Architectural Design Patterns

Daniel Hinojosa
@dhinojosa

Workshop Organization

- Every Pattern will contain
  - The Problem
  - The Pattern
  - A Diagram or Animation of the Pattern
  - The Solution
  - The Tradeoffs

“Unlike a developer, who must have a significant amount of technical depth to perform their job, a software architect must have a significant amount of technical breadth to think like an architect and see things with an architecture point of view.”

- Neal Ford & Mark Richards Fundamentals of Software Architecture

“A large part of the value of an architect is a broad understanding of technology and how to use it to solve particular problems. For example, as an architect, it is more beneficial to know that five solutions exist for a particular problem than to have singular expertise in only one.”

- Neal Ford & Mark Richards Fundamentals of Software Architecture

Architecture Styles

Layered Architecture

- Most common architectural pattern
- Organizes the codebase into horizontal layers, with each layer addressing one of the following technical concerns: interaction with the consumers, implementing business logic, and persisting the data.
- **Business Layer** - Layer is responsible for implementing and encapsulating the program's business logic
- **Data Access Layer** - Provides access to persistence mechanisms
- **Presentation Layer** - Implements the program's user interface for interactions with its consumers. The UI
- The dependency between the business logic and the data access layers makes this architectural pattern a good fit for a system with its business logic implemented using the *transaction script* or *active record pattern*

Layered Architecture

Layered Architecture

Layered Architecture Alternate

Hexagonal Architecture

- Ports and Adapters Architecture
- Decouple the system's business logic from its infrastructural components
- The infrastructure layer implements "adapters", which are concrete implementations

- Originally conceptualized by Alistair Cockburn
- Compliments Domain Driven Design, which we discuss next
- Separation of Concerns. Domain is encapsulated and interacts through the ports and adapters
- Decoupling of the business logic from all technological concerns makes the ports & adapters architecture a perfect fit

Hexagonal Architecture

Learning Domain Driven Design - Vlad Khononov

The Lab

- We will use ArchUnit
  - A Java library that will determine if packages, classes and code are in the right location
  - An essential library if you are intending to design applications according to a particular architecture, like the Hexagonal Architecture
  - We will be using Jitterted excellent Blackjack example notice how the application is divided cleanly

Domain Driven Design

Domain Driven Design

- Methodology focused about the important of the domain
- Build an *ubiquitous language* for domain
- Classifies Objects into Entities, Aggregates, Value Objects, Domain Events, and Service Objects
- One of the other important aspects to DDD is the notion of a bounded context and subdomains

Ubiquitous Language

- Define the terms of your domain
- Language of your business
- Goals
  - Precise
  - Consistent
  - Unambiguous

Value Objects

- Object that can be identified by the composition of its values
- No explicit identification is required
- Changing the attributes of any of the fields yields a different object
- Value Objects can be used to counter the “Primitive Obsession” code smell
- Typically immutable

```
class Color {
   int red;
   int green;
   int blue;
}
```

Entities

- Opposite of a value object and requires explicit identification
- Explicit identification is required
- For example, an Employee, just identified by an employee's first name and last name
- Entities are subject to change

```
class Employee {
   private Name name;
   //Constructors, equals, hashCode, etc.
}
```

↓

```
class Employee {
   private EmployeeId employeeId;
   private Name name;
   //Constructors, equals, hashCode, etc.
}
```

Aggregates

- An entity that manages a cluster of domain objects as a single unit
- Consider an `Order` to `OrderLineItem` relationship
- The root entity ensures the integrity of its parts
- Transactions should not cross aggregate boundaries
- They are domain objects(order, playlist), they are not collections, like List, Set, Map

```
class Album {
   private Name name;
   private List<Track> tracks;

   public void addTrack(Track track) {
      this.tracks.add(track);
   }

   public List<Track> getTracks() {
      return Collections.copy(tracks)
   }
}
```

Domain Services

- Stateless object that implements the business logic
- It naturally doesn't belong to any of the domain model's aggregates or value objects

```
class OrderTax {
   //no state
   private void calculateTaxWithRate(Order
         order, TaxRate taxRate) {
       //...
   }
}
```

