Lund&Bendsen
developing developers

Introduction

Whoami

Lund&Bendsen
developing developers

- Martin Højgaard Clausen
- 50 years old
- Live in Copenhagen area
- Married to Louise
- Civil engineer
- 25+ years of software experience
- Independent consultant

- Email: mhh.clausen@gmail.com

Agenda – 2 days

Lund&Bendsen
developing developers

- **Day one**
  - Introduction
  - Kafka overview
  - Kafka broker
  - Kafka producer
  - Kafka consumer
- **Day two**
  - Kafka Spring
  - Kafka Connect
  - Kafka Streams

Goals

Lund&Bendsen
developing developers

- Be familiar with principles, benefits and challenges with distributed systems and in particular event driven architecture (EDA).
- Explain the Kafka platform and its architecture.
- Be able to build a Spring Boot application that integrates with Kafka.

Tell us about yourself

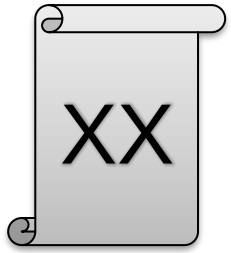
Lund&Bendsen
developing developers

- What is your job role?
- Years of experience?
- Have you worked with Kafka previously?
- Are you planning to use Kafka?
- What are your expectations for this course?

Practical stuff

Lund&Bendsen
developing developers

- All exercises will be done on a vm running in Azure
- Exercises can be worked on individually or in groups as you prefer
- Solutions are provided for most exercises
- Sometimes I will do the exercise and ask you to repeat it
- Raise you hand or feel free to interrupt me if you have questions!



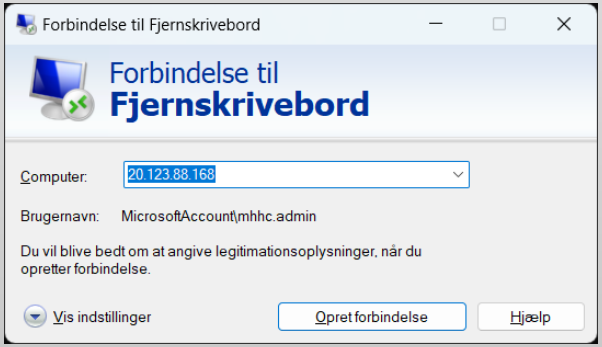
- Lunch time 11:30 - ?
- Small coffee breaks every hour?
- Anything else?

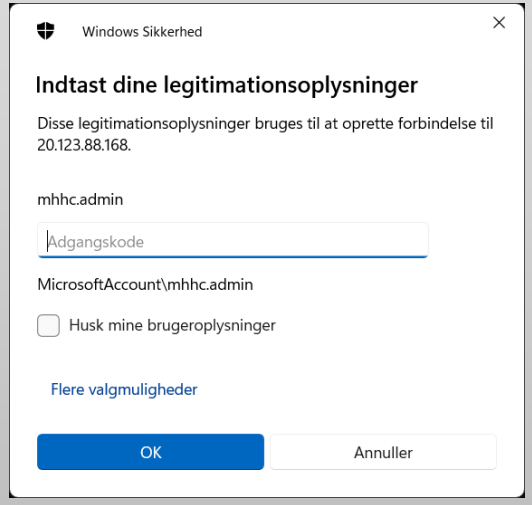
Exercise

Lund&Bendsen
developing developers

1.1

- Hand out of IP addresses
- Login to virtual machine using Microsoft Terminal Services Client
  - Type **mstsc** in the search field and enter your IP address
  - Enter username and password `mhhc.admin/thisissecret2023#`

