cannot be accessed directly by other services.

There are a few different ways to keep a service's persistent data private. You do not need to provision a database server for each service. For example, if you are using a relational database then the options are:

- Private-tables-per-service – each service owns a set of tables that must only be accessed by that service
- Schema-per-service – each service has a database schema that's private to that service
- Database-server-per-service – each service has it's own database server.

Private-tables-per-service and schema-per-service have the lowest overhead. Using a schema per service is appealing since it makes ownership clearer. Some high throughput services might need their own database server.

It is a good idea to create barriers that enforce this modularity. You could, for example, assign a different database user id to each service and use a database access control mechanism such as grants. Without some kind of barrier to enforce encapsulation, developers will always be tempted to bypass a service's API and access it's data directly.

Example

The FTGO application is an example of an application that uses this approach. Each service has database credentials that only grant it access its own (logical) database on a shared MySQL server. For more information, see this blog post.

4.1.3 Resulting context

Using a database per service has the following benefits:

- Helps ensure that the services are loosely coupled. Changes to one service's database does not impact any other services.
- Each service can use the type of database that is best suited to its needs. For example, a service that does text searches could use ElasticSearch. A service that manipulates a social graph could use Neo4j.

Using a database per service has the following drawbacks:

- Implementing business transactions that span multiple services is not straightforward. Distributed transactions are best avoided because of the CAP theorem. Moreover, many modern (NoSQL) databases don't support them.
- Implementing queries that join data that is now in multiple databases is challenging.
- Complexity of managing multiple SQL and NoSQL databases

There are various patterns/solutions for implementing transactions and queries that span services:

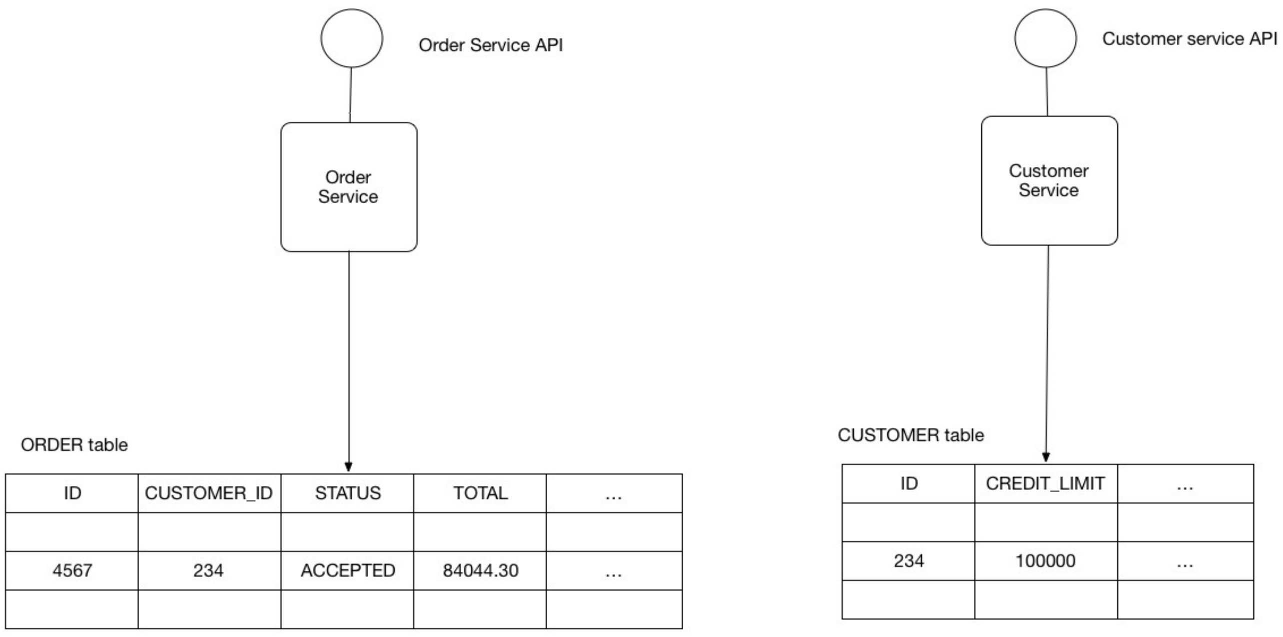
- Implementing transactions that span services - use the Saga pattern.
- Implementing queries that span services:
    - API Composition - the application performs the join rather than the database. For example, a service (or the API gateway) could retrieve a customer and their orders by

first retrieving the customer from the customer service and then querying the order service to return the customer's most recent orders.

- Command Query Responsibility Segregation (CQRS) - maintain one or more materialized views that contain data from multiple services. The views are kept by services that subscribe to events that each services publishes when it updates its data. For example, the online store could implement a query that finds customers in a particular region and their recent orders by maintaining a view that joins customers and orders. The view is updated by a service that subscribes to customer and order events.

4.2 Shared database

Let's imagine you are developing an online store application using the Microservice architecture pattern. Most services need to persist data in some kind of database. For example, the Order Service stores information about orders and the Customer Service stores information about customers.



ORDER table

ID	CUSTOMER_ID	STATUS	TOTAL	...
4567	234	ACCEPTED	84044.30	...

CUSTOMER table

ID	CREDIT_LIMIT	...
234	100000	...

4.2.1 Expectation from the Model

- Services must be loosely coupled so that they can be developed, deployed and scaled independently
- Some business transactions must enforce invariants that span multiple services. For example, the Place Order use case must verify that a new Order will not exceed the customer's credit limit. Other business transactions, must update data owned by multiple services.

- Some business transactions need to query data that is owned by multiple services. For example, the View Available Credit use must query the Customer to find the creditLimit and Orders to calculate the total amount of the open orders.
- Some queries must join data that is owned by multiple services. For example, finding customers in a particular region and their recent orders requires a join between customers and orders.
- Databases must sometimes be replicated and sharded in order to scale. See the Scale Cube.
- Different services have different data storage requirements. For some services, a relational database is the best choice. Other services might need a NoSQL database such as MongoDB, which is good at storing complex, unstructured data, or Neo4J, which is designed to efficiently store and query graph data.

4.2.2 Solution

Use a (single) database that is shared by multiple services. Each service freely accesses data owned by other services using local ACID transactions.

Example

The OrderService and CustomerService freely access each other’s tables. For example, the OrderService can use the following ACID transaction ensure that a new order will not violate the customer’s credit limit.

```
BEGIN TRANSACTION

…

SELECT ORDER_TOTAL

  FROM ORDERS WHERE CUSTOMER_ID = ?

…

SELECT CREDIT_LIMIT

FROM CUSTOMERS WHERE CUSTOMER_ID = ?

…

INSERT INTO ORDERS …

…

COMMIT TRANSACTION
```

The database will guarantee that the credit limit will not be exceeded even when simultaneous transactions attempt to create orders for the same customer.

4.2.4 Benefits

The benefits of this pattern are:

- A developer uses familiar and straightforward ACID transactions to enforce data consistency
- A single database is simpler to operate

The drawbacks of this pattern are:

- Development time coupling - a developer working on, for example, the OrderService will need to coordinate schema changes with the developers of other services that access the same tables. This coupling and additional coordination will slow down development.
- Runtime coupling - because all services access the same database they can potentially interfere with one another. For example, if long running CustomerService transaction holds a lock on the ORDER table then the OrderService will be blocked.
- Single database might not satisfy the data storage and access requirements of all services.

