sidion

DATENBANKEN? WO WIR HINFAHREN BRAUCHEN WIR KEINE DATENBANKEN.

Nicolai Mainiero
sidion

Zuhören. Analysieren. Beraten.

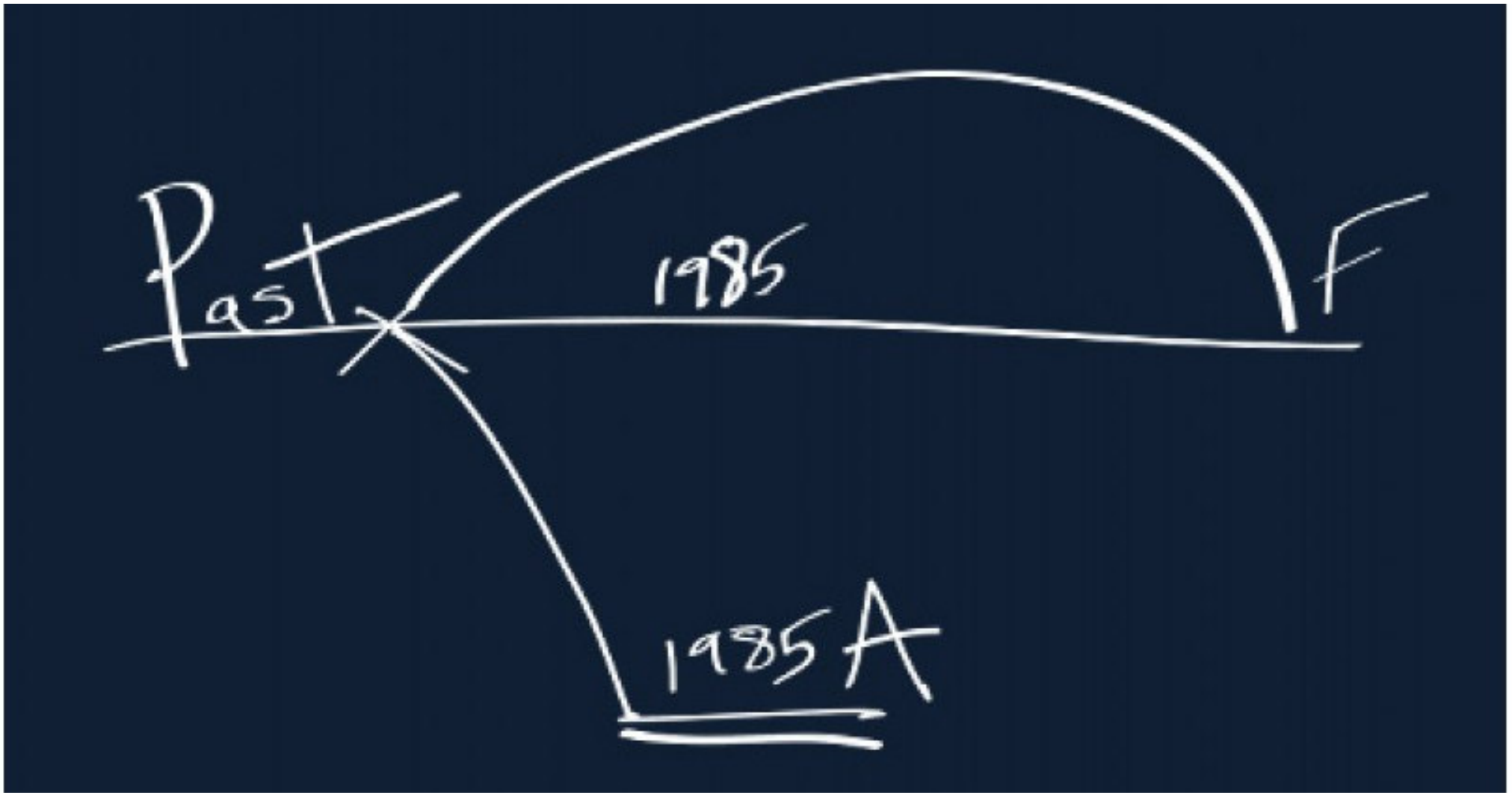
Über mich

- Diplom-Informatiker.
- Mehr als 8 Jahre Erfahrung in der Softwareentwicklung.
- Seit mehr als 3 Jahren bei sidion.