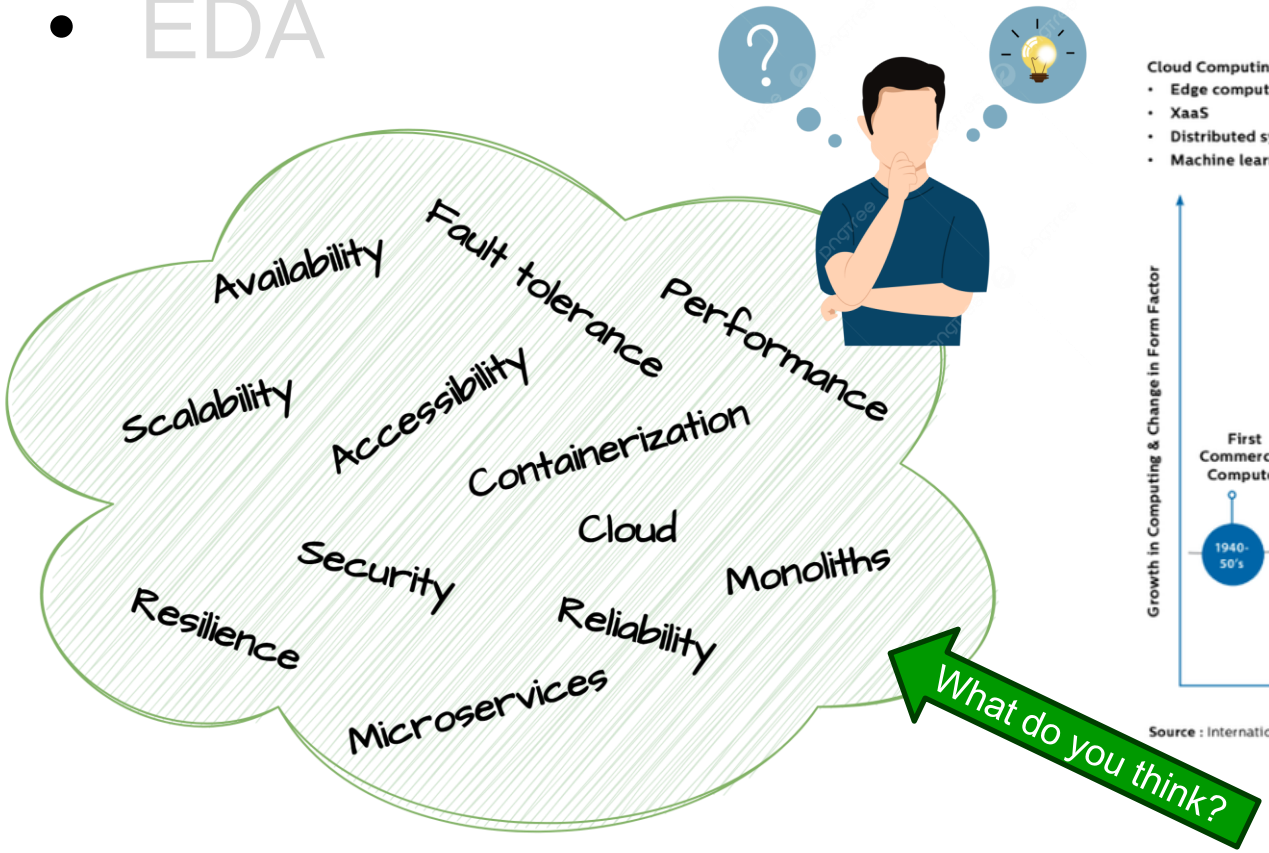


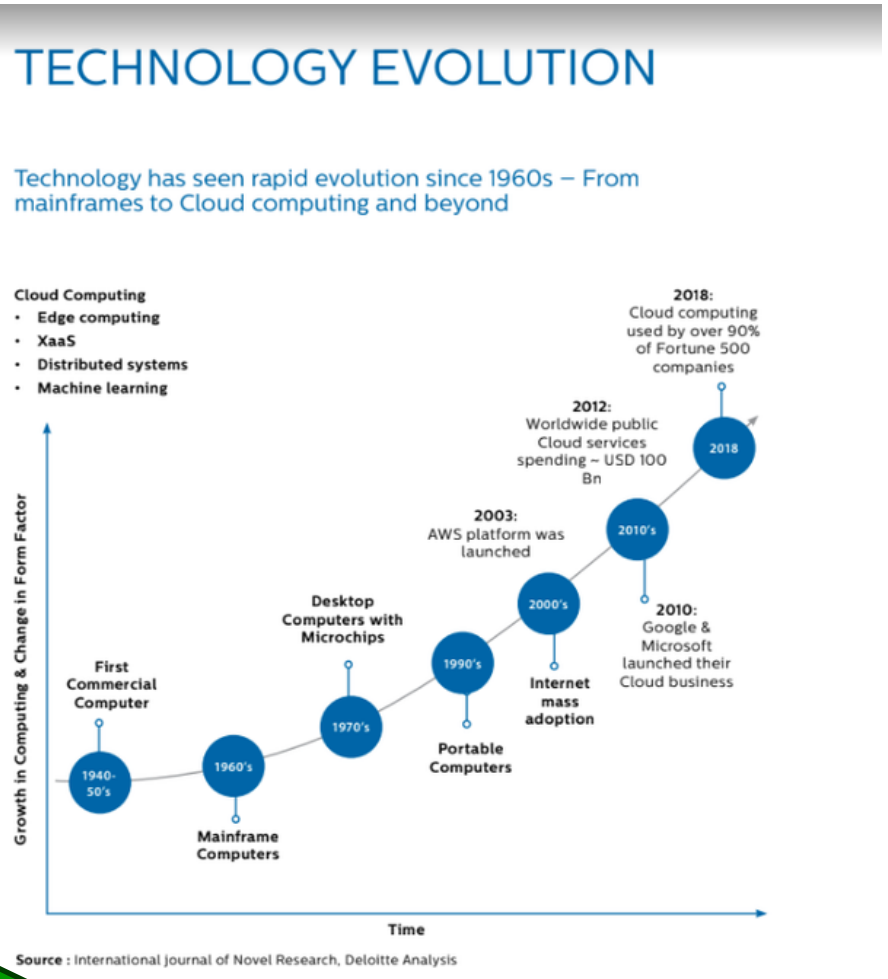


Agenda

Lund&Bendsen
developing developers

- Software architecture
- Communication
- EDA





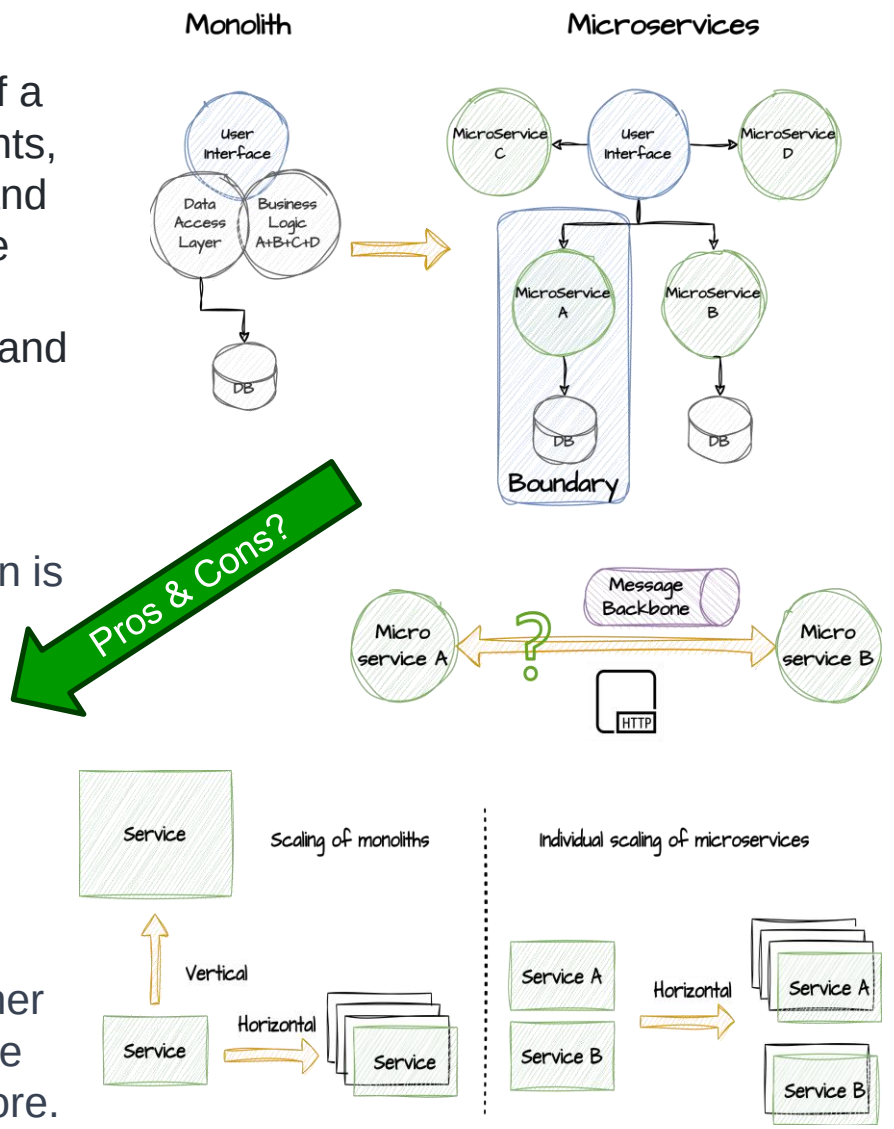
Monolith & microservices

Lund&Bendsen
developing developers

Software architecture refers to the fundamental structure of a software system. It's a blueprint that outlines the components, their relationships, and the principles guiding their design and evolution. This design includes various aspects such as the system's components, their functionalities, the interactions between them, and the properties that govern their design and evolution. Often when discussions on enterprise software architecture is made, two different approaches are being mentioned. The non-distributed and the distributed:

Monolith: In a monolithic architecture, the entire application is built as a single, tightly integrated unit. All components and modules are interconnected, and there is typically a single codebase and database. Calls between modules are in-process.

Microservices: Microservices architecture divides the application into small, isolated, independently deployable services. Each service focuses on a specific business capability (single responsibility) and communicates with other services via well-defined APIs or preferably over a message backbone. Each microservice typically has its own state store.

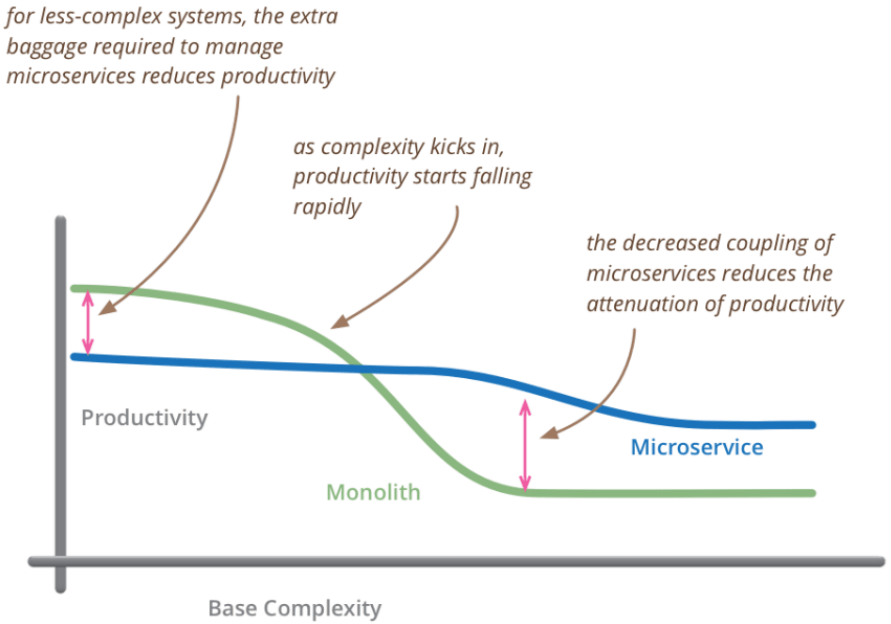


Monolith & microservices

Lund&Bendsen
developing developers

Monolith and **microservice** architectures are two fundamentally different approaches to designing and building software systems. They differ in various aspects, including their scalability, development process, and deployment.

The choice between a monolithic and microservices architecture depends on various factors, including the size and complexity of the application, the development team's expertise, scalability requirements, and organizational preferences. It's important to carefully consider these factors when deciding on the architecture for a software project.



but remember the skill of the team will outweigh any monolith/microservice choice

Microservice architecture

Lund&Bendsen
developing developers

Scalability: Microservices allow for granular scalability, where you can scale individual services independently based on their specific resource needs.
Flexibility: You can use different technologies and programming languages for different microservices, allowing you to choose the right tool for each job.
Isolation: Failures in one microservice generally don't impact the entire system, thanks to isolation, reducing the blast radius of issues.
Continuous Deployment: Microservices are conducive to continuous deployment and easier rollbacks since changes are limited to individual services.

Complexity: Microservices introduce additional complexity in terms of service discovery, load balancing, and inter-service communication.
Testing and Debugging: Testing and debugging can be more complex, especially when dealing with interactions between multiple services (integration testing).
Operational Overhead: Managing and monitoring multiple services can require additional operational overhead.
Latency: Inter-service communication over a network can introduce latency compared to in-memory calls in a monolithic application.

Monolithic architecture

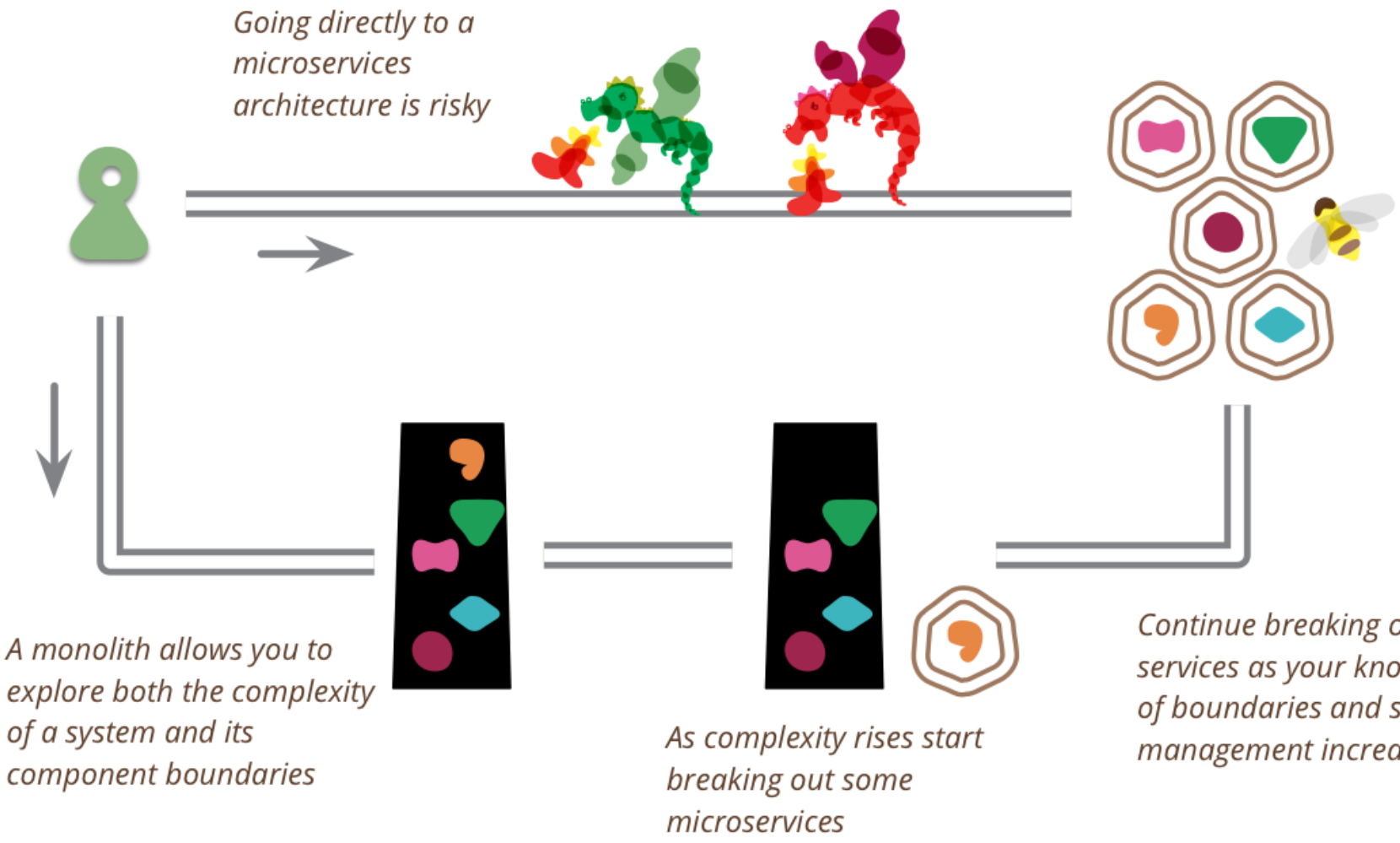
Lund&Bendsen
developing developers

Simplicity: Monoliths are typically simpler to develop and manage, especially for smaller applications, as there's only one codebase to deal with.
Performance: Monoliths can be more efficient when all components are tightly integrated, as there's no overhead for inter-service communication.
Easier Debugging: Debugging and troubleshooting can be simpler in a monolithic application because there are fewer moving parts and dependencies to consider.
Testing: Testing is often easier since the entire application can be tested as a whole.

Scalability: Scaling a monolithic application can be challenging, as you often need to scale the entire application even if only certain components require more resources.
Maintainability: As monolithic applications grow, they tend to become harder to maintain and update due to their size and complexity.
Technology Stack: You're limited to a single technology stack and programming language for the entire application.
Deployment: Deployments can be riskier because any change affects the entire application, potentially leading to downtime.