4.3 SAGA (Maintaining Data Consistency in MSA)

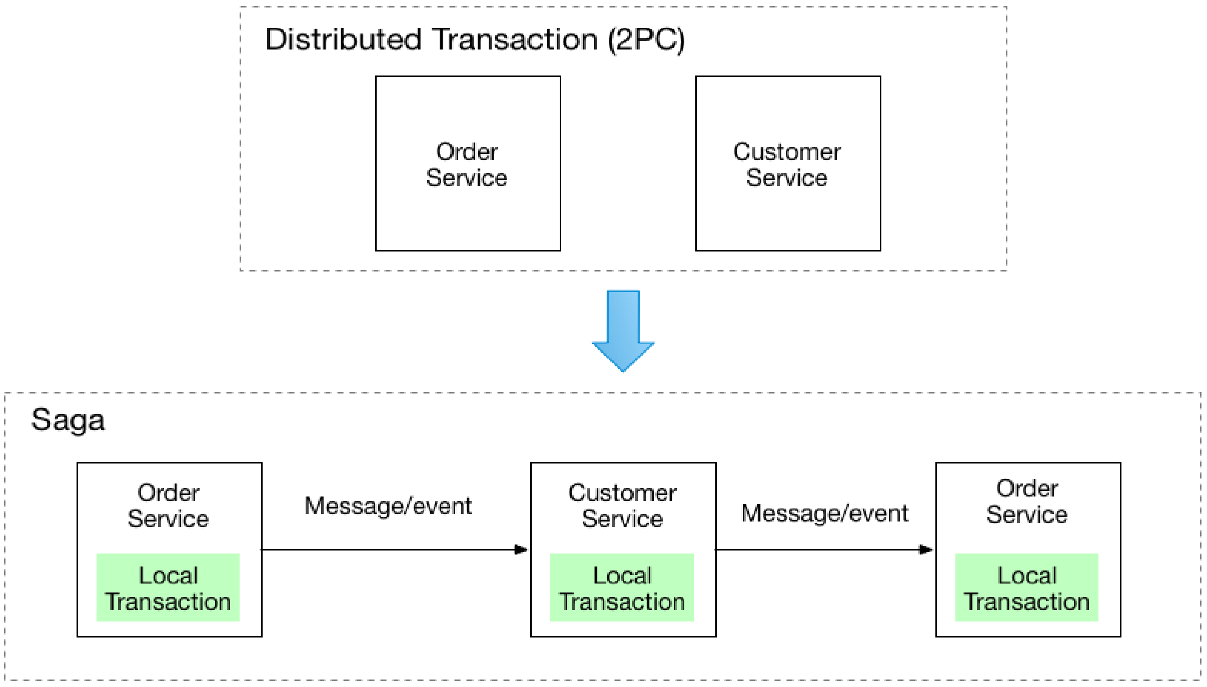
You have applied the Database per Service pattern. Each service has its own database. Some business transactions, however, span multiple service so you need a mechanism to ensure data consistency across services. For example, lets imagine that you are building an e-commerce store where customers have a credit limit. The application must ensure that a new order will not exceed the customer’s credit limit. Since Orders and Customers are in different databases the application cannot simply use a local ACID transaction.

4.3.1 Problem

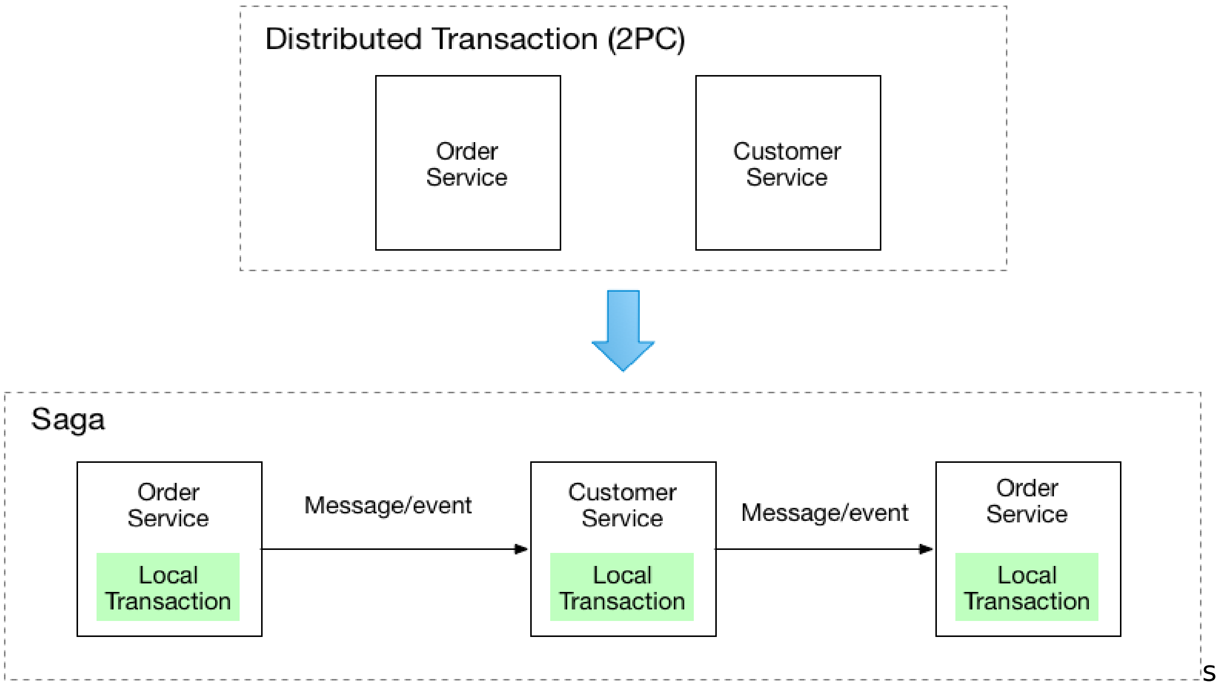
How to maintain data consistency across services when conventional ACID transaction services will not work?

4.3.2 Solution

Implement each business transaction that spans multiple services as a saga. A saga is a sequence of local transactions. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga. If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions.



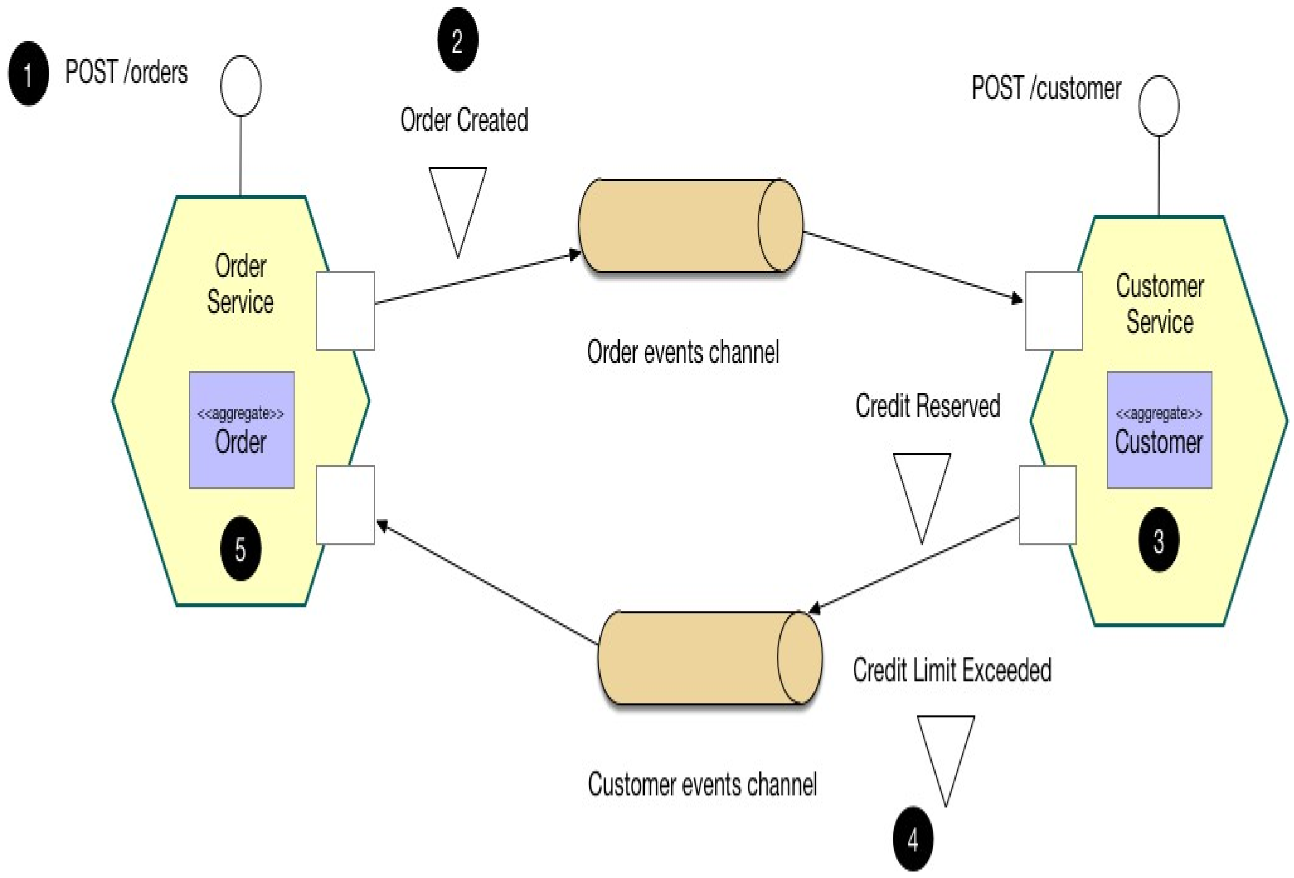
There are two ways of coordination saga



s:

- •Choreography - each local transaction publishes domain events that trigger local transactions in other services
- •Orchestration - an orchestrator (object) tells the participants what local transactions to execute

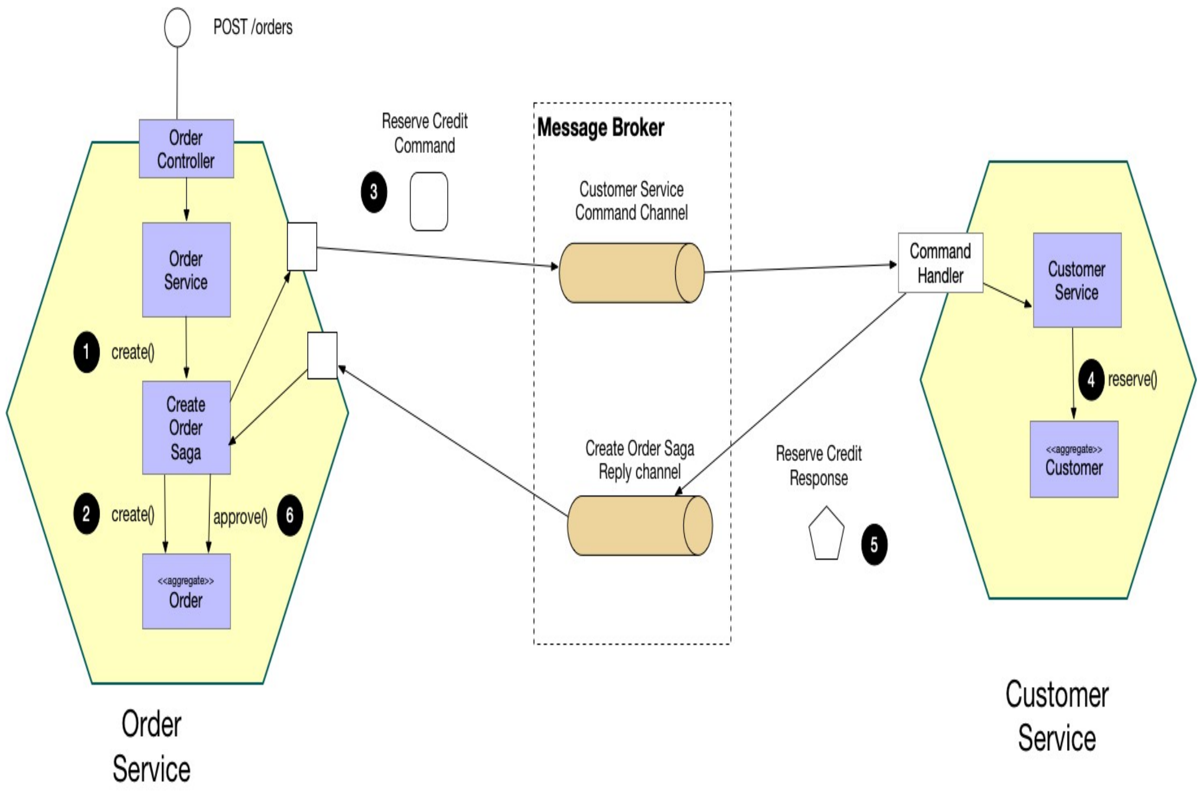
4.3.3 Choreography-based saga



An e-commerce application that uses this approach would create an order using a choreography-based saga that consists of the following steps:

1. The Order Service receives the POST /orders request and creates an Order in a PENDING state
2. It then emits an Order Created event
3. The Customer Service's event handler attempts to reserve credit
4. It then emits an event indicating the outcome
5. The OrderService's event handler either approves or rejects the Order

4.3.4 Orchestration-based saga



An e-commerce application that uses this approach would create an order using an orchestration-based saga that consists of the following steps:

1.The Order Service receives the POST /orders request and creates the Create Order saga orchestrator

2.The saga orchestrator creates an Order in the PENDING state

3.It then sends a Reserve Credit command to the Customer Service

4.The Customer Service attempts to reserve credit

5.It then sends back a reply message indicating the outcome

6.The saga orchestrator either approves or rejects the Order

Resulting context

This pattern has the following benefits:

- It enables an application to maintain data consistency across multiple services without using distributed transactions

This solution has the following drawbacks:

- The programming model is more complex. For example, a developer must design compensating transactions that explicitly undo changes made earlier in a saga.

There are also the following issues to address:

- In order to be reliable, a service must atomically update its database and publish a message/event. It cannot use the traditional mechanism of a distributed transaction that spans the database and the message broker. Instead, it must use one of the patterns listed below.

4.4 Command Query Responsibility Segregation (CQRS)

4.4.1 Context

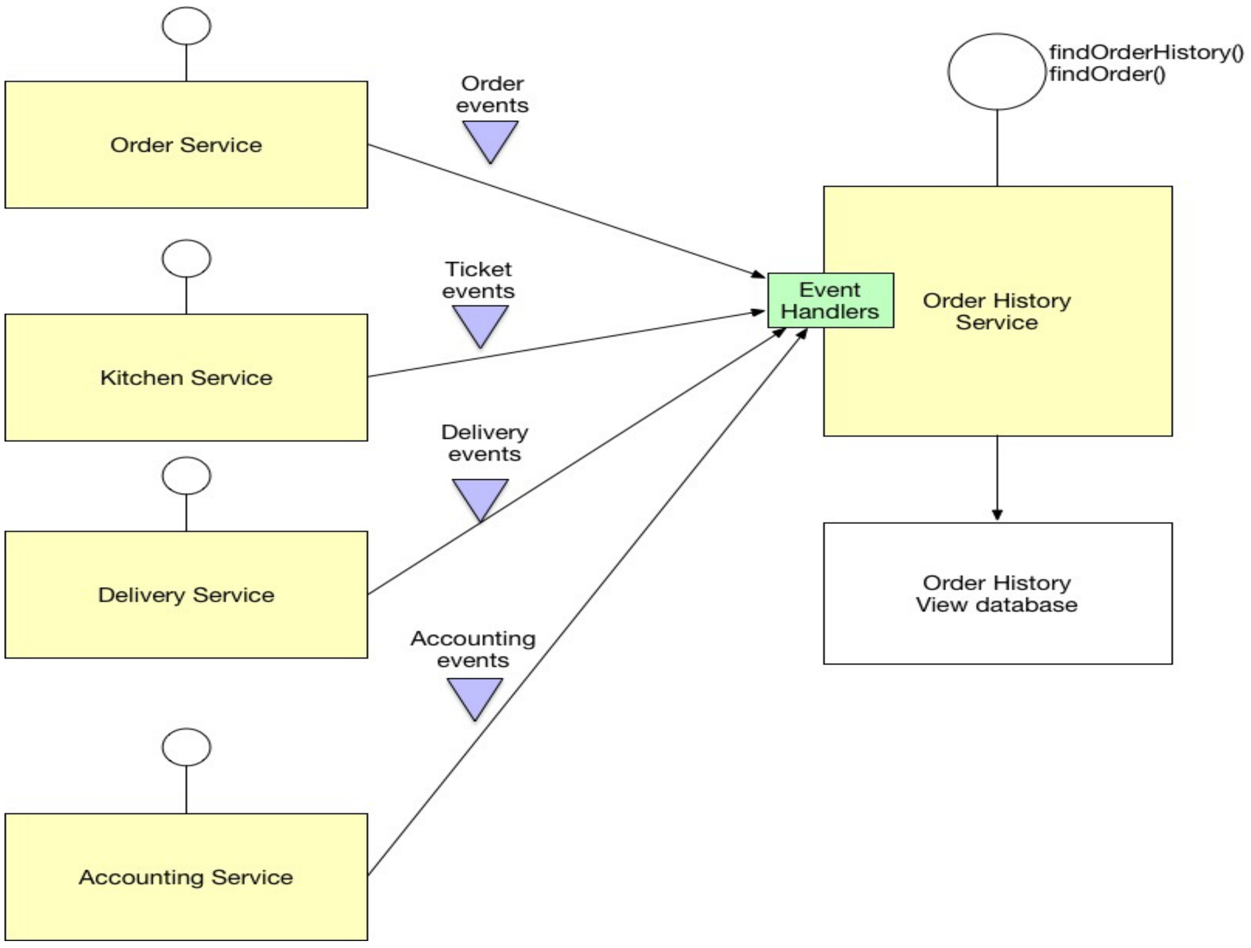
You have applied the Microservices architecture pattern and the Database per service pattern. As a result, it is no longer straightforward to implement queries that join data from multiple services. Also, if you have applied the Event sourcing pattern then the data is no longer easily queried.