Über sidion

- Inhabergeführtes Unternehmen seit 1992.
- Ca. 130 Mitarbeiter.
- Standorte: Stuttgart, Frankfurt.
- Branchen: Automotive OEMs, Banken und Finanzwesen, Logistik, Öffentlicher Dienst.

Die Vergangenheit ändern um die Zukunft zu beeinflussen



sidion

Agenda

1. CQRS und Event Sourcing
2. Vor- und Nachteile
3. Frameworks
4. Ausblick

sidion

CQRS und Event Sourcing

Ein einfaches Beispiel: Bankkonto

Das Bankkonto ist ein einfaches Beispiel für ein System, dass einzelne Ereignisse erfasst und daraus den aktuellen Zustand des Kontos ableitet.

Datum	Verwendungszweck	Wert	Saldo
26.10.1985	Kontoeröffnung	10.000,00	10.000,00
21.11.2015	Sportalmanach	-15,95	9984,05
22.11.2015	Überweisung an Needles	-500,00	9484,05

Ein einfaches Beispiel: Bankkonto

- Jede Transaktion wird gespeichert
- Der aktuellen Zustand ist die Aggregation aller Transaktion
- Den aktuellen Zustand kann man wieder herstellen, indem alle Transaktionen erneut aggregiert werden
- Die Transaktionen sind gleichzeitig ein Audit Log
- Der Zustand (Saldo) kann für jeden beliebigen Zeitpunkt bestimmt werden

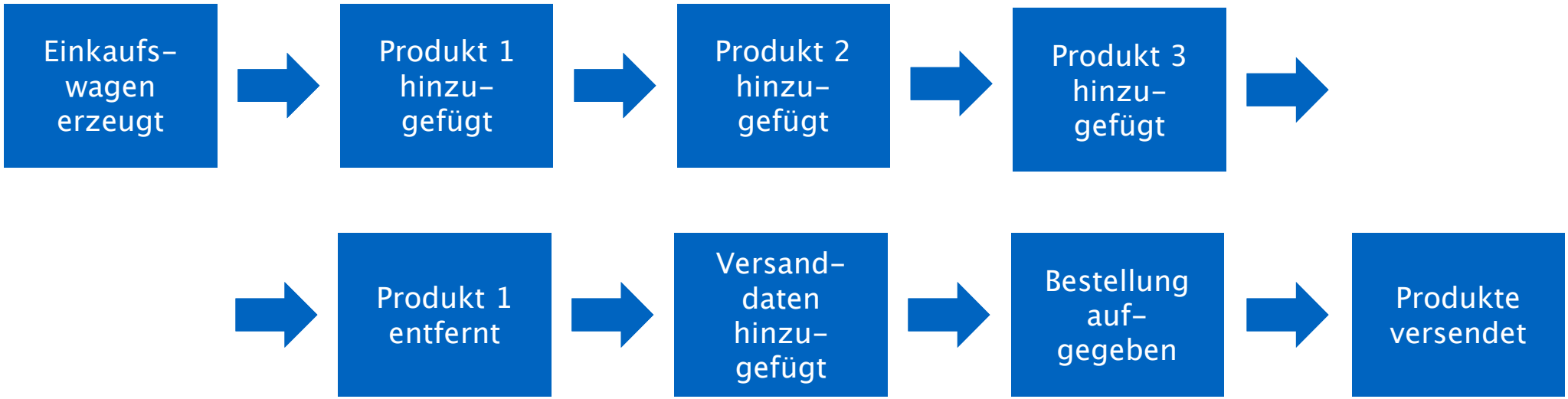
Was ist ein Event?