Martin Fowler

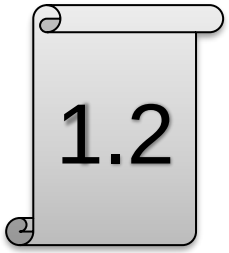
Lund&Bendsen
developing developers



Exercise

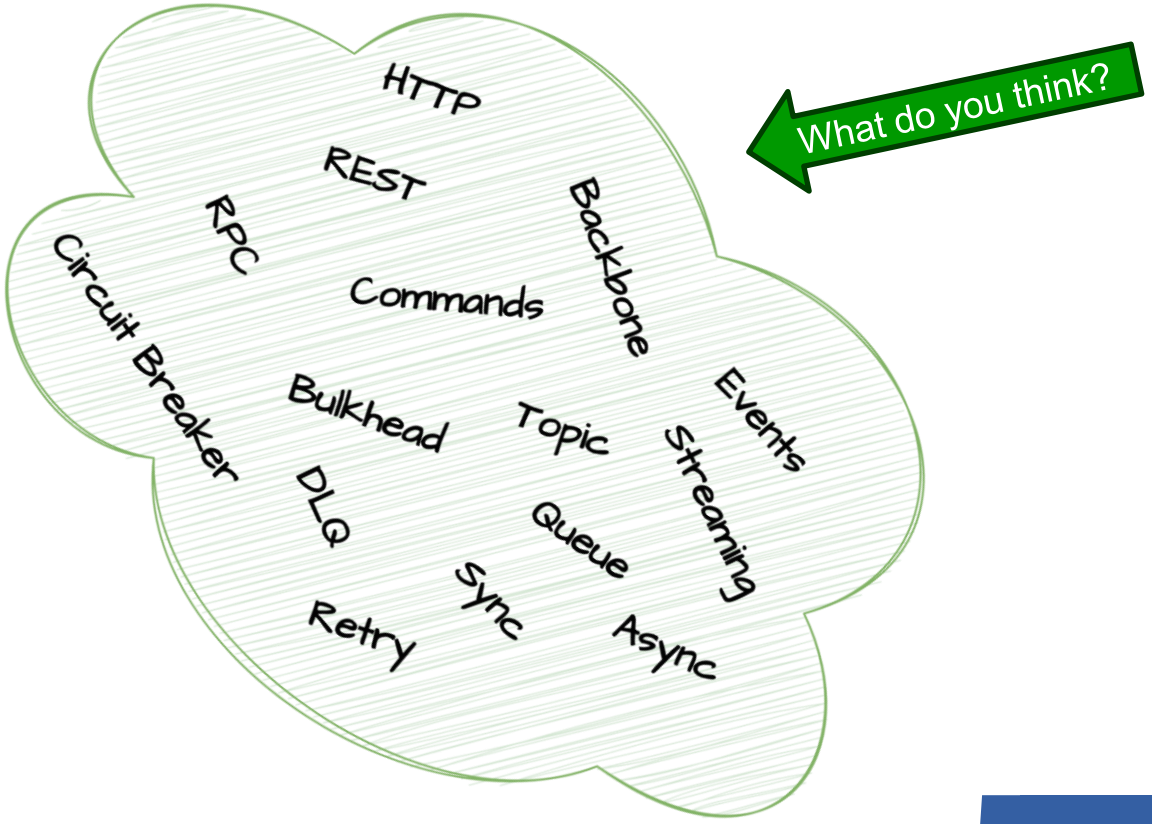
Lund&Bendsen
developing developers

- Clone repository
  - Open windows powershell by typing ‘powershell’ in the search field in the taskbar
  - In powershell navigate to the root of your c: drive by typing **cd**
  - Type: **git clone** https://github.com/LundOgBendsen/LB2628-Kafka.git
  - Type **ls** to see the folders in root. You should see a folder called LB2628-Kafka
  - Type **cd LB2628-Kafka** to navigate inside that folder.
  - Walk-thorough of the project folders



Communication

- Software architecture
- Communication
- EDA



The Reactive Manifesto

Lund&Bendsen
developing developers

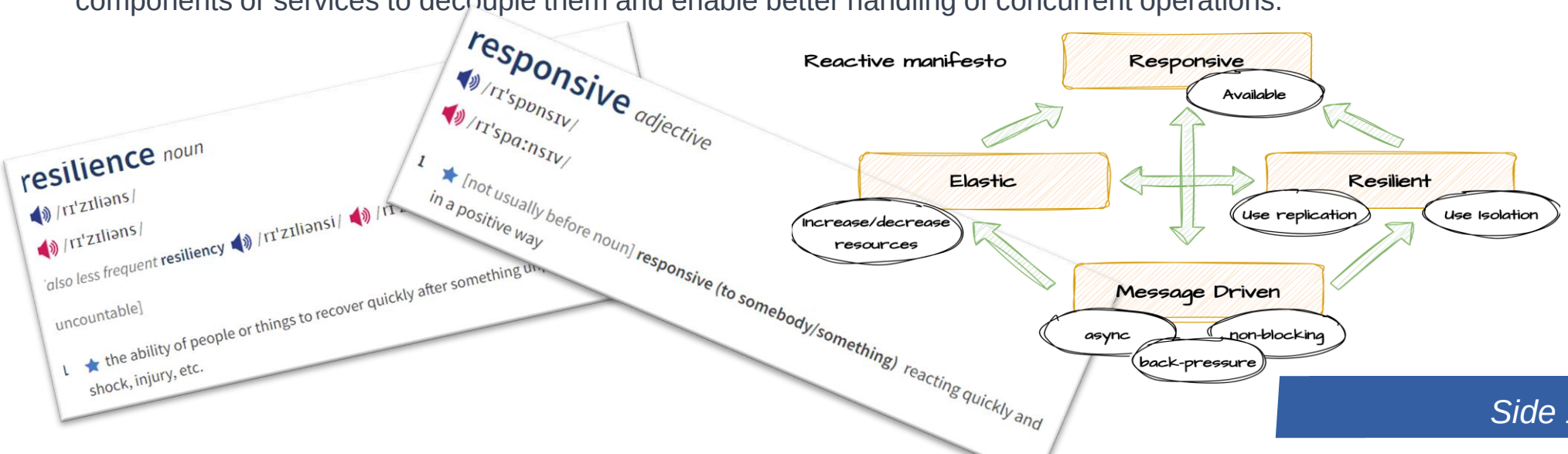
Published in 2014 by Jonas Bonér, 'The Reactive Manifesto' formulates several principles and best practices when building distributes systems. The emphasis lies on message-driven, making the components in the system loosely-coupled by favoring an asynchronous programming model. Having loosely coupled components brings several other qualities to the system.

Responsiveness: This principle emphasizes the need for systems to respond promptly to user requests and external events. The system must remain available even under heavy loads.

Resilience: Reactive systems are designed to detect and handle failures gracefully, with mechanisms such as fault tolerance, error recovery, and redundancy.

Elasticity: Elasticity refers to the capability of a system to scale its resources up or down dynamically in response to changing workloads

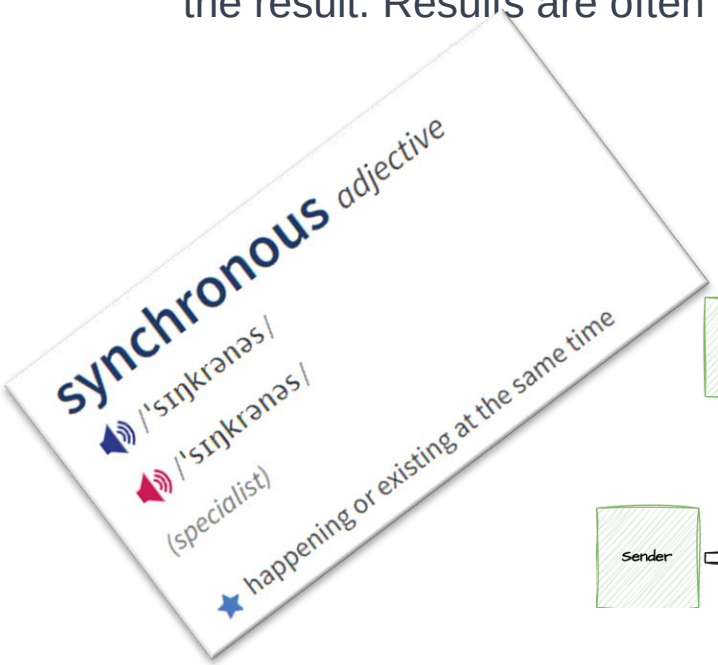
Message-Driven: Message-driven architecture involves using asynchronous message passing between components or services to decouple them and enable better handling of concurrent operations.

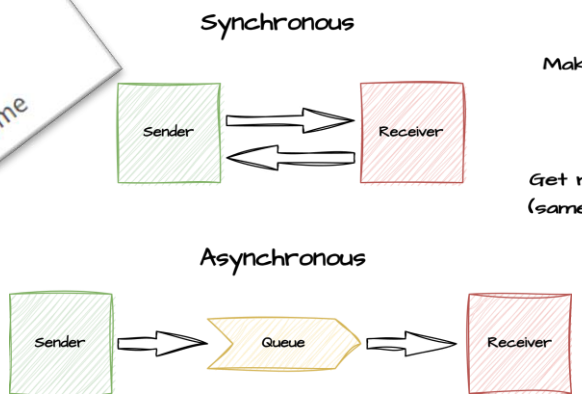


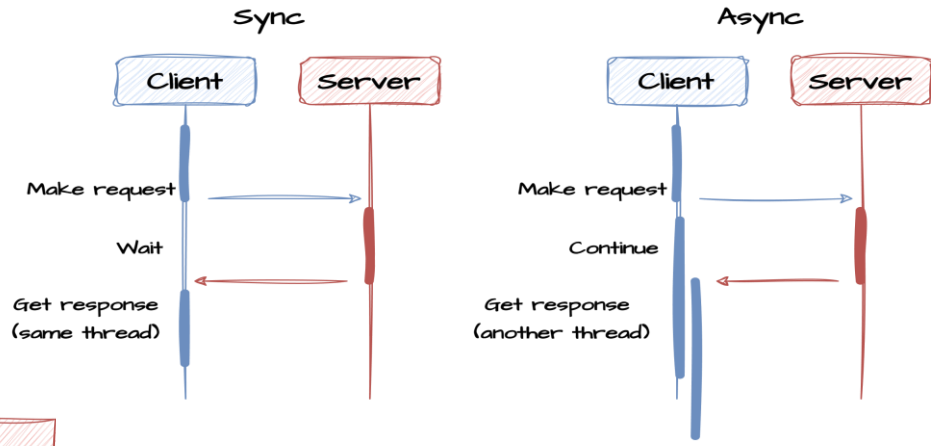
Sync vs Async

Lund&Bendsen
developing developers

- **Sync Calls**: Synchronous calls are blocking and occupies a thread while waiting for the response from the server to complete. This is suitable for simple, straightforward operations where blocking the caller's execution is acceptable and the response time is fast and predictable. Can be combined with various resiliency tools (circuit-breaker, retry and bulkhead) to reduce risk of thread starvation
- **Async Calls**: Asynchronous calls are preferred for tasks that involve waiting for external resources, such as I/O operations (e.g., file I/O, network requests), user interactions in graphical user interfaces (GUIs), and tasks where responsiveness and scalability are critical. Async calling code is typically harder to comprehend as it relies on Futures / callbacks to communicate the result. Results are often handled on a different thread than the calling thread







Synchronous calls

Lund&Bendsen
developing developers

Blocking: In a synchronous call, the caller waits for the called function to complete before proceeding with further execution. The execution of the caller is blocked until the called function finishes its task. A timeout can often be specified to avoid endless waiting

Order of Execution: Synchronous calls are executed sequentially, one after the other, in the order they are invoked. The program follows a predictable and linear flow.