4.4.2 Problem

How to implement a query that retrieves data from multiple services in a microservice architecture?

4.4.3 Solution

Define a view database, which is a read-only replica that is designed to support that query. The application keeps the replica up to data by subscribing to Domain events published by the service that own the data.



Examples

- My book’s FTGO example application has the Order History Service, which implements this pattern.
- There are several Eventuate-based example applications that illustrate how to use this pattern.

4.4.4 Benefits

This pattern has the following benefits:

- Supports multiple denormalized views that are scalable and performant
- Improved separation of concerns = simpler command and query models
- Necessary in an event sourced architecture

This pattern has the following drawbacks:

- Increased complexity
- Potential code duplication
- Replication lag/eventually consistent views

4.5 Transactional outbox

4.5.1 Context

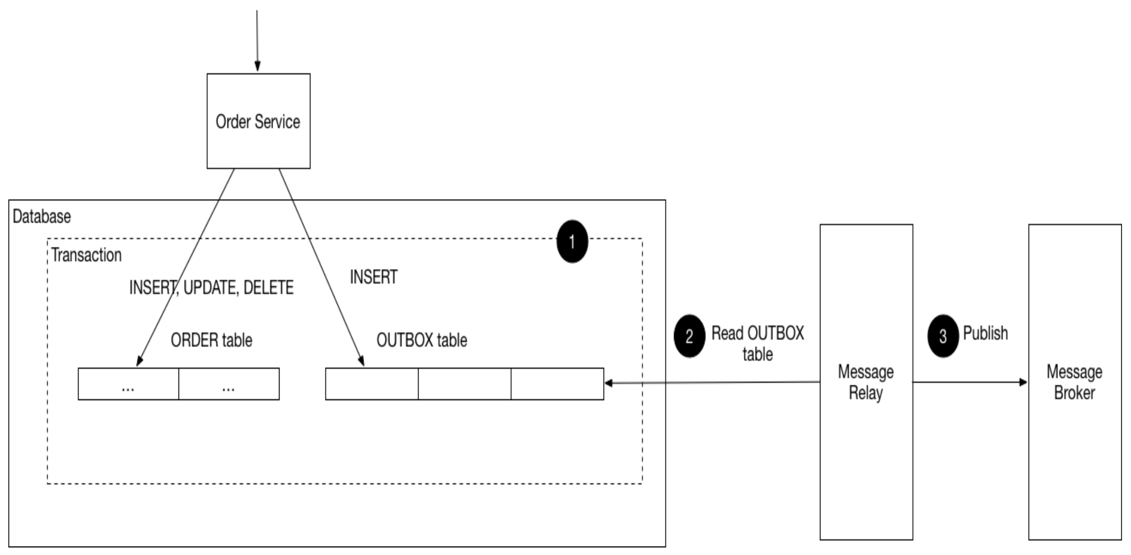
A service command typically needs to update the database **and** send messages/events. For example, a service that participates in a saga needs to atomically update the database and sends messages/events. Similarly, a service that publishes a domain event must atomically update an aggregate and publish an event. The database update and sending of the message must be atomic in order to avoid data inconsistencies and bugs. However, it is not viable to use a distributed transaction that spans the database and the message broker to atomically update the database and publish messages/events.

4.5.2 Problem

How to reliably/atomically update the database and publish messages/events?

4.5.3 Solution

A service that uses a relational database inserts messages/events into an outbox table (e.g. MESSAGE) as part of the local transaction. An service that uses a NoSQL database appends the messages/events to attribute of the record (e.g. document or item) being updated. A separate Message Relay process publishes the events inserted into database to a message broker.



4.5.4 Benefits

This pattern has the following benefits:

- The service publishes high-level domain events
- No 2PC required

This pattern has the following drawbacks:

- Potentially error prone since the developer might forget to publish the message/event after updating the database.

This pattern also has the following issues:

- The Message Relay might publish a message more than once. It might, for example, crash after publishing a message but before recording the fact that it has done so. When it restarts, it will then publish the message again. As a result, a message consumer must be idempotent, perhaps by tracking the IDs of the messages that it has already processed. Fortunately, since Message Consumers usually need to be idempotent (because a message broker can deliver messages more than once) this is typically not a problem.

Related patterns

- The Saga and Domain event patterns create the need for this pattern.
- The Event sourcing is an alternative solution
- There are two patterns for implementing the message relay:
    - The Transaction log tailing pattern
    - The Polling publisher pattern

Chapter 5
Micro Services Packaging and Deployment

5.1 Multiple service instances per host

You have applied the Microservice architecture pattern and architected your system as a set of services. Each service is deployed as a set of service instances for throughput and availability.

5.1.1 Expectations from the Packaging and Deployment Scenario

- •Services are written using a variety of languages, frameworks, and framework versions
- •Each service consists of multiple service instances for throughput and availability
- •Service must be independently deployable and scalable
- •Service instances need to be isolated from one another
- •You need to be able to quickly build and deploy a service
- •You need to be able to constrain the resources (CPU and memory) consumed by a service
- •You need to monitor the behavior of each service instance
- •You want deployment to reliable
- •You must deploy the application as cost-effectively as possible

5.1.2 Solution

Run multiple instances of different services on a host (Physical or Virtual machine).

There are various ways of deploying a service instance on a shared host including:

- •Deploy each service instance as a JVM process. For example, a Tomcat or Jetty instances per service instance.
- •Deploy multiple service instances in the same JVM. For example, as web applications or OSGI bundles.

5.1.3 Benefits

The benefits of this pattern include:

- •More efficient resource utilization than the Service Instance per host pattern

The drawbacks of this approach include:

- Risk of conflicting resource requirements
- Risk of conflicting dependency versions
- Difficult to limit the resources consumed by a service instance
- If multiple services instances are deployed in the same process then its difficult to monitor the resource consumption of each service instance. Its also impossible to isolate each instance

5.2 Single Service Instance per Host

You have applied the Microservice architecture pattern and architected your system as a set of services. Each service is deployed as a set of service instances for throughput and availability.

5.2.1 Expectations

- Services are written using a variety of languages, frameworks, and framework versions
- Each service consists of multiple service instances for throughput and availability
- Service must be independently deployable and scalable
- Service instances need to be isolated from one another
- You need to be able to quickly build and deploy a service
- You need to be able to constrain the resources (CPU and memory) consumed by a service
- You need to monitor the behavior of each service instance
- You want deployment to reliable
- You must deploy the application as cost-effectively as possible

5.2.2 Solution

Deploy each single service instance on its own host

5.2.3 Benefits

The benefits of this approach include:

- Services instances are isolated from one another
- There is no possibility of conflicting resource requirements or dependency versions
- A service instance can only consume at most the resources of a single host
- Its straightforward to monitor, manage, and redeploy each service instance

5.2.4 Drawbacks of this approach include:

- Potentially less efficient resource utilization compared to Multiple Services per Host because there are more hosts

5.3 Service Instance per VM

You have applied the Microservice architecture pattern and architected your system as a set of services. Each service is deployed as a set of service instances for throughput and availability.

5.3.1 Expectations

- Services are written using a variety of languages, frameworks, and framework versions
- Each service consists of multiple service instances for throughput and availability
- Service must be independently deployable and scalable
- Service instances need to be isolated from one another
- You need to be able to quickly build and deploy a service
- You need to be able to constrain the resources (CPU and memory) consumed by a service
- You need to monitor the behavior of each service instance
- You want deployment to reliable
- You must deploy the application as cost-effectively as possible

5.3.2 Solution

Package the service as a virtual machine image and deploy each service instance as a separate VM

Examples

- Netflix packages each service as an EC2 AMI and deploys each service instance as a EC2 instance.

5.3.3 Benefits and Drawbacks

The benefits of this approach include:

- Its straightforward to scale the service by increasing the number of instances. Amazon Autoscaling Groups can even do this automatically based on load.
- The VM encapsulates the details of the technology used to build the service. All services are, for example, started and stopped in exactly the same way.
- Each service instance is isolated
- A VM imposes limits on the CPU and memory consumed by a service instance
- IaaS solutions such as AWS provide a mature and feature rich infrastructure for deploying and managing virtual machines. For example,
    - Elastic Load Balancer -
    - Autoscaling groups
    - …

The drawbacks of this approach include:

- Building a VM image is slow and time consuming

5.4 Service instance per container

You have applied the Microservice architecture pattern and architected your system as a set of services. Each service is deployed as a set of service instances for throughput and availability.