Application Services

- Makes use of Repositories (Storage Abstractions)
- Aggregate instances and then sent for transforming to a transformed object
- The transformed object will typically be routed to the UI

```
class OrderService {
   private OrderRepository
              orderRepository;
   private void persistOrder(Order) {
       //...
   }
}
```

The Diagram

Learning Domain Driven Design - Vlad Khononov

The Diagram

Learning Domain Driven Design - Vlad Khononov

Contexts are neither big nor small, but useful

The Diagram

Learning Domain Driven Design - Vlad Khononov

Contexts are neither big nor small, but useful

Bounded Context Interaction

Partnership

- One team can notify a second team about a change in the API, and the second team will cooperate and adapt—no drama or conflicts
- Two way coordination between contexts
- Both sides cooperate on integration, neither is interested in blocking the other

Learning Domain Driven Design - Vlad Khononov

- Not a good fit for geographically distributed teams due to synchronization
- Good CI/CD is needed

Shared Kernel

- Cases when the same model of a subdomain, or a part of it, will be implemented in multiple bounded contexts.
- Shared model is designed according to the needs of all of the bounded contexts
- The model would have to be consistent across all contexts for others to use

Learning Domain Driven Design - Vlad Khononov

- After changing the shared kernel, integration tests would have to be invoked to ensure changes are not destructive

Customer-Supplier

- The supplier provides services to the customer
- Both the supplier and the customer can succeed independently
- But there is an imbalance of power, something has to give
- Either the Supplier dictates the terms, or the Customer does

Learning Domain Driven Design - Vlad Khononov

- This is where there are patterns between bounded contexts called conformist, anticorruption layer, and open-host patterns

Conformist

- Favor the upstream, the upstream doesn't care about the customer's needs
- Take it or leave it
- Often occurs with service providers where you have say about the terms or how they operate

Learning Domain Driven Design - Vlad Khononov

- Reasons include upstream may be an industry standard, a well-established model, or it is just good enough

Anti-Corruption Layer

- Favor the upstream, the upstream doesn't care about the customer's needs, same as *Conformist*
- Anticorruption Layer is a layer that will interpret the API changes from the supplier to it's own model
- Works well if the upstream model is inefficient or inconvenient, or the upstream contract changes often

Learning Domain Driven Design - Vlad Khononov

Open-Host Service

- The tables have turned! The power is with the customers
- It is now the supplier that has to adapt by providing multiple public interface customized for the customer
- This is done by decoupling the implementation from the public interface
- The supplier does not conform, but provides an adequate *published language*

Learning Domain Driven Design - Vlad Khononov

Learning Domain Driven Design - Vlad Khononov

What does it mean?

- You have defined each of the bounded contexts, and each of the context have their own domain.
- Each context can interact with each other via interface if within the same application, or using a port to interact with it outside (see Hexagonal)
- Support either Microservice style, or monolith style.

Learning Domain Driven Design - Vlad Khononov

- Your domain architecture is defined and responsive to change

Circuit Breaker

The Problem

- When a service synchronously requests from *a remote call* there is a big possibility of failure
- That remote call is that is a dependent may either be out or just taking too long
- Threads, since running synchronously will be taken up
- One service's failures or latency can cause a cascading failure

The Solution

- The client should invoke the remote service by proxy
- When the number of consecutive failures crosses a threshold, the circuit trips
- All attempts to invoke the remote service will ***fail*** immediately
- After the timeout expires, the circuit breaker allows limited traffic to go through
- If it those test connections succeed, the circuit breaker resumes normal operation
- Otherwise, a new timeout is triggered

Closed Circuit Breaker

- The request from the application is routed to the operation. Proxy maintains a count of the number of recent failures
- If the call to the operation is unsuccessful the proxy increments this count
- If the number of recent failures exceeds a specified threshold within a given time period, the proxy is placed into the Open state.
- At this point the proxy starts a timeout timer, and when this timer expires the proxy is placed into the Half-Open state.
- The purpose of the timeout timer is to give the system time to fix the problem that caused the failure before allowing the application to try to perform the operation again.

Open Circuit Breaker

- The request from the application fails immediately and an exception is returned to the application
- Usually Open is a positive thing, but in this case it is not.