Response Time: Since synchronous calls block the caller, they can lead to unforeseeably long processing time and impact other parts of the caller code. If synchronous calls are part of a transaction or other time-critical operations, these operations may time out.

Resource Utilization: Synchronous calls can be resource-intensive, as resources such as CPU and memory may be tied up while waiting for the called function to return.

Simplicity: Synchronous calls are often simpler to reason about and implement, as they follow a straightforward execution flow.

Asynchronous calls

Lund&Bendsen
developing developers

Non-Blocking: In an asynchronous call, the caller does not wait for the called function to complete. Instead, the caller continues with its execution, and the called function runs in the background or executes in a remote process and the response is handled in a separate process.

Parallel Execution: Asynchronous calls can be executed in parallel or concurrently with other tasks, allowing for better utilization of resources and potentially faster program execution

Response Time: Async calls can lead to faster response times, especially when dealing with tasks that may take some time to complete, such as I/O bound operations..

Resource Efficiency: Asynchronous calls are typically more resource-efficient because they do not block resources while waiting. This can result in better scalability and responsiveness.

Complexity: Implementing and managing asynchronous code can be more complex than synchronous code. It often involves handling callbacks, promises, or async/await constructs to manage the flow of execution.

Concurrency Control: Asynchronous code may require additional attention to concurrency control to prevent issues like race conditions and data inconsistency when multiple tasks run concurrently.

Error Handling: Handling errors in asynchronous code can be more challenging due to the distributed nature of async operations. Proper error handling and debugging is crucial.

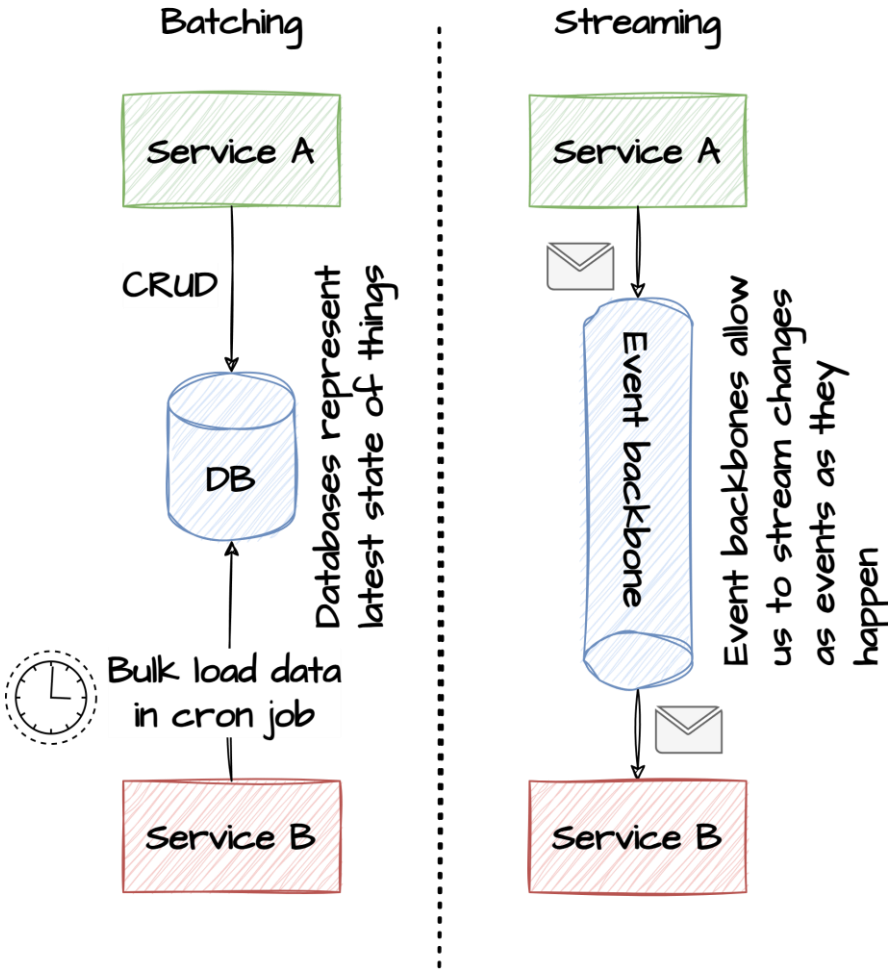
Data at-rest or in-transit

Lund&Bendsen
developing developers

With a message-driven architecture focus changes from data at-rest (stored in a data-store) to data in-transit (infinite stream of messages)

Database (at-rest): Data at rest is typically held in data-stores such as relational database tables. Often, these tables hold information about the current state of the entity that the table models. Data are written to the database using CRUD operations through a well-defined API. Consumers of the data are not notified about changes to the entities. Exchange of data between services happen through batch processing.

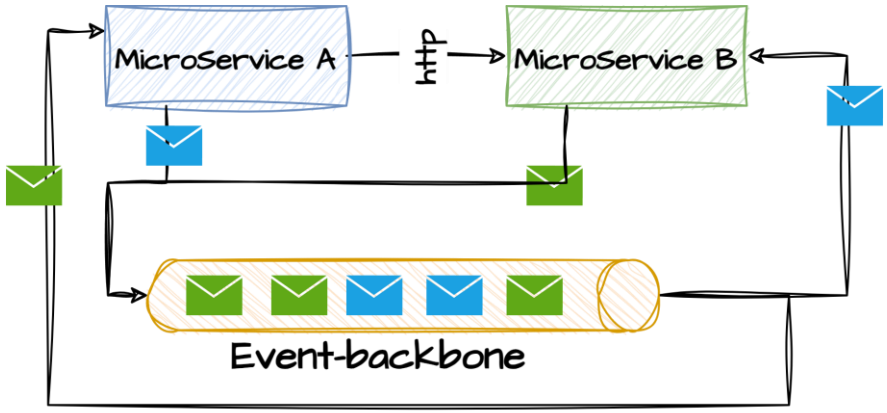
Event backbone (in-transit): No central database of the domain entities exist. Instead, changes are communicated and exchanged as streams of messages between services. Consumers of the entity messages may build up local state that project the entities in ways that are optimized for that consumer. Each entity is owned by a service. The owning service may expose an API that allows other services to query the entity.



Event backbone (EB)

Lund&Bendsen
developing developers

An event backbone, often referred to as an event bus or an event broker, is a core component of a distributed message driven architecture. It serves as the central infrastructure that facilitates the flow of events (i.e., discrete pieces of information about something that happened) within a software system or across multiple systems. The primary purpose of an event backbone is to ensure the reliable and efficient communication of events between event producers and event consumers. The event-backbone may offer additional services such as pub/sub, durability, replay of messages, priority queues and advanced routing.



kafka

Rabbit MQ

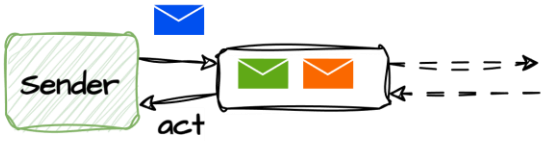
amazon Kinesis

Cloud DataFlow

Amazon SNS

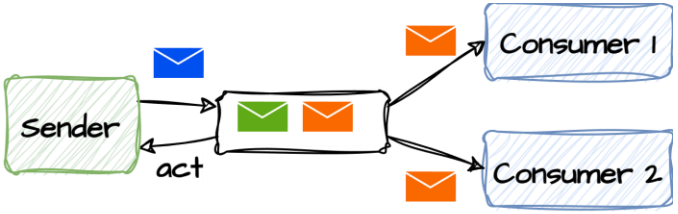
Why use an Event Backbone?

Lund&Bendsen
developing developers



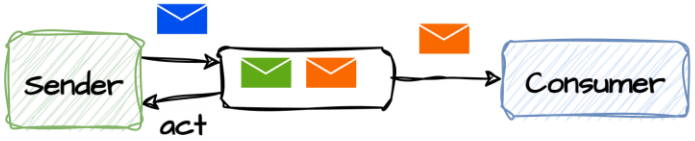
Reduce knowledge of systems

Senders and consumers are decoupled.
They don't know about each other



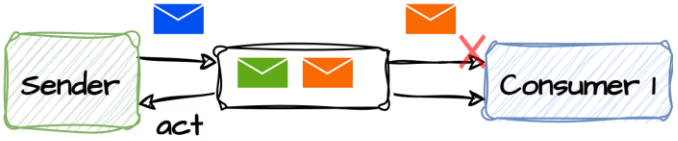
Parallel processing

With pub/sub notify multiple consumers
Can scale to run processes in parallel



Reduce pressure off consumers

Protect consumers from being overloaded
by buffering up messages



Prevent message being lost

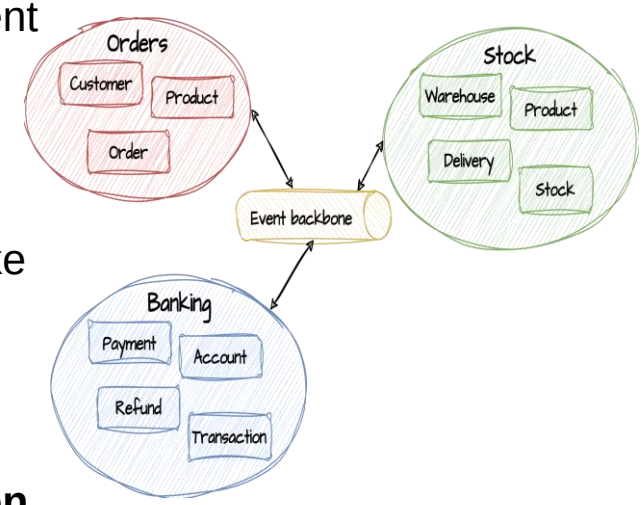
Messages can be reprocced after failures

DDD and Bounded contexts

Lund&Bendsen
developing developers

DDD (domain driven development) is about technical- and domain model alignment so that domain experts and technical team members working together can communicate about the model in an unambiguous way.