Etwas das in der Vergangenheit passiert ist.

- Events sollen mit Verben repräsentiert werden
  - CustomerRelocated, CargoShipped oder InventoryLossageRecorded
- Enthalten weitere Daten
- Sind Immutable
- Sind Domänenspezifisch
- Sind Wiederholbar

Event Sourcing – Ein Strom von Events

- Events werden gespeichert, z.B. Append-Only Speicher
- Event beschreibt die Änderung zwischen zwei Zuständen
- Events werden nicht gelöscht



Events auswerten

Problem

- Den aktuellen Zustand ermitteln ohne alle Events betrachten zu müssen
- Events mit weiteren Daten verknüpfen
- Abfragen auf den Events ausführen

Lösung

→ **C**ommand **Q**uery **R**esponsibility **S**egregation (CQRS)

Was ist Command Query Responsibility Segregation (CQRS)?

- **Ursprung ist das Konzept der Command Query Separation**

  It states that every method should either be a command that performs an action, or a query that returns data to the caller, but not both. In other words, Asking a question should not change the answer. More formally, methods should return a value only if they are referentially transparent and hence possess no side effects.

  Bertrand Meyer – 2014

- **Command Query Responsibility Segregation**
  - Lange als Erweiterung von CQS betrachtet
  - Mittlerweile jedoch als eigenes Pattern akzeptiert

- **CQRS teilt Commands und Queries in zwei getrennte Objekte**

Command und Query aufteilen

```
public interface CustomerService {

  void makeCustomerPreferred(int id);

  Customer getCustomer(int id);

  Set<Customer> getCustomersWithName(String
name);

  Set<Customer> getPreferredCustomers();

  void changeCustomerLocale(int id, Locale
locale);

  void createCustomer(Customer c);

  void editCustomerDetails(CustomerDetails
cd);
}
```

Command

```
public interface CustomerWriteService {

  void makeCustomerPreferred(int id);

  void changeCustomerLocale(int id, Locale
locale);

  void createCustomer(Customer c);

  void editCustomerDetails(CustomerDetails
cd);
}
```

Query

```
public interface CustomerReadService {

  Customer getCustomer(int id);

  Set<Customer> getCustomersWithName(String
name);

  Set<Customer> getPreferredCustomers();

}
```

Gründe für die Aufteilung in Commands und Querys

Anforderung	Command	Query
Konsistenz	Leichter mit konsistenten Daten zu arbeiten	Den meisten Systemen reicht es wenn die Daten irgendwann konsistent sind
Speicherung	Am besten normalisiertes Datenmodell	Am besten denormalisiertes Datenmodell
Skalierbarkeit	Relevant	Sehr wichtig