Half-Open Circuit Breaker

- A limited number of requests from the application are allowed to pass through and invoke the operation
- If these requests are successful, it's assumed that the fault that was previously causing the failure has been fixed and the circuit breaker switches to the **Closed** state
- If any request fails, the circuit breaker assumes that the fault is still present so it reverts back to the **Open** state and restarts the timeout timer to give the system a further period of time to recover from the failure

The Diagram

Source: https://martinfowler.com/bliki/CircuitBreaker.html

Notes on Circuit Breaker

- You must handle the exception thrown by the circuit breaker, or use Either types so that you can have the opportunity to degrade, or offer better exceptions
- Review the types of exceptions that are thrown, many are different. You may need to adjust your strategy based on the exception
- A circuit breaker can and should log all the requests for health monitoring

Notes on Circuit Breaker

- A circuit breaker can and should log all the requests for health monitoring
- If these requests are successful, it's assumed that the fault that was previously causing the failure has been fixed and the circuit breaker switches to the **Closed** state
- If any request fails, the circuit breaker assumes that the fault is still present, so it reverts back to the **Open** state and restarts the timeout timer to give the system a further period of time to recover from the failure

The Tradeoffs

- Not used for private owned resources like an in-memory database
- This would not be a substitute for handling exceptions for internal programmatic functionality, this is an architectural pattern

The Lab

- We will use a library call Resilience4j
  - The heir apparent to Netflix' Hystrix used for performing retry, bulkhead, and short circuit, and more
  - https://resilience4j.readme.io/docs/

Retry

The Problem

- There can be transient faults when it comes to networking applications
  - Momentary loss of network
  - Unavailable Service
  - Service is busy
- Services may also throttle requests if the load is too high until resources start easing

The Solution

- Handle the failures with the following:
  - **Cancel**. If the fault indicates that the failure isn't transient or is unlikely to be successful if repeated, the application should cancel the operation and report an exception. For example, an authentication failure caused by providing invalid credentials is not likely to succeed no matter how many times it's attempted.
  - **Retry.** If the specific fault reported is unusual or rare, it might have been caused by unusual circumstances such as a network packet becoming corrupted while it was being transmitted. In this case, the application could retry the failing request again immediately because the same failure is unlikely to be repeated and the request will probably be successful.
  - **Retry after delay.** If the fault is caused by one of the more commonplace connectivity or busy failures, the network or service might need a short period while the connectivity issues are corrected or the backlog of work is cleared. The application should wait for a suitable time before retrying the request.

The Diagram

1: Application invokes operation on hosted service. The request fails, and the service host responds with HTTP response code 500 (internal server error).
2: Application waits for a short interval and tries again. The request still fails with HTTP response code 500.
3: Application waits for a longer interval and tries again. The request succeeds with HTTP response code 200 (OK).

Source: https://docs.microsoft.com/en-us/azure/architecture/patterns/retry

Retry Strategies

- Retries can be done using a backoff time period. This can be incremental or exponential
- Retries should log all the attempts for diagnosis
- Exponential Backoff is useful if the service is frequently busy
- Retries are very common with messaging systems, but this will often end in an *at least once* semantic

The Lab

- Again we will turn our attention to Resilience4j
  - This time we will perform a retry pattern so as to ensure that we can retry transient failures
  - We should back off those transient failures after we are not receiving a timely response

Bulkhead

The Problem

- Elements of an application are isolated into pools so that if one fails, the others will continue to function
- An application may constitute multiple services, with each service having one or more consumers
- Resources may become exhausted
- We will require that thread pools do not exceed a certain threshold

The Solution

- Partition Service instances into groups based on consumer load and availability requirements
- A consumer that calls multiple services may be assigned a connection pool for each service, and that connection pool can either have a task on a queue, or backed by another thread
- If a service begins to fail, it only affects the connection pool assigned for that service, allowing the consumer to continue using the other services

The Diagram

The Diagram

Task
Task
Task
Task
Bulkhead-A
srv-a
Task
Task
Task
Task
Bulkhead-B
srv-b

The Tradeoffs

- Great for isolating resources used to consume a set of backend services, especially if the application can provide some level of functionality even when one of the services is not responding.
- Perfect for isolating critical consumers from standard consumers.
- It protects the application from cascading failures.
- It does add some complexity

The Lab

- Again we will turn our attention to Resilience4j
  - This time we will perform a bulkhead pattern so as to ensure that not all threads will be consumed for various purposes

Ambassador

The Problem

- Applications require a multitude of services like
  - Circuit breaking
  - Monitoring
  - Metering
  - Datasource Connectivity and Proxying
- There are risks when it comes to application you are unfamiliar with.