In DDD, **bounded contexts** help to establish clear boundaries between different parts of the system by grouping related domain **entities**. This allows for better separation of work as smaller teams can focus on sub domains. Inside a boundary, the model must be **consistent** and **coherent**, meaning that the entities must be inter-related and dependent on each other. Messages sent between services inside a bounded context is referred to as **domain events**.



The same entity name may occur within different bounded contexts (like Product in Orders and Stock).

A public language (model) needs to be specified when communicating between bounded contexts.

Messages sent between bounded context are referred to as **integration events**. Sometimes translations to/from integration events must happen before sending to the backbone or receiving from the backbone if the producing or consuming bounded context is not aware of integration event language.

Messages between bounded context

Lund&Bendsen
developing developers

Conformist

Don't translate message, consume as is

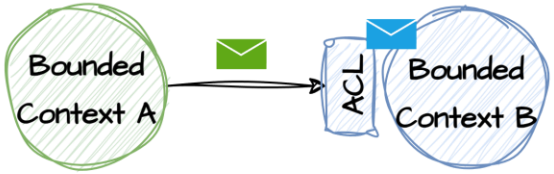
Open-Host Service

Producer does not conform to contract

Messages get translated prior to publishing

Enrich

Somtimes you want consumers getting more info vs producers providing it



Anti-corruption

Consumer does not conform to contract

Messages get translated before consumption

EB challenges

Lund&Bendsen
developing developers

What challenges do you see?

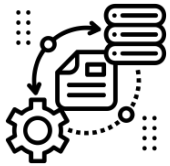
When building a distributed and loosely coupled architecture that is based on an event backbone there are some common challenges that you need to be aware of.

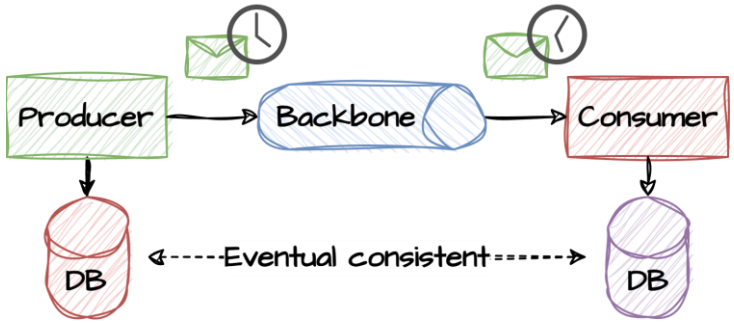
Data (eventual) Consistency:

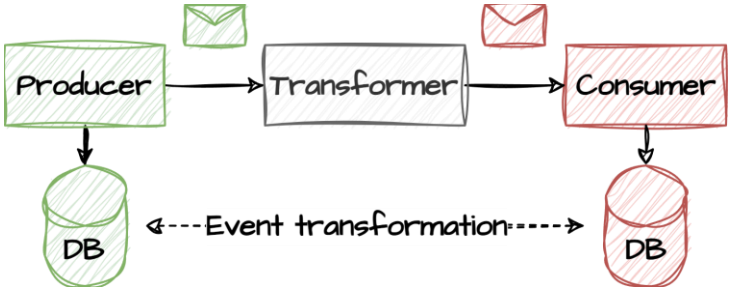
Eventual consistency is a consistency model used in distributed computing to achieve high availability. If no new updates are made to a given data item, eventually all accesses to that item will return the last updated value. Common challenges is ‘read your own write’, which states that the system guarantees that, once an item has been updated, any attempt to read the record by the same client will return the updated value

Data Transformation and Validation:

Microservices often have different data requirements and formats. Ensuring that data is properly transformed and validated as it moves between services is essential to prevent data-related issues.







EB challenges

Lund&Bendsen
developing developers

Data Ownership:

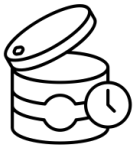
Determining which microservice is the authoritative source of certain data can be challenging. This is especially true in cases where multiple microservices may have their own copies of the same data.

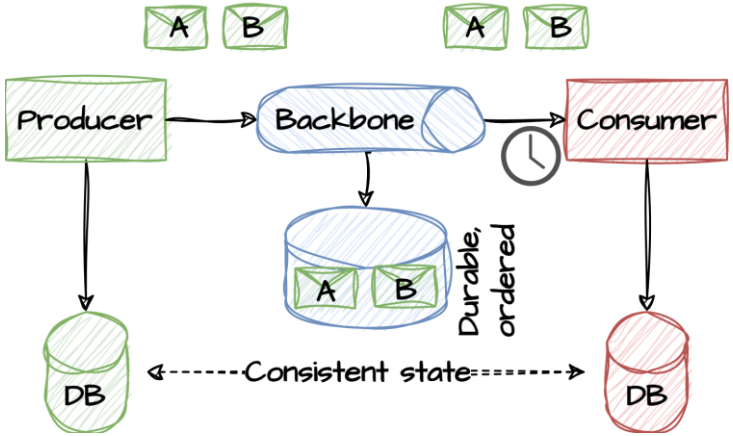
Message Ordering:

Ensuring the correct order of messages or events is vital in many microservices applications. Guaranteeing that events are processed in the right sequence can be challenging, especially when dealing with distributed messaging systems.

Message Durability:

Guaranteeing the durability of messages or events, even in the face of failures, is crucial. This may require using durable message queues or event stores.



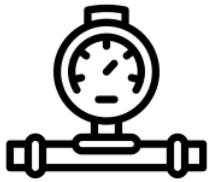


EB challenges

Lund&Bendsen
developing developers

Back Pressure:

When data is flowing into a microservice faster than it can process, it may use back-pressure to make the producing component / EB slow down. Managing back pressure and ensuring that services can handle spikes in traffic is a challenge.

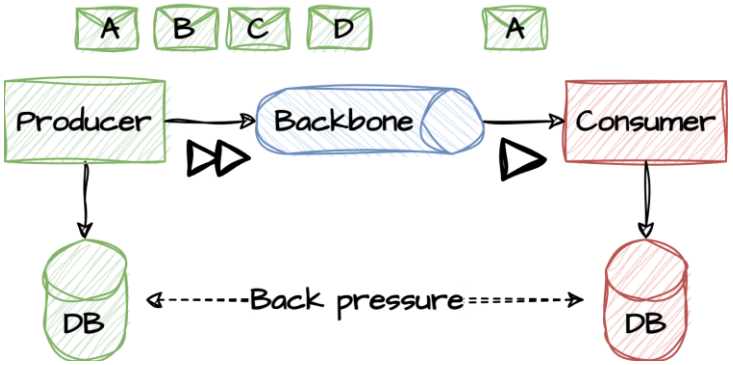


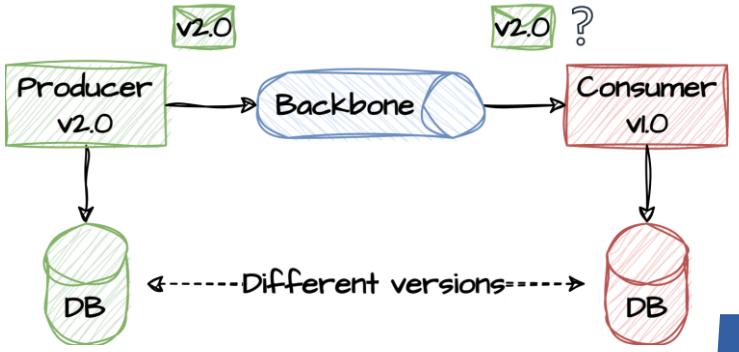
Data Privacy and Security:

Ensuring data privacy and security is a challenge when data flows between services. Implementing proper access controls, encryption, and data masking is crucial to protect sensitive information.

Data Versioning:

As microservices evolve independently, data schemas may change. Managing data versioning and handling backward compatibility is important to avoid breaking existing dataflows.





EB challenges

Lund&Bendsen
developing developers

Error Handling:

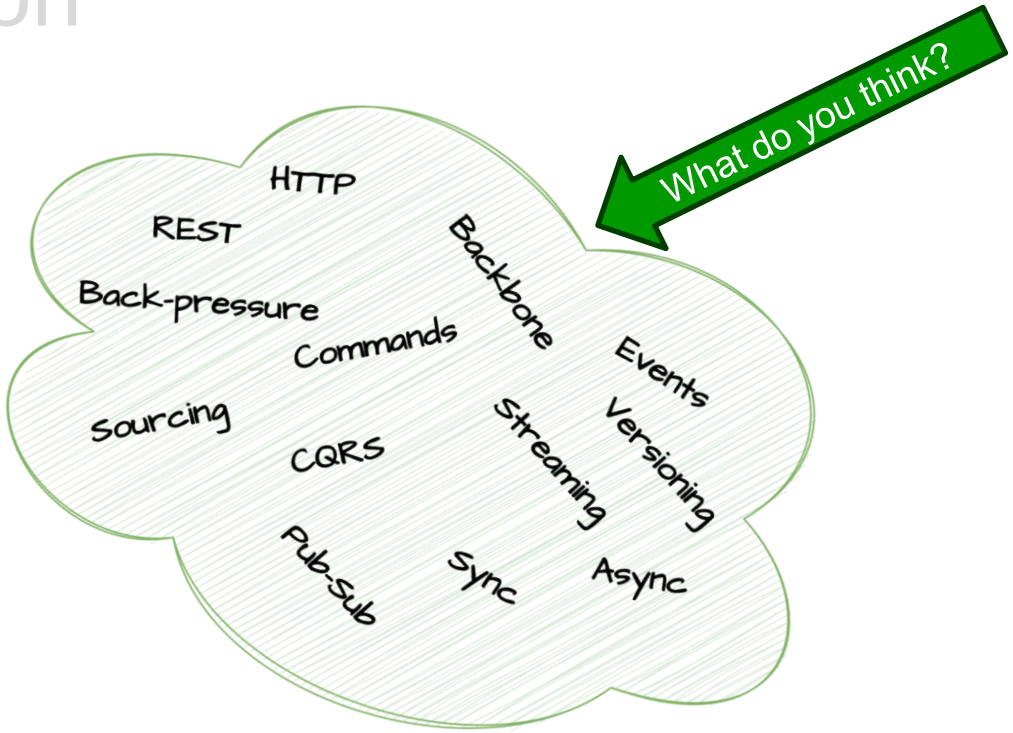
Handling errors that occur during dataflow is essential. Microservices should be designed to handle failures gracefully, including retries, dead-letter queues, and error notifications.

Monitoring and Debugging:

Understanding the flow of data between microservices and diagnosing issues can be difficult. Comprehensive monitoring and logging are necessary to trace the flow of data and troubleshoot problems.