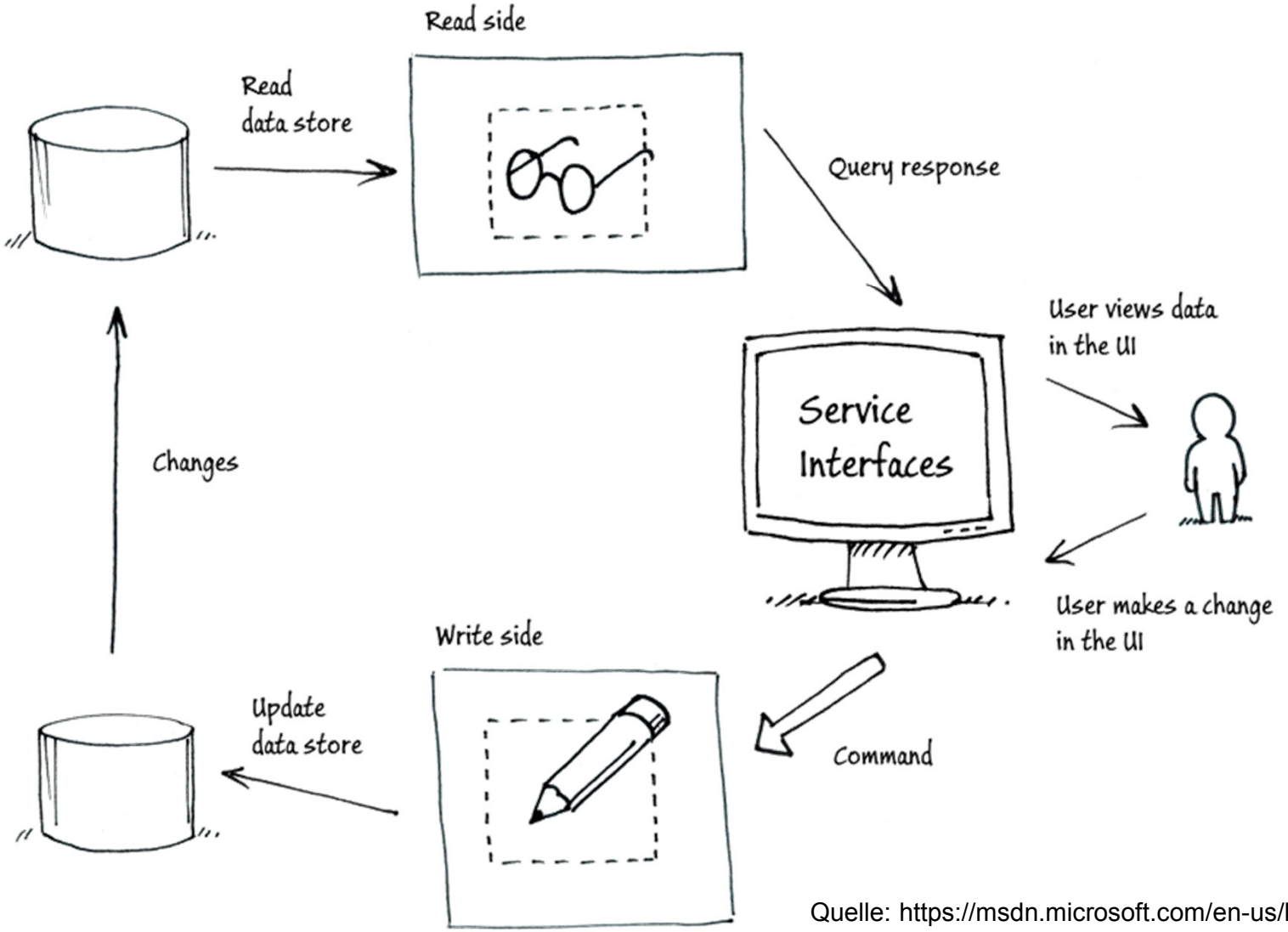
Commands

- Als Imperativ formuliert, z.B. *OrderCart, BookFlight*
- Aufforderung an das System etwas zu tun
- Ermöglichen ein Domain Driven Design
- Reduzieren Komplexität, da lesender Zugriff nicht mehr berücksichtigt werden muss
- Werden von einem Empfänger verarbeitet