The Solution

- An Ambassador pattern is a proxy for your application to external services
- Deploy the proxy to the same environment, e.g. same pod
- The proxy can monitor performance metrics such as latency or resource usage, and this monitoring happens in the same host environment as the application
- You can make updates to the ambassador without affecting the legacy application
- This can be typically done as a sidecar in the same pod

The Tradeoffs

- There is minimal latency overhead
- The solution is better off to be integrated to the application itself
- When client connectivity features are consumed by a single language, a better option might be a client library that is distributed to the development teams as a package

The Diagram

The ambassador will be intercept traffic and provide services for caching, security, circuit breaking, and more

Ambassador vs Sidecar

Ambassador:

- Used for monitoring, logging, routing, security (like TLS), all *networking based tasks*
- Used to provide circuit breaking
- Ambassador services can be deployed as a sidecar to accompany the lifecycle of a consuming application or service

Side Car:

- Additional container attached to a parent application and provides supporting features for the application
- If it includes networking it would be considered an ambassador as well
- If it purely logging, configuration it is typically just a sidecar
- It can get rather blurry when trying to distinct between the two

The Diagram

Service A
Service B
HTTP/1.1, HTTP/2,
gRPC or TCP --
with or without
mTLS
Proxy
Proxy
Policy checks,
telemetry
Adapter
Mixer
Adapter
Configuration
data to proxies
Configuration
data
TLS certificates
to proxies
Pilot
Galley
Citadel
Control Plane

Event Sourcing

The Problem

- Typical CRUD Database have issues
  - It is a change log where information get overridden, and not all data is like that
  - Slowdown of performance, when writing to a typical datastore there can be contention and lack of scalability
  - Unless there is an implementation of auditing much of this data is lost
- Event Sourcing compliments CQRS (to be covered soon)

The Solution

- Events are recorded in an append only store
- Events are described with discrete events like:
  - `AddedItemToCart`
  - `RemovedItemFromCart`
  - `ClearedCart`
- Therefore it can be used to materialize view, prepare the data for the purposes of being *efficiently* read by a UI or business analytics for consumption

The Diagram

Append-Only & Immutable

```
3002
```

```
{“event”: “ItemAdded”,
 “item_id”: “2001”
 “quantity”: 3}
```

The Diagram

The Diagram

Append-Only & Immutable

The Diagram

Append-Only & Immutable

```
3043
```

```
{“event”: “ItemAdded”,
 “item_id”: “1644”
 “quantity”: 2}
```

The Diagram

Append-Only & Immutable

The Diagram

Append-Only & Immutable

The Diagram

Append-Only & Immutable

Producer

Key	Value
3002	{“event”: “ItemAdded”,“item_id”: “2001”, “quantity”, “3”}
3043	{“event”: “ItemAdded”,“item_id”: “1644”, “quantity”, “2”}

Consumer

```
3002
```

```
{“event”: “ItemAdded”,
 “item_id”: “2001”
 “quantity”: 1}
```

The Diagram

Append-Only & Immutable

The Diagram

Append-Only & Immutable

Producer

Key	Value
3002	{“event”: “ItemAdded”,“item_id”: “2001”, “quantity”, “3”}
3043	{“event”: “ItemAdded”,“item_id”: “1644”, “quantity”, “2”}
3002	{“event”: “ItemAdded”,“item_id”: “2001”, “quantity”, “1”}

Consumer

The Tradeoffs

- **Eventually Consistent -** The data that is available to be read will soon be there, but will not be there immediately
- Immediately real-time consistency is not available. It’s very close though.
- If you do not require audit trails, history, and roll backs, it might be overkill
- If you do not expect a lot of conflicts, this too will be overkill

Competing Consumers

The Context

- We require constant processing of a number of requests
- Pass them through a messaging system to another service (a consumer service) that handles them asynchronously
- Using a single instance for consuming can be daunting
- Consumers might be overwhelmed
- Workloads need to be balanced across multiple consumers