Lund&Bendsen
developing developers

- Software architecture
- Communication
- EDA



EDA overview

Lund&Bendsen
developing developers

Over the past few years, there has been a movement from focusing on data at-rest (monioliths and service-oriented architecture with API services backed by databases) to focusing on data in-transit (event-driven architecture)

Event-driven architecture (EDA) is a software design pattern that enables an organization to detect “events” or important business moments (such as a transaction, site visit etc) and act on them in real time or near real time. This pattern replaces the traditional “request/response” architecture where services would have to wait for a reply before moving on to the next task. The synchronous communication approach does not scale well and introduces blocking code and increases risks of failures.

In EDA, events are emitted by an event emitter and broadcasted to any number of listeners through an event backbone. The event emitter has no knowledge of whom may consume the event. Event listeners are also completely decoupled from the event emitters and may emit new events onto the backbone as a response to the event. The decoupling of producers and consumers adds several benefits to the system design. The decoupling allows producers and consumers to produce/consume and different speed rates, allow new consumers to be added seamlessly and replay historic events.

The downside is added complexity when introducing an event pipeline in between components, observability issues and versioning of data.

Events, Commands and Queries

Lund&Bendsen
developing developers

Events

Something happened (provide identity + context)

Zero or many consumers (completely decoupled)

Single producer (owner of the event and its data)

Owned by producer

Commands

Invoke behavior (target a specific recipient)

Single consumer (consumer executes the command)

Multiple senders (Anyone can send a command)

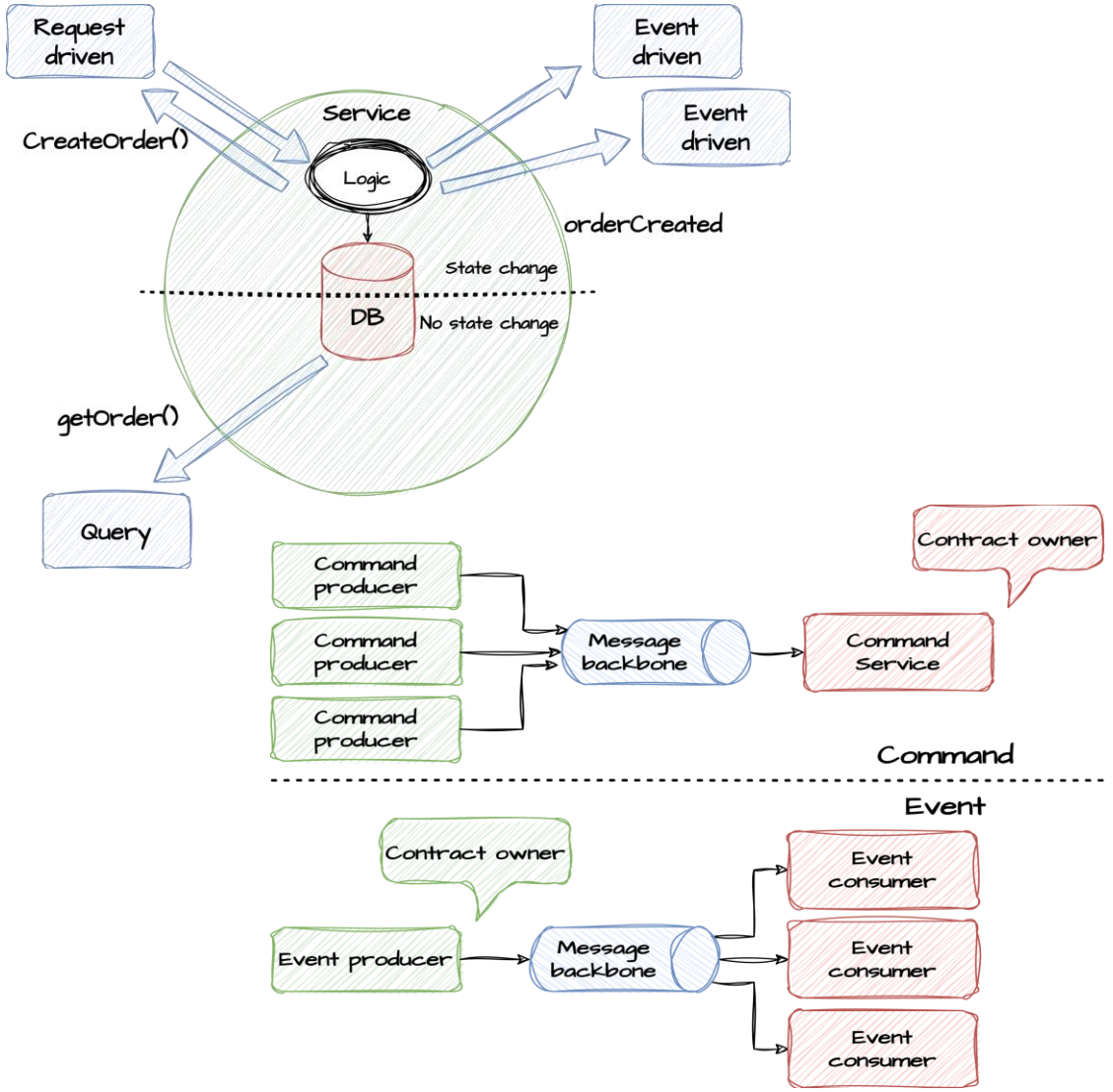
Owned by consumer

Queries

Peer-to-peer. Typically in the form of HTTP calls

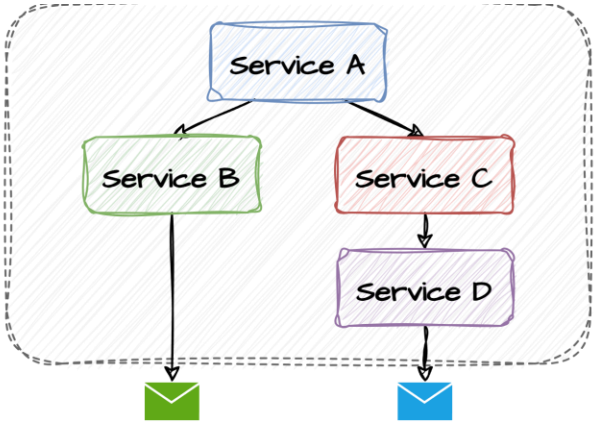
No side-effects (state changes) in the service.

Read-only and typically backed by a data-store



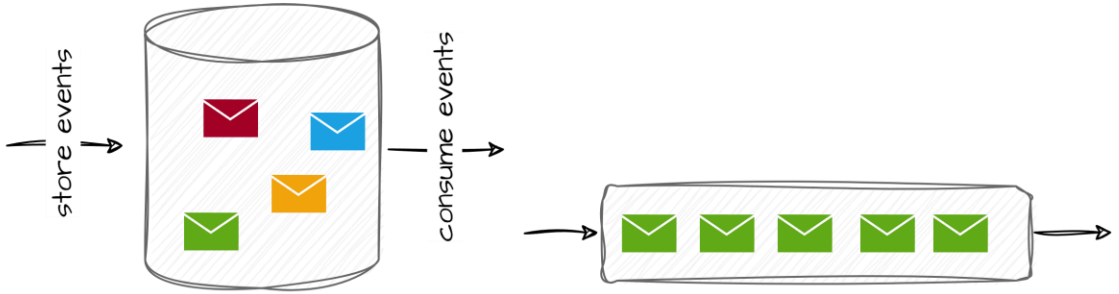
Event communication patterns

Lund&Bendsen
developing developers



Process Manager

Central unit that
orchestrates communication



Event Sourcing

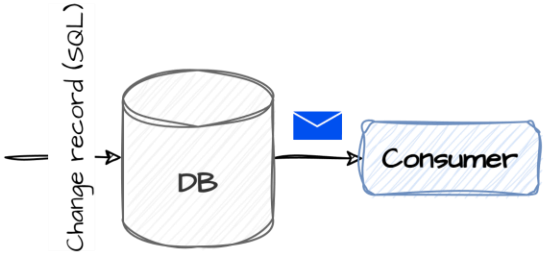
Store event and build
current state using events

Event streaming

Unbounded stream of events

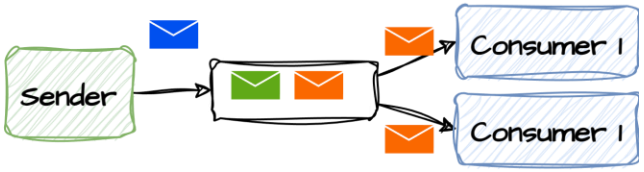
Point-to-point

Sender put message on queue
Receiver reads from queue and deletes message



Change data capture

Consume changes directly from DB



Pub/Sub

Publish same message to
many consumers. Messages preserved

Events and de-coupling

Lund&Bendsen
developing developers

Events is about something that has happened in the past. Consumers often need to react to those changes to get the most recent state of the entity behind that event. How do we do that in a distributed system without introducing coupeling? Event may carry more or less information which is a trade-off between high-coupling due to additional lookups (consistency) or risk of carrying stale data but with no additional lookup (availability).

Notification event

Minimal payload, expect consumers to fetch data at the owner

Small risk of breaking data contract

Low coupling

Delta event

Only communicate changes (deltas)

Medium risk of breaking data contract

Medium coupling

Event-carried state transfer

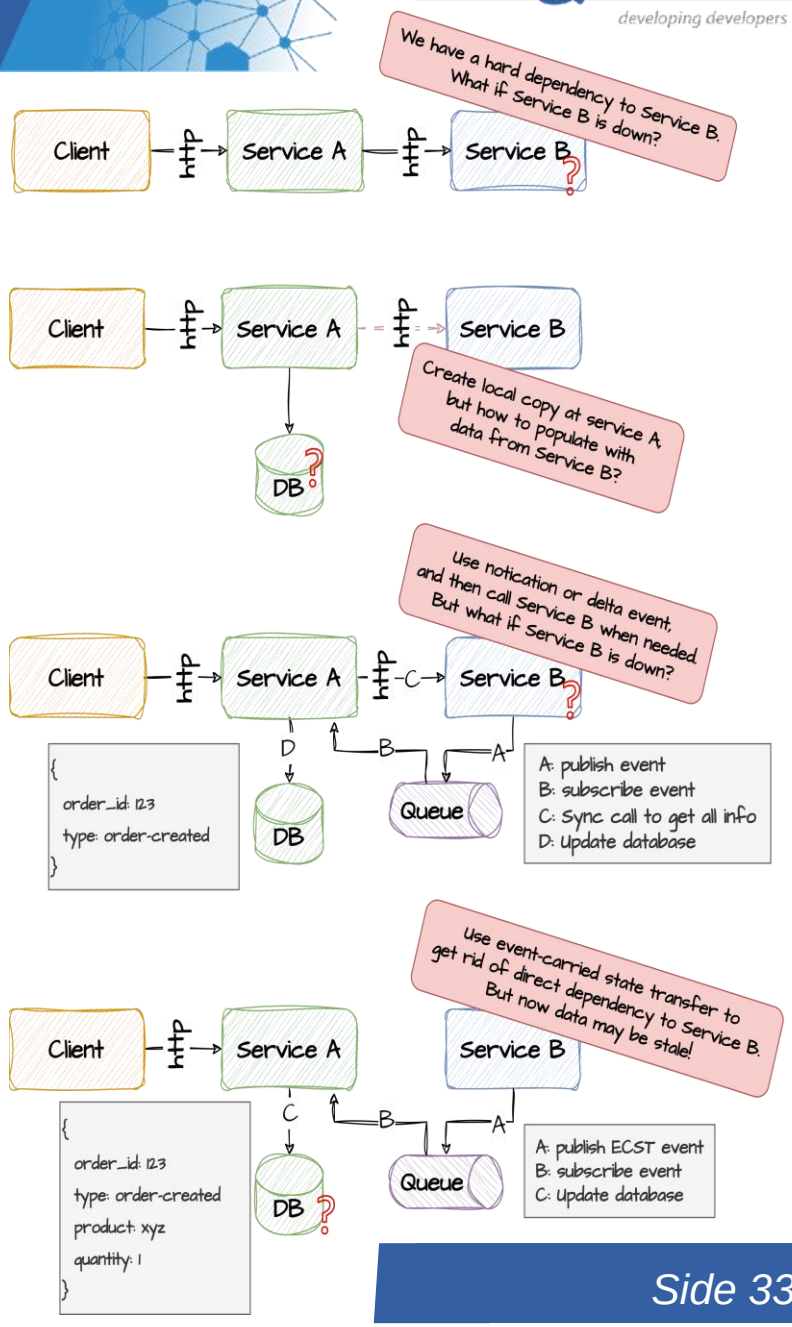
Rich events so no need to fetch additional data fro owner.