5.4.1 Expectation from Deployment

- Services are written using a variety of languages, frameworks, and framework versions
- Each service consists of multiple service instances for throughput and availability
- Service must be independently deployable and scalable
- Service instances need to be isolated from one another
- You need to be able to quickly build and deploy a service
- You need to be able to constrain the resources (CPU and memory) consumed by a service
- You need to monitor the behavior of each service instance
- You want deployment to reliable
- You must deploy the application as cost-effectively as possible

Solution

Package the service as a (Docker) container image and deploy each service instance as a container

Examples

Docker is becoming an extremely popular way of packaging and deploying services. Each service is packaged as a Docker image and each service instance is a Docker container. There are several Docker clustering frameworks including:

- Kubernetes
- Marathon/Mesos
- Amazon EC2 Container Service

5.4.2 Benefits and Drwbacks

The benefits of this approach include:

- It is straightforward to scale up and down a service by changing the number of container instances.

- The container encapsulates the details of the technology used to build the service. All services are, for example, started and stopped in exactly the same way.
- Each service instance is isolated
- A container imposes limits on the CPU and memory consumed by a service instance
- Containers are extremely fast to build and start. For example, it’s 100x faster to package an application as a Docker container than it is to package it as an AMI. Docker containers also start much faster than a VM since only the application process starts rather than an entire OS.

The drawbacks of this approach include:

- The infrastructure for deploying containers is not as rich as the infrastructure for deploying virtual machines.

5.4 Serverless deployment

You have applied the Microservice architecture pattern and architected your system as a set of services. Each service is deployed as a set of service instances for throughput and availability.

5.4.1 Expectations

- Services are written using a variety of languages, frameworks, and framework versions
- Each service consists of multiple service instances for throughput and availability
- Service must be independently deployable and scalable
- Service instances need to be isolated from one another
- You need to be able to quickly build and deploy a service
- You need to be able to constrain the resources (CPU and memory) consumed by a service
- You need to monitor the behavior of each service instance
- You want deployment to reliable
- You must deploy the application as cost-effectively as possible

Solution

Use a deployment infrastructure that hides any concept of servers (i.e. reserved or preallocated resources)- physical or virtual hosts, or containers. The infrastructure takes your service’s code and runs it. You are charged for each request based on the resources consumed.

To deploy your service using this approach, you package the code (e.g. as a ZIP file), upload it to the deployment infrastructure and describe the desired performance characteristics.

The deployment infrastructure is a utility operated by a public cloud provider. It typically uses either containers or virtual machines to isolate the services. However, these details are hidden from you. Neither you nor anyone else in your organization is responsible for managing any low-level infrastructure such as operating systems, virtual machines, etc.

Examples

There are a few different serverless deployment environments:

- AWS Lambda
- Google Cloud Functions
- Azure Functions

They offer similar functionality but AWS Lambda has the richest feature set. An AWS Lambda function is a stateless component that is invoked to handle events. To create an AWS Lambda function you package your the NodeJS, Java or Python code for your service in a ZIP file, and upload it to AWS Lambda. You also specify the name of function that handles events as well as resource limits.

When an event occurs, AWS Lambda finds an idle instance of your function, launching one if none are available and invokes the handler function. AWS Lambda runs enough instances of your function to handle the load. Under the covers, it uses containers to isolate each instance of a lambda function. As you might expect, AWS Lambda runs the containers on EC2 instances.

There are four ways to invoke a lambda function. One option is to configure your lambda function to be invoked in response to an event generated by an AWS service such as S3, DynamoDB or Kinesis. Examples of events include the following:

- an object being created in a S3 bucket
- an item is created, updated or deleted in a DynamoDB table
- a message is available to read from a Kinesis stream
- an email being received via the Simple email service.

Another way to invoke a lambda function is to configure the AWS Lambda Gateway to route HTTP requests to your lambda. AWS Gateway transforms an HTTP request into an event object, invokes the lambda function, and generates a HTTP response from the lambda function’s result.

You can also explicitly invoke your lambda function using the AWS Lambda Web Service API. Your application that invokes the lambda function supplies a JSON object, which is passed to the lambda function. The web service call returns the value returned by the lambda.

The fourth way to invoke a lambda function is periodically using a cron-like mechanism. You can, for example, tell AWS to invoke you lambda function every five minutes.

The cost of each invocation is a function of the duration of the invocation, which is measured in 100 millisecond increments, and the memory consumed.

5.4.2 Benefits and Drawbacks

The benefits of using serverless deployment include:

- It eliminates the need to spend time on the undifferentiated heavy lifting of managing low-level infrastructure. Instead, you can focus on your code.
- The serverless deployment infrastructure is extremely elastic. It automatically scales your services to handle the load.
- You pay for each request rather than provisioning what might be under utilized virtual machines or containers.

The drawbacks of serverless deployment include:

- Significant limitation and constraints - A serverless deployment environment typically has far more constraints that a VM-based or Container-based infrastructure. For example, AWS Lambda only supports a few languages. It is only suitable for deploying stateless applications that run in response to a request. You cannot deploy a long running stateful application such as a database or message broker.
- Limited “input sources” - lambdas can only respond to requests from a limited set of input sources. AWS Lambda is not intended to run services that, for example, subscribe to a message broker such as RabbitMQ.
- Applications must startup quickly - serverless deployment is not a good fit your service takes a long time to start
- Risk of high latency - the time it takes for the infrastructure to provision an instance of your function and for the function to initialize might result in significant latency. Moreover, a serverless deployment infrastructure can only react to increases in load. You cannot proactively pre-provision capacity. As a result, your application might initially exhibit high latency when there are sudden, massive spikes in load.

The deployment infrastructure will internally deploy your application using one of the other patterns. It will most likely use Service Service per Host pattern.

Chapter 6

Accessing Micro Services

6.1 API Gateway / Backends for Frontends

Let’s imagine you are building an online store that uses the Microservice architecture pattern and that you are implementing the product details page. You need to develop multiple versions of the product details user interface:

- HTML5/JavaScript-based UI for desktop and mobile browsers - HTML is generated by a server-side web application
- Native Android and iPhone clients - these clients interact with the server via REST APIs

In addition, the online store must expose product details via a REST API for use by 3rd party applications.

A product details UI can display a lot of information about a product. For example, the Amazon.com details page for POJOs in Action displays:

- Basic information about the book such as title, author, price, etc.
- Your purchase history for the book
- Availability
- Buying options
- Other items that are frequently bought with this book
- Other items bought by customers who bought this book
- Customer reviews
- Sellers ranking
- …

Since the online store uses the Microservice architecture pattern the product details data is spread over multiple services. For example,

- Product Info Service - basic information about the product such as title, author
- Pricing Service - product price
- Order service - purchase history for product
- Inventory service - product availability
- Review service - customer reviews …

Consequently, the code that displays the product details needs to fetch information from all of these services.

6.1.2 Problem

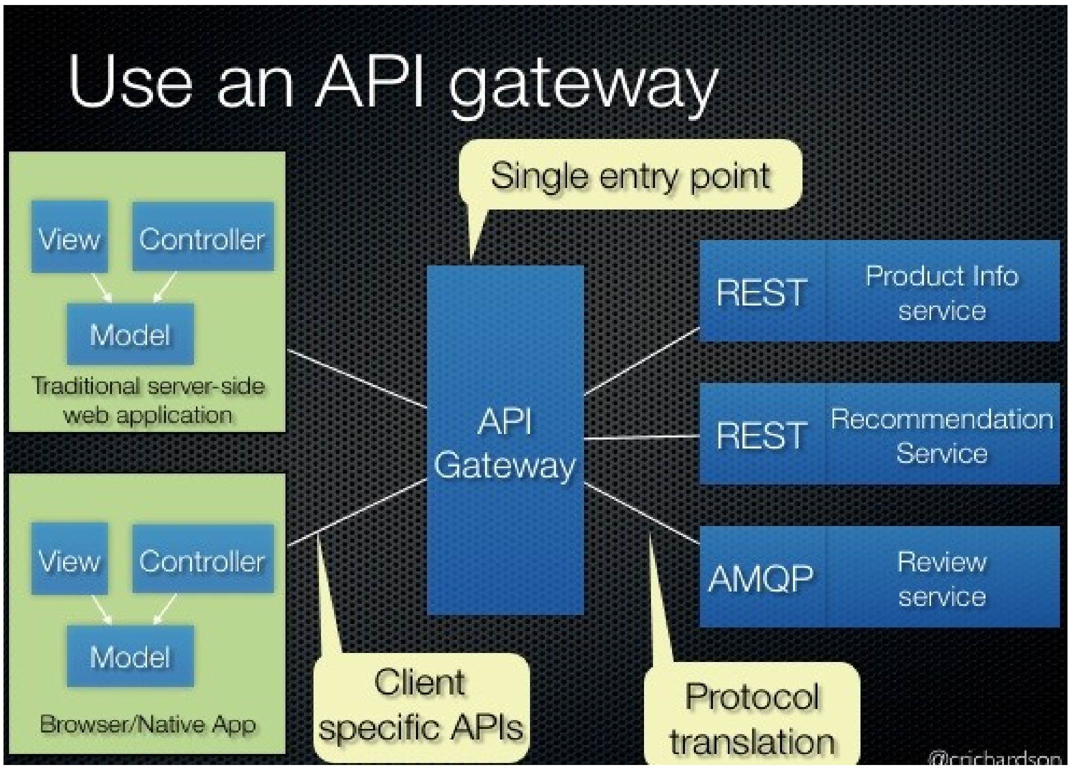
How do the clients of a Microservices-based application access the individual services?