The Solution

- Use a message queue or pub/sub to send messages over to the consumer
- Each consumer can process the message in a way that doesn't interfere with other consumers
- Consumers become fault tolerant
- Improves reliability, delivered at least once
- By using offsets any consumer that takes on messages that the others haven't then the consumer can process those same messages

The Diagram

topic-a
Partition 0:
Partition 1:
Partition 2:
Partition 3:

consumer-1

The Diagram

topic-a
Partition 0:
Partition 1:
Partition 2:
Partition 3:
consumer-1
consumer-2
consumer-3
consumer-4

The Tradeoffs

- It may not be easy to separate the application workload into discrete tasks
- There's a high degree of dependence between tasks
- Tasks must be performed synchronously, and the application logic must wait for a task to complete before continuing
- Tasks must be performed in a specific sequence

The Lab

- Let’s take a look at Kafka Consumers which happen to adhere to this design pattern

Claim Check

The Context

- Many messaging applications can’t accept really large payloads
- Send a claim check to the messaging platform and store that payload to an external service.
- This will prevent the messaging service from being overwhelmed or being slowed down
- Less expensive overall, since storage is cheaper than messaging

The Solution

- Store the entire message payload into an external service, like a datastore
- Get a reference, like a integer identifier to stored payload
- Attach and send the reference to the message broker *inside of the message*
- The consumer of the message will use the identifier to retrieve the large payload

The Diagram

The Diagram

The Diagram

The Diagram

The Lab

- Next we will use Kafka and Schema Registry to implement the following behaviors
  - Loading a Schema on a Datastore called schema registry
  - Seeing that the consumer gets the same data

Materialized View

The Problem

- Currently much of the data revolves on how it is stored, not how it is read
- Data when read needs to transformed and prepared
- Each entity typically has too much information for it to be queryable and usable
  - For example, one data entry in a document style database can have other aggregates that are not absolutely necessary
- Data may also need to be joined, cleansed, or engineered for a particular purpose, like machine learning

The Diagram

Orders

Id	State	Customer
1	NY	Carlos
2	AZ	Ann
3	CA	Pritha
4	CA	Antoni
5	AZ	Amani
6	NY	Widelene

Products

ID	Name	Cost
1	Dr. Magestic's Tonic	40
2	Dragonfeel Pillows	200
3	Sumatra Fresh	430
4	Mars Rover	212
5	Ennui Engine	190
6	Pittsburgh Passion	410

Order Items

ID	Order ID	Quantity	Product ID
1	1	23	1
2	1	2	2
3	2	4	4
4	2	5	6
5	2	60	5
6	3	10	4

ID	State	Qty	Total
1	NY	30	4000
2	AZ	75	3500
3	CA	123	12322
4	NJ	80	3892
5	TX	3	1200
6	PA	90	3304

Materialized Report

Id	State	Customer
1	NY	Carlos
2	AZ	Ann
3	CA	Pritha
4	CA	Antoni
5	AZ	Amani
6	NY	Widelene

ID	Name	Cost
1	Dr. Magestic's Tonic	40
2	Dragonfeel Pillows	200
3	Sumatra Fresh	430
4	Mars Rover	212
5	Ennui Engine	190
6	Pittsburgh Passion	410

ID	Order ID	Quantity	Product ID
1	1	23	1
2	1	2	2
3	2	4	4
4	2	5	6
5	2	60	5
6	3	10	4

ID	State	Qty	Total
1	NY	30	4000
2	AZ	75	3500
3	CA	123	12322
4	NJ	80	3892
5	TX	3	1200
6	PA	90	3304

The Solution

- Create a different perspective of the data that you need
- This can be implemented by the datastore itself
  - Oracle
  - PostgreSQL
- Can be performed by Stream Processing Frameworks
  - Kafka
  - Spark
  - Flink

The Tradeoffs

- If your source is already simple and easy to query, there is no need
- If immediate consistency is required, then query direct

CQRS

CQRS

- **Command and Query Responsibility Segregation**
- Separating reads, add-updates, for a databases
- Benefits include better performance, scalability, and security
- Better evolution over time
- Prevents Merge Conflicts:
  - Database locking ensures that the updates don't change the same data concurrently, but it doesn't ensure that multiple independent changes result in a consistent data model.