Rick of data becomming stale and breaking data contracts

Low coupling

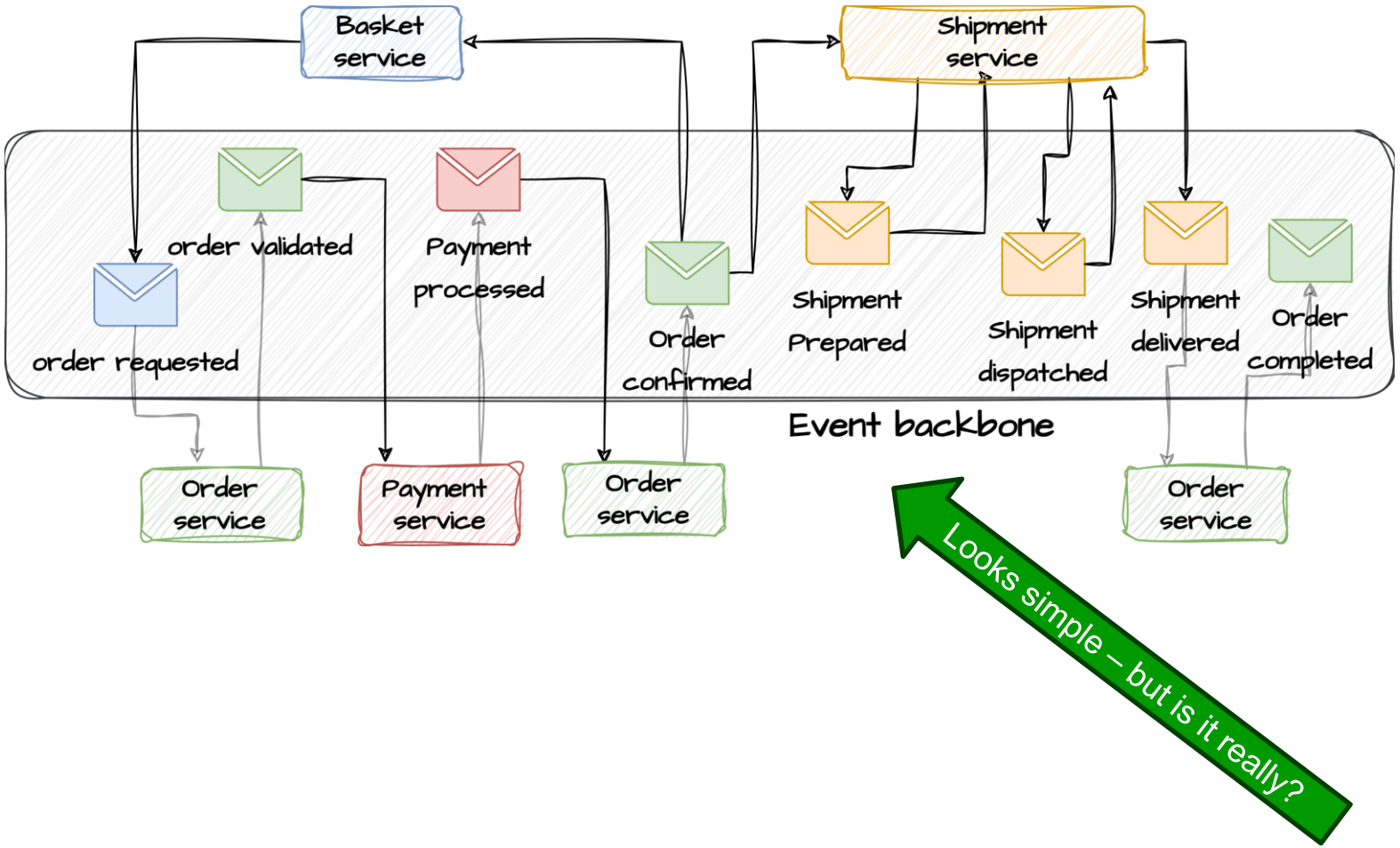
Internal and External events (for DDD)

- Used when there are multiple bounded contexts (domains)
- Internal (domain) events are used inside the domain whereas external (integration) events are used between domains

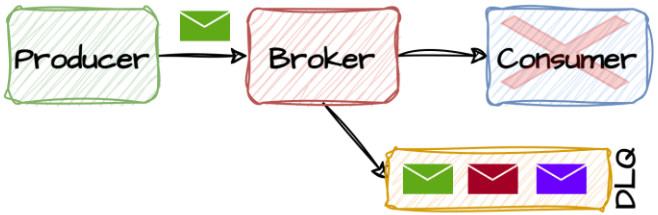


EDA at a larger scale

Lund&Bendsen
developing developers



Event delivery failures

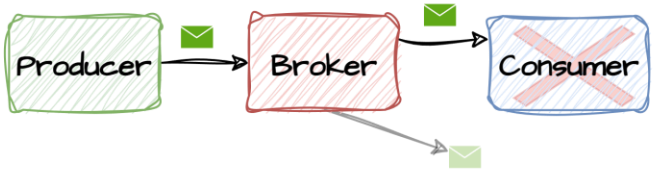


Dead-letter queues

Event fails to reach target, message can be sent to special queue for retry/inspection

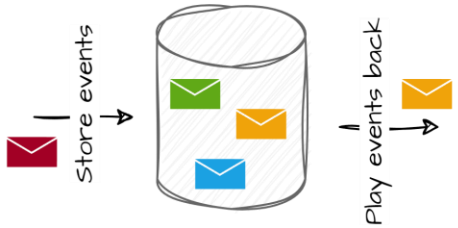
Redrive policies

Sometimes the broker may retry delivery for you based on certain retry criterias



Dropping events

Sometimes it makes sence to just drop events if they cannot reach target system



Archive / replay

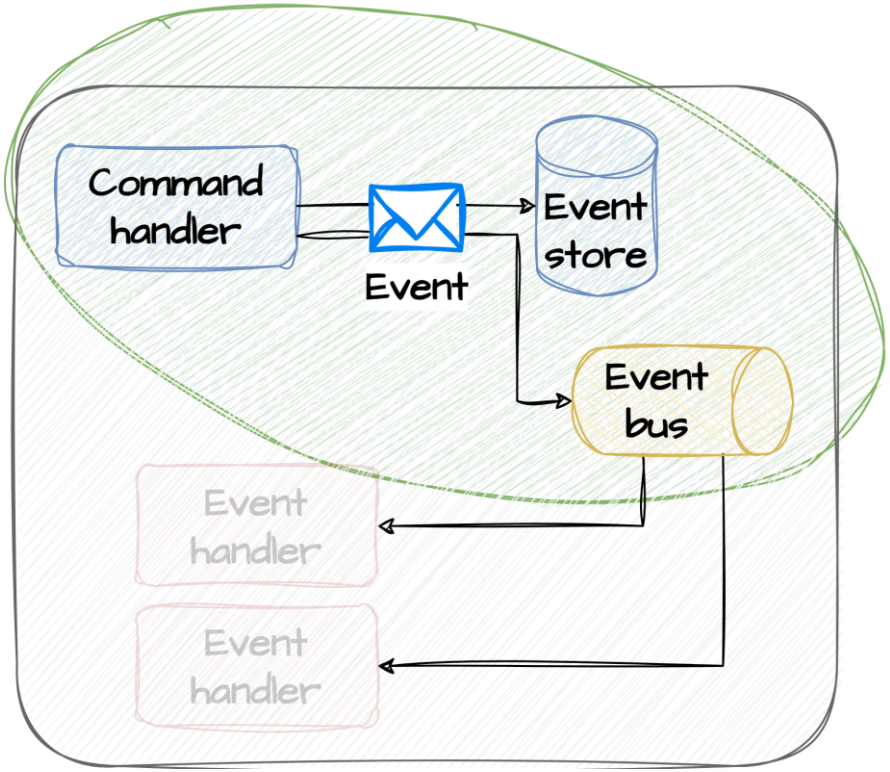
Store events and retry later
Make sure consumers are idempotent

Event Sourcing

Lund&Bendsen
developing developers

Event sourcing is a way of modeling and persisting event data as a series of **immutable** and **ordered** events rather than focusing on the current state of the entity behind the event.