Forces

- The granularity of APIs provided by microservices is often different than what a client needs. Microservices typically provide fine-grained APIs, which means that clients need to interact with multiple services. For example, as described above, a client needing the details for a product needs to fetch data from numerous services.
- Different clients need different data. For example, the desktop browser version of a product details page desktop is typically more elaborate then the mobile version.
- Network performance is different for different types of clients. For example, a mobile network is typically much slower and has much higher latency than a non-mobile network. And, of course, any WAN is much slower than a LAN. This means that a native mobile client uses a network that has very difference performance characteristics than a LAN used by a server-side web application. The server-side web application can make multiple requests to backend services without impacting the user experience where as a mobile client can only make a few.
- The number of service instances and their locations (host+port) changes dynamically
- Partitioning into services can change over time and should be hidden from clients
- Services might use a diverse set of protocols, some of which might not be web friendly

6.1.3 Solution

Implement an API gateway that is the single entry point for all clients. The API gateway handles requests in one of two ways. Some requests are simply proxied/routed to the appropriate service. It handles other requests by fanning out to multiple services.

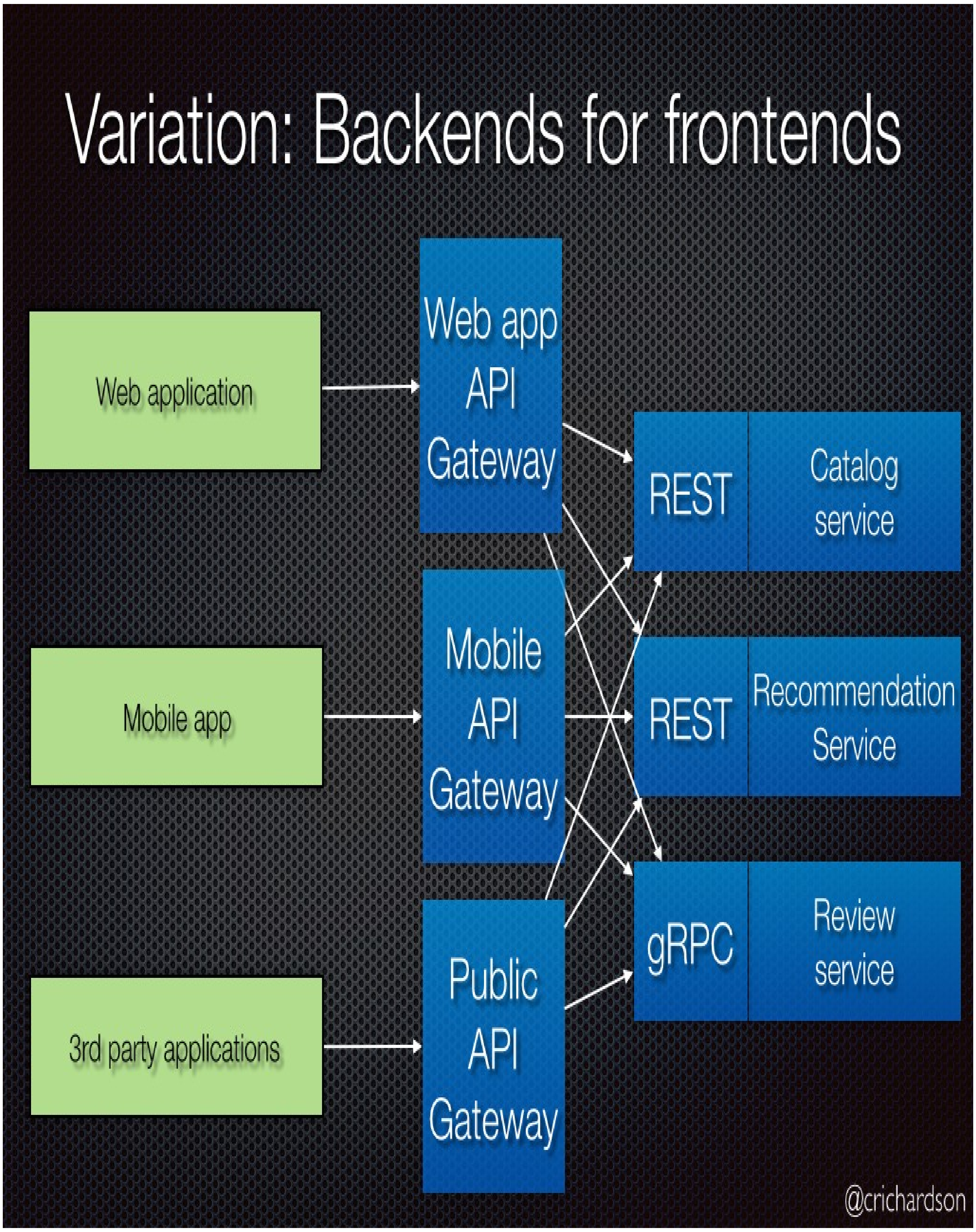


Rather than provide a one-size-fits-all style API, the API gateway can expose a different API for each client. For example, the Netflix API gateway runs client-specific adapter code that provides each client with an API that's best suited to its requirements.

The API gateway might also implement security, e.g. verify that the client is authorized to perform the request

6.1.4 Variation: Backends for frontends

A variation of this pattern is the Backends for frontends pattern. It defines a separate API gateway for each kind of client.



In this example, there are three kinds of clients: web application, mobile application, and external 3rd party application. There are three different API gateways. Each one is provides an API for its client.

Examples

- Netflix API gateway
- A simple Java/Spring API gateway from the Money Transfer example application.

6.1.5 Benefits and Drawbacks

Using an API gateway has the following benefits:

- Insulates the clients from how the application is partitioned into microservices
- Insulates the clients from the problem of determining the locations of service instances
- Provides the optimal API for each client
- Reduces the number of requests/roundtrips. For example, the API gateway enables clients to retrieve data from multiple services with a single round-trip. Fewer requests also means less overhead and improves the user experience. An API gateway is essential for mobile applications.
- Simplifies the client by moving logic for calling multiple services from the client to API gateway
- Translates from a “standard” public web-friendly API protocol to whatever protocols are used internally

The API gateway pattern has some drawbacks:

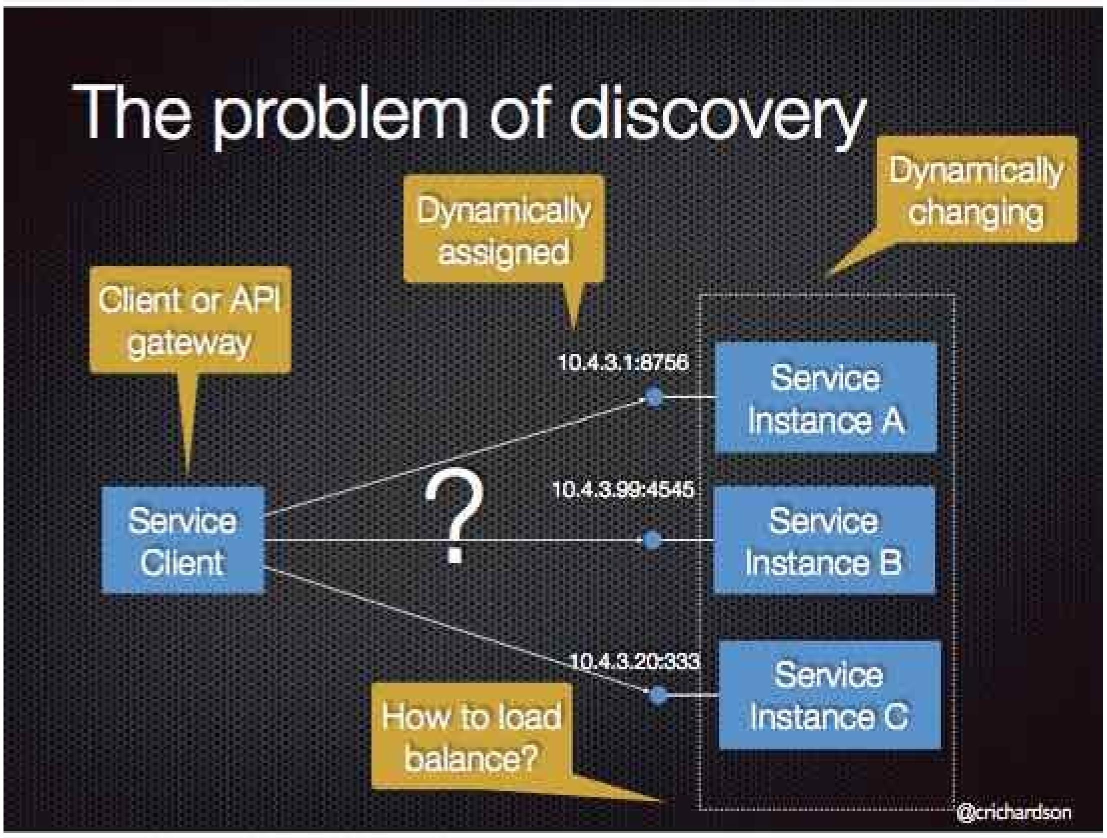
- Increased complexity - the API gateway is yet another moving part that must be developed, deployed and managed
- Increased response time due to the additional network hop through the API gateway - however, for most applications the cost of an extra roundtrip is insignificant.

Issues:

- How implement the API gateway? An event-driven/reactive approach is best if it must scale to scale to handle high loads. On the JVM, NIO-based libraries such as Netty, Spring Reactor, etc. make sense. NodeJS is another option.

6.2 Client-side service discovery

Services typically need to call one another. In a monolithic application, services invoke one another through language-level method or procedure calls. In a traditional distributed system deployment, services run at fixed, well known locations (hosts and ports) and so can easily call one another using HTTP/REST or some RPC mechanism. However, a modern microservice-based application typically runs in a virtualized or containerized environments where the number of instances of a service and their locations changes dynamically.



Consequently, you must implement a mechanism for that enables the clients of service to make requests to a dynamically changing set of ephemeral service instances.

6.2.1 Problem

How does the client of a service - the API gateway or another service - discover the location of a service instance?

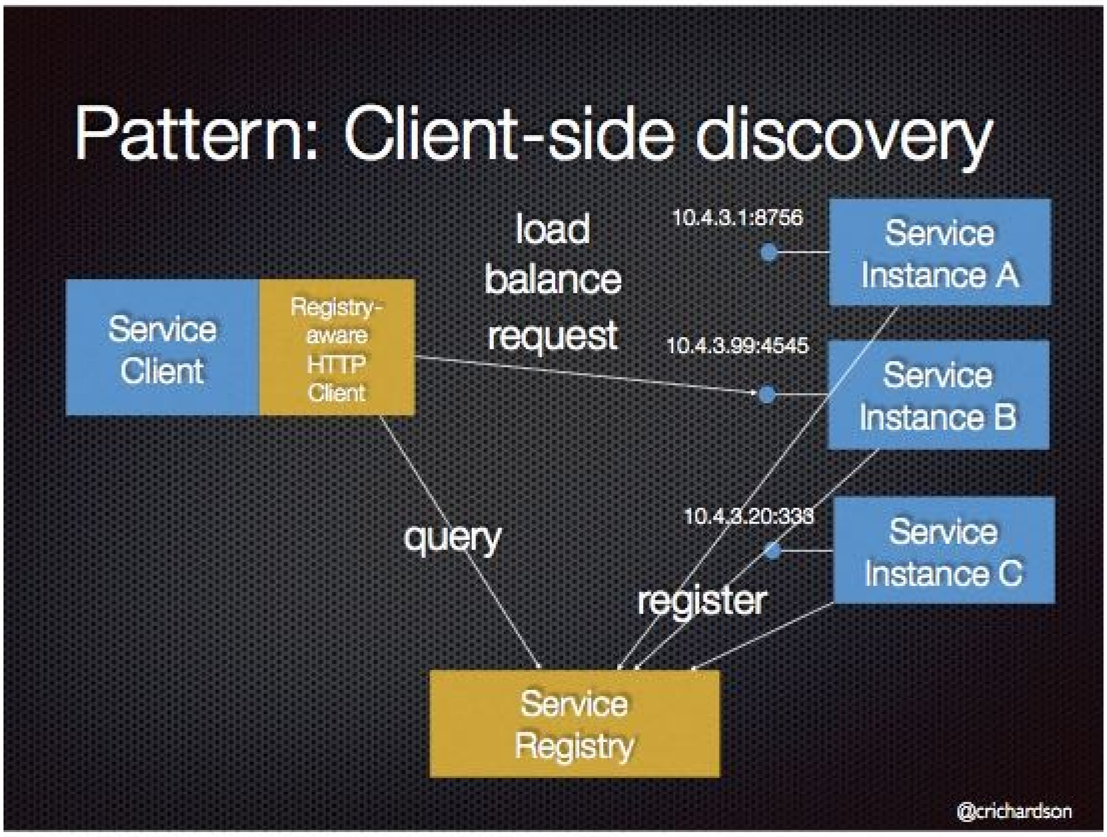
Forces

- •Each instance of a service exposes a remote API such as HTTP/REST, or Thrift etc. at a particular location (host and port)
- •The number of services instances and their locations changes dynamically.
- •Virtual machines and containers are usually assigned dynamic IP addresses.
- •The number of services instances might vary dynamically. For example, an EC2 Autoscaling Group adjusts the number of instances based on load.

6.2.3 Solution

When making a request to a service, the client obtains the location of a service instance by querying a Service Registry, which knows the locations of all service instances.

The following diagram shows the structure of this pattern.



This is typically handled by a Microservice chassis framework

Examples

The Microservices Example application is an example of an application that uses client-side service discovery. It is written in Scala and uses Spring Boot and Spring Cloud as the Microservice chassis. They provide various capabilities including client-side discovery.

6.4 3rd Party Registration

You have applied either the Client-side Service Discovery pattern or the Server-side Service Discovery pattern. Service instances must be registered with the service registry on startup so that they can be discovered and unregistered on shutdown.

6.4.1 Problem

How are service instances registered with and unregistered from the service registry?

Forces

- Service instances must be registered with the service registry on startup and unregistered on shutdown
- Service instances that crash must be unregistered from the service registry
- Service instances that are running but incapable of handling requests must be unregistered from the service registry

6.4.2 Solution

A 3rd party registrar is responsible for registering and unregistering a service instance with the service registry. When the service instance starts up, the registrar registers the service instance with the service registry. When the service instance shuts downs, the registrar unregisters the service instance from the service registry.

Examples

Examples of the 3rd Party Registration pattern include:

- Netflix Prana - a “side car” application that runs along side a non-JVM application and registers the application with Eureka.
- AWS Autoscaling Groups automatically (un)registers EC2 instances with Elastic Load Balancer
- Joyent’s Container buddy runs in a Docker container as the parent process for the service and registers it with the registry
- Registrator - registers and unregisters Docker containers with various service registries
- Clustering frameworks such as Kubernetes and Marathon (un)register service instances with the built-in/implicit registry

Resulting context

The benefits of the 3rd Party Registration pattern include:

- The service code is less complex than when using the Self Registration pattern since its not responsible for registering itself
- The registrar can perform health checks on a service instance and register/unregister the instance based the health check

There are also some drawbacks:

- The 3rd party registrar might only have superficial knowledge of the state of the service instance, e.g. RUNNING or NOT RUNNING and so might not know whether it can handle requests. However, as mentioned above some registrars such as Netflix Prana perform a health check in order to determine the availability of the service instance.
- Unless the registrar is part of the infrastructure it’s another component that must be installed, configured and maintained. Also, since it’s a critical system component it needs to be highly available.

Chapter 7
Logging , Tracing and Health Checkup for Micro Services

7.1 Log aggregation

You have applied the Microservice architecture pattern. The application consists of multiple services and service instances that are running on multiple machines. Requests often span multiple service instances.

Each service instance generates writes information about what it is doing to a log file in a standardized format. The log file contains errors, warnings, information and debug messages.