The Diagram

What if we need to query...
`SELECT state, count(*) from orders group by state`, over and over?

State	Amount	Customer
NY	4000	Carlos
AZ	300	Ann
CA	1000	Pritha
CA	340	Antoni
AZ	2133	Amani

The Diagram

Write Database

State	Amount	Customer
NY	4000	Carlos
AZ	300	Ann
CA	1000	Pritha
CA	340	Antoni
AZ	2133	Amani

Read Database

State	Total
NY	4000
AZ	2433
CA	1340

The Diagram

Write Database

State	Amount	Customer
NY	4000	Carlos
AZ	300	Ann
CA	1000	Pritha
CA	340	Antoni
AZ	2133	Amani
NY	900	Widelene

Read Database

State	Total
NY	4000
AZ	2433
CA	1340

The Diagram

Write Database

State	Amount	Customer
NY	4000	Carlos
AZ	300	Ann
CA	1000	Pritha
CA	340	Antoni
AZ	2133	Amani
NY	900	Widelene

Read Database

State	Total
NY	4900
AZ	2433
CA	1340

The Lab

- Next we will use Kafka to implement the following behaviors
  - Event Sourcing
  - Materialized Views
  - CQRS (Command Query Responsibility Segregation)

Strangler Fig

The Context

- Services can become obsolete.
- New features over time can make things difficult to manage
- Need responsible and gradual migration to the new system
- Clients would need to be updated to the new location
- Once moved, the old system can be responsibly removed

The Solution

- Create a façade called the *Strangler Facade* which will intercept requests that is going into a legacy system.
- The façade routes the requests to either the legacy or new services
- Existing features can be migrated to the new system gradually and consumers use the same interface

The Diagram

The Diagram

The Diagram

The Tradeoffs

- Ensure that the façade doesn’t become a bottleneck
- Ensure that the façade keeps up with the migration
- After the migration, the legacy would go away or kept

Canary Deployments

- A Different view of the Strangler Fig
- Deployment Strategy - Popularly used by Netflix
- Should be a part of CI/CD
- Rolling out releases to a subset of users or servers and measure the response typically using metrics packages like Prometheus
- No Downtime
- If the subset fails, doesn’t perform well, or doesn’t meet standards, rollback to the previous

Canary Deployments

Canary Deployments

Canary Deployments

Canary Deployments

How to do with Kubernetes

- Use Label Selectors on the Service and appropriate sizing
- Use a Deployment object of the workload
- For consistency with common Deployment object, update the version of the original deployment object if continuing with the rollout

Gatekeeper

The Context

- Services may have potentially sensitive information
- If a hacker is able to get into the hosting environment, they can have access to a gamut of information like credentials, storage keys and datastores.

The Solution

- Create a façade that will be entry point of all requests
- The Gateway facade will be in charge of:
  - Accepting Requests
  - Sanitizing the Requests
  - Validating the Requests
  - Mutating the Requests

The Solution

- Create a façade that will be entry point of all requests
- The Gateway facade will be in charge of:
  - Accepting Requests
  - Sanitizing the Requests
  - Validating the Requests
  - Mutating the Requests

The Diagram

The Diagram

The Diagram

Internet
kubectl
Worker node
Master node
kubelet
kube-proxy
Pod
Pod
docker
API server
controller-manager
(replication, namespace,
serviceaccounts, ...)
scheduler
Worker node
kubelet
kube-proxy
Pod
Pod
docker
etcd

Valet Key

The Context

- Services may:
  - Reach into a storage account and stream it to a client
  - Accept a payload from a client and store it into a storage account
- Each of the scenarios may present an unnecessary middleman
- We can provide any client access to the datastore without interaction to the service, but that means that the service will not longer act as *Gatekeeper*
- Applications must be able to securely control access to data in a granular way

The Solution

- Restrict access to the data store's public connection
- Provide the client with a key or token that the data store can validate.
- The token is referred to as a valet key
- Valet key has a time to live
- Contains access permissions like roles, which can be specific

The Diagram

Amazon Security Token Service
https://docs.aws.amazon.com/STS/latest/APIReference/welcome.html

The Diagram

https://www.vaultproject.io/docs/secrets/aws

The Lab

- We will use Vault to generate tokens to use with a PostgreSQL database
  - We will initialize the connection to Vault
  - Set up a connection and role
  - Get a temporary credential to the database which a client can use to gain direct access.