Queries

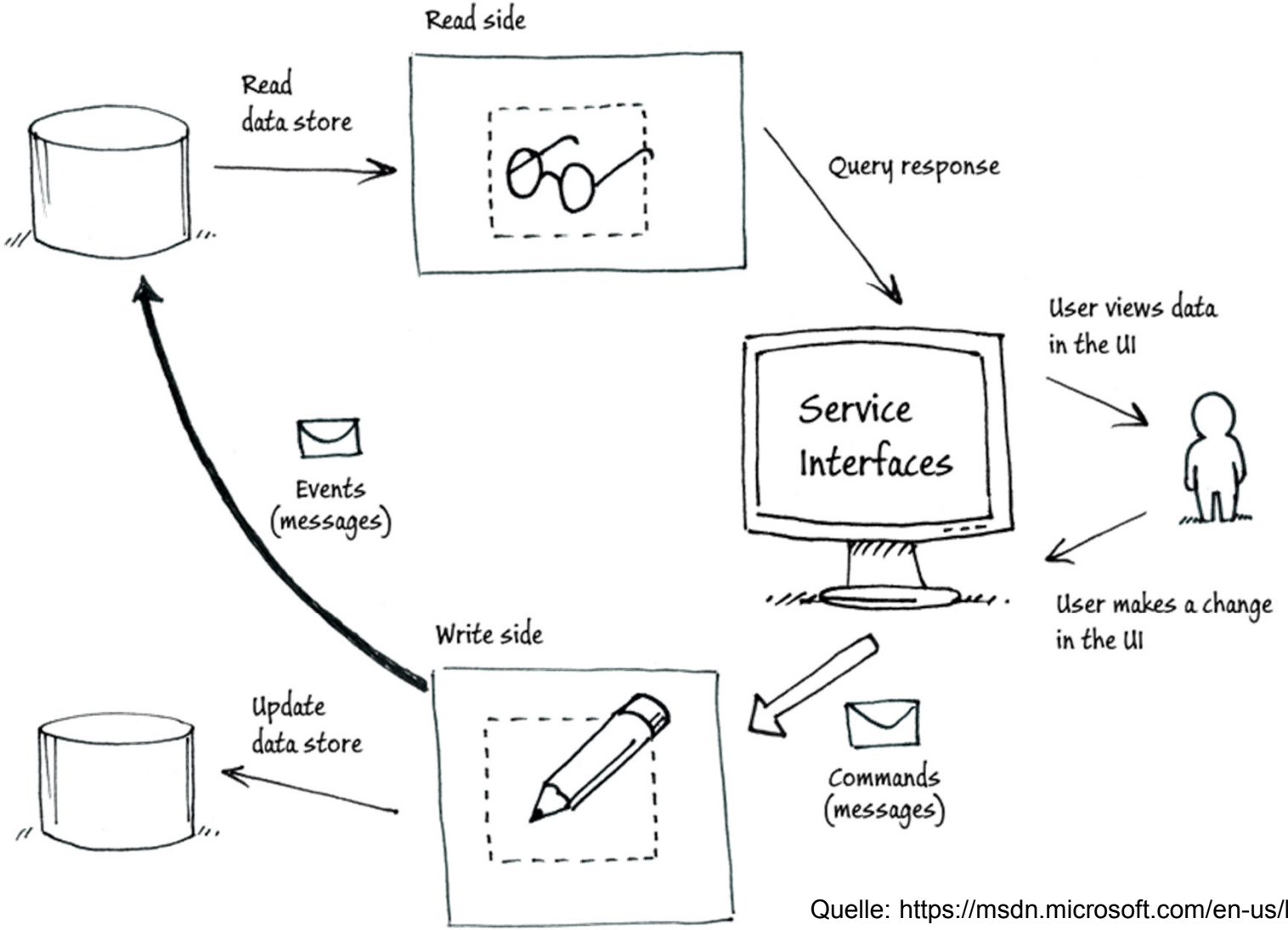
- Anfragen an das System
- Leichtgewichtige Implementierung, die direkt aus dem Datenspeicher liest
- Anfragen sind simpler
- Anfragen können einfacher optimiert werden
- Anfragen sind schneller