7.1.1 Problem

How to understand the behavior of an application and troubleshoot problems?

Forces

- Any solution should have minimal runtime overhead

7.1.2 Solution

Use a centralized logging service that aggregates logs from each service instance. The users can search and analyze the logs. They can configure alerts that are triggered when certain messages appear in the logs.

Examples

- AWS Cloud Watch

Resulting Context

This pattern has the following issues:

- handling a large volume of logs requires substantial infrastructure

7.2 Application metrics

You have applied the Microservice architecture pattern.

7.2.1 Problem

How to understand the behavior of an application and troubleshoot problems?

Forces

- Any solution should have minimal runtime overhead

7.2.2 Solution

Instrument a service to gather statistics about individual operations. Aggregate metrics in centralized metrics service, which provides reporting and alerting. There are two models for aggregating metrics:

- push - the service pushes metrics to the metrics service
- pull - the metrics services pulls metrics from the service

Examples

- Instrumentation libraries:
  - Coda Hale/Yammer Java Metrics Library
  - Prometheus client libraries
- Metrics aggregation services
  - Prometheus
  - AWS Cloud Watch

7.2.3 Benefits and Drawbacks

This pattern has the following benefits:

- It provides deep insight into application behavior

This pattern has the following drawbacks:

- Metrics code is intertwined with business logic making it more complicated

This pattern has the following issues:

- Aggregating metrics can require significant infrastructure

7.3 Audit logging

You have applied the Microservice architecture pattern.

7.3.1 Problem

How to understand the behavior of users and the application and troubleshoot problems?

Forces

- It is useful to know what actions a user has recently performed: customer support, compliance, security, etc.

7.3.2 Solution

Record user activity in a database.

Examples

This pattern is widely used.

7.3.3 Resulting Context

This pattern has the following benefits:

- Provides a record of user actions

This pattern has the following drawbacks:

- The auditing code is intertwined with the business logic, which makes the business logic more complicated

7.4 Distributed tracing

You have applied the Microservice architecture pattern. Requests often span multiple services. Each service handles a request by performing one or more operations, e.g. database queries, publishes messages, etc.

7.4.1 Problem

How to understand the behavior of an application and troubleshoot problems?

Forces

- External monitoring only tells you the overall response time and number of invocations - no insight into the individual operations
- Any solution should have minimal runtime overhead
- Log entries for a request are scattered across numerous logs

7.4.2 Solution

Instrument services with code that

- Assigns each external request a unique external request id
- Passes the external request id to all services that are involved in handling the request
- Includes the external request id in all log messages
- Records information (e.g. start time, end time) about the requests and operations performed when handling a external request in a centralized service

This instrumentation might be part of the functionality provided by a Microservice Chassis framework.

Examples

The Microservices Example application is an example of an application that uses client-side service discovery. It is written in Scala and uses Spring Boot and Spring Cloud as the Microservice chassis. They provide various capabilities including Spring Cloud Sleuth, which provides support for distributed tracing. It instruments Spring components to gather trace information and can delivers it to a Zipkin Server, which gathers and displays traces.

The following Spring Cloud Sleuth dependencies are configured in build.gradle:

```
dependencies {

   compile "org.springframework.cloud:spring-cloud-sleuth-stream"

   compile "org.springframework.cloud:spring-cloud-starter-sleuth"

   compile "org.springframework.cloud:spring-cloud-stream-binder-rabbit"
```

RabbitMQ is used to deliver traces to Zipkin.

The services are deployed with various Spring Cloud Sleuth-related environment variables set in the docker-compose.yml:

```
environment:

   SPRING_RABBITMQ_HOST: rabbitmq

   SPRING_SLEUTH_ENABLED: "true"

   SPRING_SLEUTH_SAMPLER_PERCENTAGE: 1

                                                                  SPRING_SLEUTH_WEB_SKIPPATTERN:
"/api-docs.*|/autoconfig|/configprops|/dump|/health|/info|/metrics.*|/mappings|/trace|/
swagger.*|.*\\.png|.*\\.css|.*\\.js|/favicon.ico|/hystrix.stream"
```

This properties enable Spring Cloud Sleuth and configure it to sample all requests. It also tells Spring Cloud Sleuth to deliver traces to Zipkin via RabbitMQ running on the host called rabbitmq.

The Zipkin server is a simple, Spring Boot application:

```
@SpringBootApplication

@EnableZipkinStreamServer

public class ZipkinServer {



   public static void main(String[] args) {

      SpringApplication.run(ZipkinServer.class, args);

   }
}
```

It is deployed using Docker:

```
zipkin:
  image: java:openjdk-8u91-jdk
  working_dir: /app
  volumes:
    - ./zipkin-server/build/libs:/app
  command: java -jar /app/zipkin-server.jar --server.port=9411
  links:
    - rabbitmq
  ports:
    - "9411:9411"
  environment:
    RABBIT_HOST: rabbitmq
```

Resulting Context

This pattern has the following benefits:

- It provides useful insight into the behavior of the system including the sources of latency
- It enables developers to see how an individual request is handled by searching across aggregated logs for its external request id

This pattern has the following issues:

- Aggregating and storing traces can require significant infrastructure

7.5 Exception tracking

You have applied the Microservice architecture pattern. The application consists of multiple services and service instances that are running on multiple machines. Errors sometimes occur when handling requests. When an error occurs, a service instance throws an exception, which contains an error message and a stack trace.

7.5.1 Problem

How to understand the behavior of an application and troubleshoot problems?

Forces

- Exceptions must be de-duplicated, recorded, investigated by developers and the underlying issue resolved
- Any solution should have minimal runtime overhead

Solution

Report all exceptions to a centralized exception tracking service that aggregates and tracks exceptions and notifies developers.

Resulting Context

This pattern has the following benefits:

- It is easier to view exceptions and track their resolution

This pattern has the following drawbacks:

- The exception tracking service is additional infrastructure

7.6 Health Check API

You have applied the Microservice architecture pattern. Sometimes a service instance can be incapable of handling requests yet still be running. For example, it might have ran out of database connections. When this occurs, the monitoring system should generate a alert. Also, the load balancer or service registry should not route requests to the failed service instance.

7.6.1 Problem

How to detect that a running service instance is unable to handle requests?

Forces

- An alert should be generated when a service instance fails
- Requests should be routed to working service instances

7.6.2 Solution

A service has an health check API endpoint (e.g. HTTP /health) that returns the health of the service. The API endpoint handler performs various checks, such as

- the status of the connections to the infrastructure services used by the service instance
- the status of the host, e.g. disk space
- application specific logic

A health check client - a monitoring service, service registry or load balancer - periodically invokes the endpoint to check the health of the service instance.

Examples

The Microservices Example application is an example on an application implements a health check API. It is written in Scala and uses Spring Boot and Spring Cloud as the Microservice chassis. They provide various capabilities including a health check endpoint. The endpoint is implemented by the Spring Boot Actuator module. It configures a /health HTTP endpoint that invokes extensible health check logic.

To enable a /health endpoint, first define actuator as a dependency:

```
dependencies {

   compile "org.springframework.boot:spring-boot-starter-actuator"
```

Second, enable Spring Boot autoconfiguration:

```
@SpringBootApplication

class UserRegistrationConfiguration {
```

At this point, your application will have a health check endpoint with default behavior.

You can customize this behavior by defining one or more Spring beans that implement the HealthIndicator interface:

```
class UserRegistrationConfiguration {

   @Bean

    def discoveryHealthIndicator(discoveryClient : EurekaClient ) : HealthIndicator =
new DiscoveryHealthIndicator(discoveryClient)
```

A HealthIndicator must implement a health() method, which returns a Health value.

Resulting Context

This pattern has the following benefits:

- The health check endpoint enables the health of a service instance to be periodically tested

This pattern has the following drawbacks:

- The health check might not sufficiently comprehensive or the service instance might fail between health checks and so requests might still be routed to a failed service instance

7.7 Log deployments and changes

You have applied the Microservice architecture pattern.

7.7.1 Problem

How to understand the behavior of an application and troubleshoot problems?

Forces

- It useful to see when deployments and other changes occur since issues usually occur immediately after a change

7.7.2 Solution

Log every deployment and every change to the (production) environment.

Examples

A deployment tool can, for example, publish a pseudo-metric whenever it deploys a new version of a service. This metric can then be displayed alongside other metrics enabling changes in application behavior to be correlated with deployments. See Tracking Every Release by Mike Brittain

AWS Cloud Trail provides logs of AWS API calls.

7.7.3 Benefits

This pattern has the following benefits:

- Enables deployments and changes to be easily correlated with issues leading to faster resolution