Data Mesh

The Context

- Data can be siloed *Database per Service*. To circumvent, we created a data lake or *shared database*
- Problem is that the data lake, ended up as a data monolith
- By using a shared storage, we lost governance, and ownership
- We wanted to apply bounded context patterns, and did so with services, but lost the point with data.

The Solution

- There are two patterns: Database per Service, and Shared Database
- Data Mesh is an attempt at bringing not only the database but owning all data
- Data Mesh is about providing data ownership per context, and follows these principles:
  - Domain-oriented decentralized data ownership and architecture
  - Data as a product
  - Self-serve data infrastructure as a platform
  - Federated computational governance
- *The accountability of data quality shifts upstream as close to the source of the data as possible.*

The corps was composed of all arms of the service, was **self-sustaining**, and could fight on its own until other corps could join in the battle. The corps itself was a headquarters to which units could be attached. **It might have attached two to four divisions of infantry with their organic artillery, it had its own cavalry division and corps artillery, plus support units**. With this organization a corps was expected to be able to hold its ground against, or fight off an enemy army for a least a day, when neighboring corps could come to its aid. "Well handled, it can fight or alternatively avoid action, and maneuver according to circumstances without any harm coming to it, because an opponent cannot force it to accept an engagement but if it chooses to do so it can fight alone for a long time."

https://www.napoleon-series.org/military-info/organization/c_armycorps.html

The Diagram

https://martinfowler.com/articles/data-mesh-principles.html

The Diagram

https://martinfowler.com/articles/data-mesh-principles.html

The Diagram

Anti-pattern: responsibility for data becomes the domain of the DWH team

Pattern: Ownership of a data asset given to the "local" team that is most familiar with it

Centralized Data Ownership

Decentralized Data Ownership

https://www.confluent.io/blog/how-to-build-a-data-mesh-using-event-streams/

The Diagram

Domain-oriented decentralized data ownership and architecture	So that the ecosystem creating and consuming data can scale out as the number of sources of data, number of use cases, and diversity of access models to the data increases; simply increase the autonomous nodes on the mesh.
Data as a product	So that data users can easily discover, understand and securely use high quality data with a delightful experience; data that is distributed across many domains.
Self-serve data infrastructure as a platform	So that the domain teams can create and consume data products autonomously using the platform abstractions, hiding the complexity of building, executing and maintaining secure and interoperable data products.
Federated computational governance	So that data users can get value from aggregation and correlation of independent data products - the mesh is behaving as an ecosystem following global interoperability standards; standards that are baked computationally into the platform.

https://martinfowler.com/articles/data-mesh-principles.html

Saga

The Context

- We require distributed transactions across through events
- A transaction is a single unit of logic or work, sometimes made up of multiple operations.
- Within a transaction, an event is a state change that occurs to an entity, and a command encapsulates all information needed to perform an action or trigger a later event.
- Transactions must be atomic, consistent, isolated, and durable (ACID). Transactions within a single service are ACID, but cross-service data consistency requires a cross-service transaction management strategy.

ACID Transactions

- **Atomicity** is an indivisible and irreducible set of operations that must all occur or none occur.
- **Consistency** means the transaction brings the data only from one valid state to another valid state.
- **Isolation** guarantees that concurrent transactions produce the same data state that sequentially executed transactions would have produced.
- **Durability** ensures that committed transactions remain committed even in case of system failure or power outage.

The Diagram

Choreography based pattern

Source: Building Microservices, 2nd Edition Sam Newman Published by O'Reilly Media, Inc.

The Diagram

Orchestrator based pattern where state of the transaction is owned by a service

Source: Building Microservices, 2nd Edition Sam Newman Published by O'Reilly Media, Inc.

The Diagram

https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-data-persistence/saga-pattern.html

Thank You

- Email: dhinojosa@evolutionnext.com
- Github: https://www.github.com/dhinojosa
- Twitter: http://twitter.com/dhinojosa
- Linked In: http://www.linkedin.com/in/dhevolutionnext