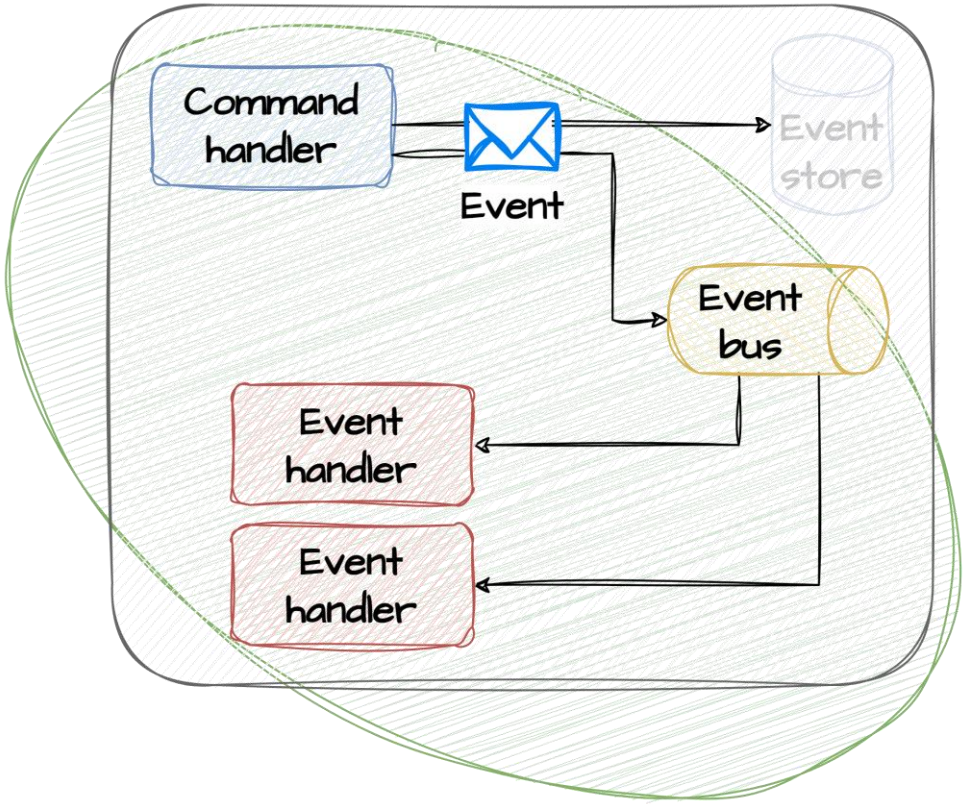
Events are stored in an event store which is local to the service (bounded context) that it lives in. Events in the event store can be replayed and thereby used for state reconstruction or used for auditing or debugging purposes.



Event Streaming

Lund&Bendsen
developing developers

Event streaming is an **ordered**, **unbounded** sequence of **immutable** events flowing from producers to consumers through an event backbone with an **at-least-once** delivery guarantee. Whenever a service or bounded context makes changes to an entity that I owns, this can be published as an event on an event backbone. The event may carry context depending on the design. Event streaming uses a pub-sub approach to enable decoupled communication between systems. Consumers subscribe to a topic/channel on the event pipeline, and producers produce events to these topics/channels. Services may act as both consumers and producers of events. The pub-sub design decouples the producers and consumers, making it easier to scale each part of the system individually.



Event Projection - CQRS

Lund&Bendsen
developing developers

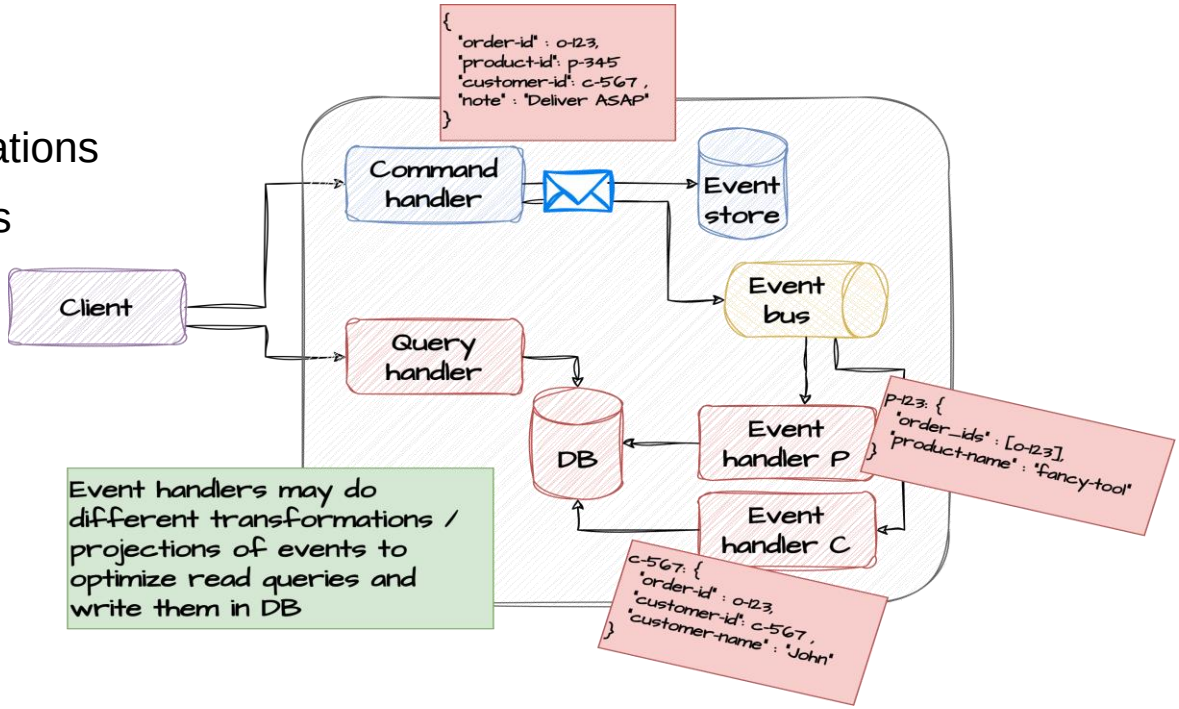
CQRS, which stands for Command Query Responsibility Segregation, is an architectural pattern used in software development to separate the concerns of handling commands (write operations) and queries (read operations) within an application. By separating read/write operations this way you can optimize each part independently. Read operations typically have different forms and transformers (event handlers) can project the same write operations into different projections that are read optimized.

Advantages:

- Scalability – Separate read/write operations
- Flexibility – Create different projections

Disadvantages:

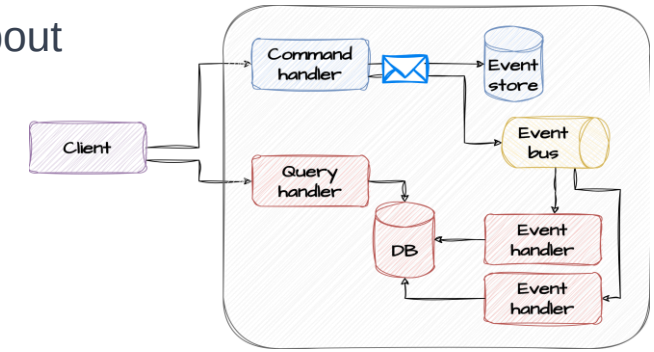
- Complexity in code
- Eventual consistency



EDA concepts

Lund&Bendsen
developing developers

Event-Driven Architecture (EDA) is an architectural pattern that focuses on the flow of events (i.e., discrete pieces of information about something that happened) within a software system. It promotes decoupled, asynchronous communication between different components, enabling responsiveness, scalability, and flexibility.



Events:

Events are the core building blocks of an EDA. They represent significant occurrences or changes in the system or the environment.

Events are typically represented as structured data (e.g., JSON, XML) and include information about what happened and, optionally, any relevant metadata.

Event Producers:

Event producers are components or systems that generate and emit events. These can be various parts of the system, such as microservices, applications, sensors, or external services.

Producers publish events to event channels or brokers.

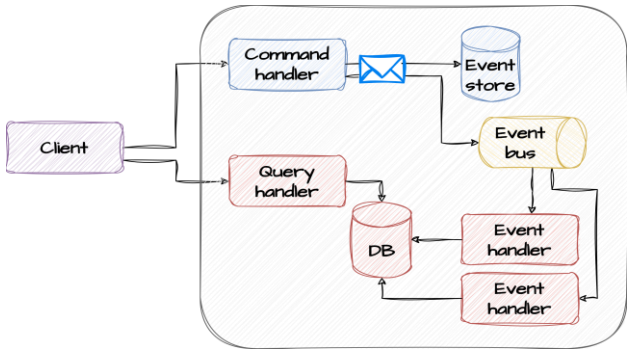
EDA concepts

Lund&Bendsen
developing developers

Event channels or Brokers:

Event channels, also known as event brokers, are intermediaries that facilitate the communication of events between producers and consumers.

They ensure that events are delivered reliably, maintain event order, and provide various features such as topic-based routing and event storage.



Publish-Subscribe Model:

EDA typically follows a publish-subscribe model. Event producers publish events to specific event channels or topics, and event consumers subscribe to those channels or topics.

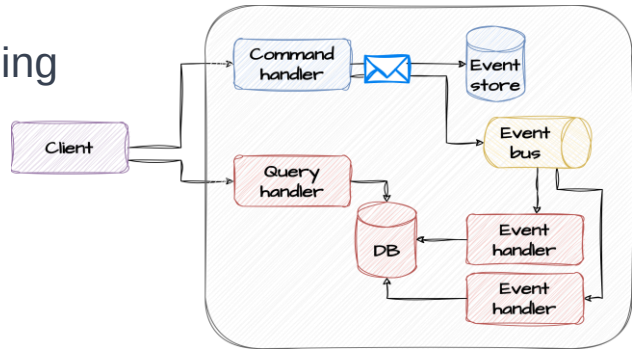
This decouples producers and consumers, allowing for a dynamic and flexible communication pattern.

Lund&Bendsen
developing developers

Event Consumers:

Event consumers are components responsible for processing incoming events. They contain the logic to react to events and trigger appropriate actions.

Event consumers may include business logic, data processing, and database updates, among other tasks.



Event Routing and Transformation:

In complex systems, event routing may be necessary to direct events to the appropriate consumers. Event routers or brokers often handle this task.

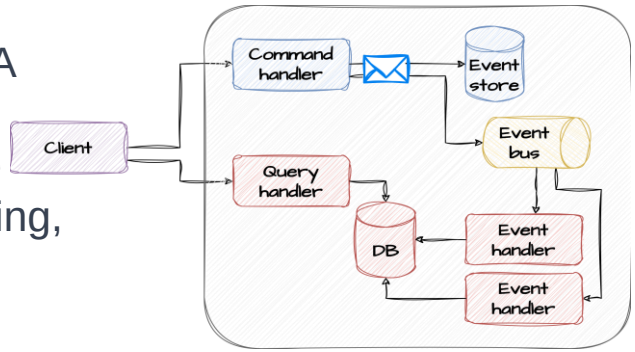
Events may also need to be transformed to match the requirements of different consumers.

EDA concepts

Lund&Bendsen
developing developers

Event Sourcing:

Event Sourcing is about persisting state. Its a technique used in EDA where the state of a system is determined by a sequence of events rather than the current state. Each event represents a change to the system's state. Event Sourcing can be valuable for auditing, versioning, and maintaining a full history of system changes



Monitoring and Logging:

Comprehensive monitoring and logging are essential in EDA to track the flow of events, diagnose issues, and ensure system reliability.

Tools for event monitoring and centralized logging are often used to gain visibility into event processing.

Error Handling and Retry Mechanisms:

EDA systems should implement robust error handling and retry mechanisms to handle transient failures and ensure reliable event delivery.

Dead-letter queues or error queues may be used to capture events that cannot be processed immediately.