Zwei getrennte Pfade



Quelle: https://msdn.microsoft.com/en-us/library/jj591573.aspx

Zwei getrennte Pfade mit Events und Messages



Quelle: https://msdn.microsoft.com/en-us/library/jj591573.aspx

sidion

Vor- und Nachteile

Vorteile

- Hohe Skalierbarkeit
  - Schreib- und Lesezugriffe sind unabhängig voneinander skalierbar
- Ermöglicht Domain Driven Design
- Eignet sich zum Einsatz in Serviceorientierten Architekturen, etwa im Cloud Computing
- Verbesserte Sicherheit durch getrennte Rollen der Schreib- und Leseoperationen
- Simultaner Einsatz verschiedener Versionen derselben Software möglich
  - Beibehaltung von Rückwärtskompatibilität möglich
  - Migration auf neue Version im Live-Betrieb ohne Downtime möglich
- Anpassbarkeit an veränderte Business-Anforderungen

Vorteile

- Entwicklung der einzelnen Bestandteile durch unterschiedliche Teams
- Nachvollziehbarkeit, da alle Änderungen aufgezeichnet werden
- Jederzeit nachträgliche Datenanalyse möglich
- Deterministische Fehleranalyse möglich

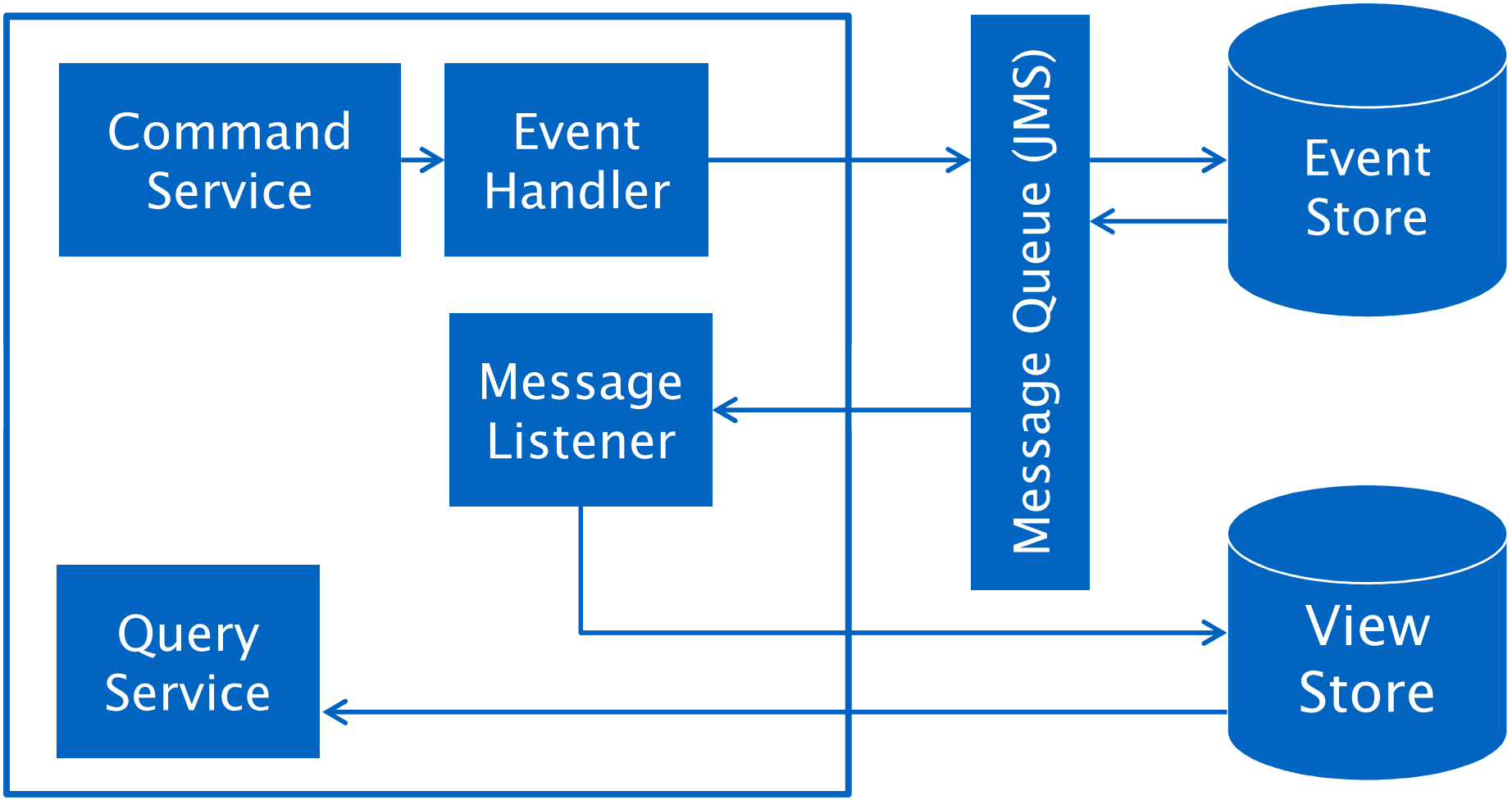
Nachteile

- Hoher Aufwand in der Softwareentwicklung
  - Eignet sich eher für Projekte mit vielen gleichzeitigen Benutzern
  - Wenn Nachvollziehbarkeit wichtig ist
- Benötigt passende Infrastruktur
- Transaktionen sind schwer umzusetzen
- Queries können durchgeführt werden, bevor Commands durchgeführt wurden. Client sieht eventuell veraltete Daten

sidion

Bibliotheken und Frameworks

Java EE


Command Service
Event Handler
Message Queue (JMS)
Event Store
Message Listener
Query Service
View Store

Java EE

- Bestehende Applikation kann migriert werden
- Keine vollständige Neuentwicklung notwendig
- Weiterentwicklung in Richtung Microservice möglich

Akka Persistence

- Inspiriert und offizieller Nachfolger von eventsourced
- Low-Level Framework um Event-Sourcing und CQRS zu implementieren
  - Verarbeitung von Commands
  - Persistierung der Events
  - Persistence Query um Events zu lesen und die Daten für lesende Seite aufzubereiten
- Kann zum Beispiel mit Play! zu einer Web-Anwendung oder Microservice ergänzt werden
- Angenehmer mit Scala als mit Java zu nutzen

eventuate.io

- Software as a Service oder als lokale Installation
- Automatische Events, wenn sich Daten ändern
- Schnelle und Skalierbare Queries durch materialized views
- Integrierte Unterstützung für zeitliche Abfragen
- Bibliothek zum Einbinden in eigenen Code basierend auf RxJava
- Gute Code-Beispiele frei verfügbar

sidion

Ausblick

Ausblick

- Möglichkeit Monolithen aufzubrechen bzw. Legacy-Systeme zu modernisieren
- Flexiblere Optionen zur Skalierung der Anwendung
- Audit-Logs als Zugabe
- Keine Datenbankmigration nötig
- Mögliche Datensynchronisation für Microservices
- Fast Data mit Hilfe von Streamprocessing

sidion

Fragen?

sidion

Heßbrühlstr. 15
70565 Stuttgart

Mergenthalerallee 10-12
65760 Frankfurt-Eschborn

www.sidion.de

Zuhören. Analysieren. Beraten.

sidion

Referenzen

- **A CQRS journey** (https://msdn.microsoft.com/en-us/library/jj554200.aspx)
- **Turning the database inside-out with Apache Samza** (http://www.confluent.io/blog/turning-the-database-inside-out-with-apache-samza/)
- **Building and Deploying Microservices with Event Sourcing, CQRS and Docker** (http://www.infoq.com/presentations/microservices-docker-cqrs)
- **lagom** (http://www.lagomframework.com/documentation/1.0.x/Home.html)
- **Axon Framework** (http://www.axonframework.org/)
- **Akka Persistence** (http://doc.akka.io/docs/akka/2.4.3/java/persistence.html)