Leichtgewichtige Reporting-Architektur, inspiriert von CQRS

Simon Kerler

Lead Software Engineer

JUNGHEINRICH

Agenda

1. Wer bin ich?
2. Einführung in die Domäne
3. Kurzer Abriss zu CQRS
4. Iterative Umsetzung der Architektur
5. Re-Cap
6. Erweiterungsmöglichkeiten
7. Fazit

JUNGHEINRICH

Wer bin ich?
Simon Kerler

- M. Sc. Informatik
- 7+ Jahre Erfahrung in Software-Entwicklung
  - Großkonzern, Startups, Freelance
- Jetzt: Jungheinrich Digital Solutions (JDS)
- Position: Lead Software Engineer

- Fachlicher Fokus u. a. auf
  - Software-Architektur
  - Domain Driven Design
  - Refactoring
  - Performance Optimierung


www.simonkerler.de


github.com/namelessvoid


linkedin.com/in/simon-kerler


simon.kerler@jungheinrich.de
simon_kerler@web.de

Einführung in die Domäne
Jungheinrich Flottenmanagement System

- Jungheinrich: Flurförderfahrzeuge (Gabelstapler) + Intralogistik + Dienstleistung
- JDS: Digitale Lösungen z. B. das Flottenmanagement System

Flottenmanagement System:

- Web- / Cloud-Anwendung
- Holistische Sicht der Kund:innen auf ihre Flotte
- Aktive und passive Features
- Wichtig für heute: Rechnungs-Domäne

Einführung in die Domäne

Flottenmanagement System: Rechnungs-Domäne

Kunde erhält Transparenz über Kosten und Rechnungen:

- Rechnungsverwaltung
- Rechnungsübersicht
- Fahrzeugkosten
- Warenhauskosten
- Filter-Optionen für analytischen Drill-Down
- Festlegen von Budgets zur Kostenkontrolle

Einführung in die Domäne

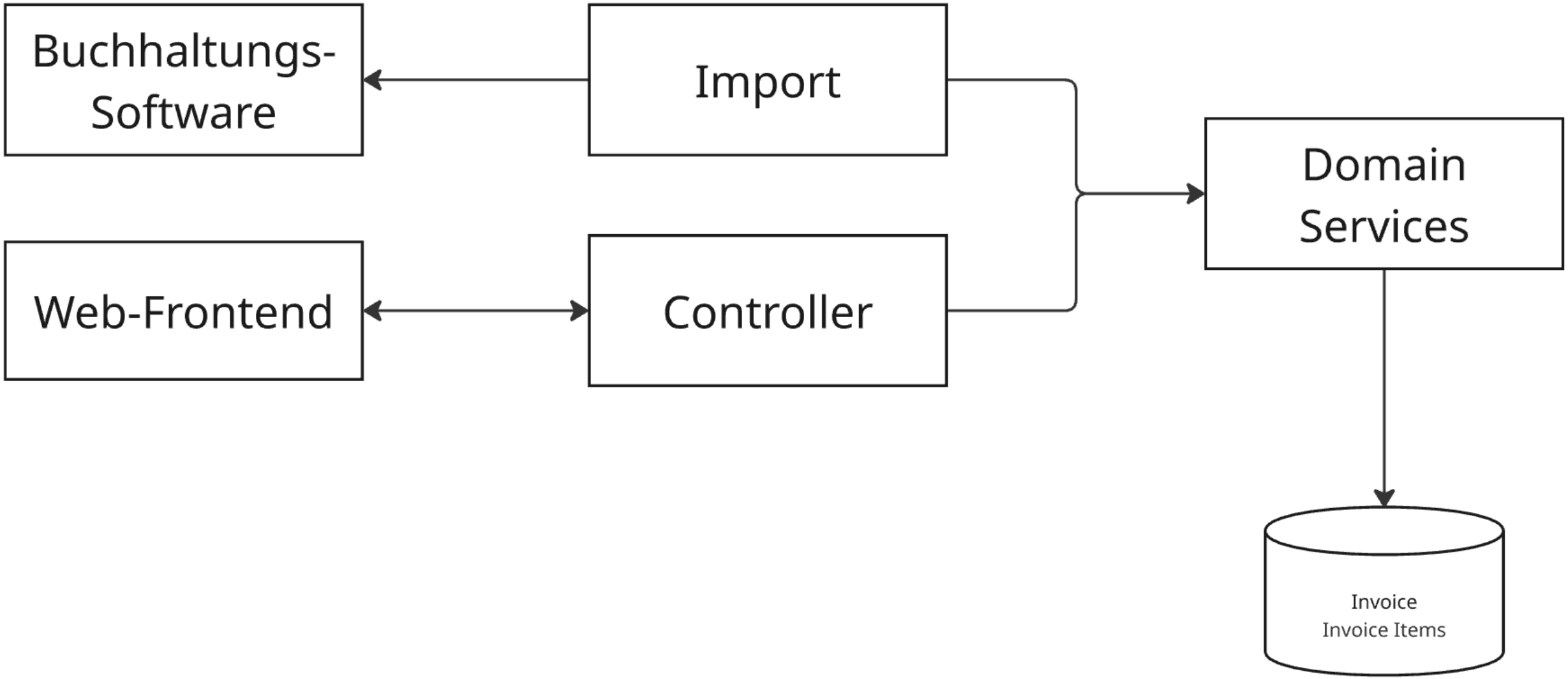
Flottenmanagement System: Rechnungs-Domäne

Tech Stack:

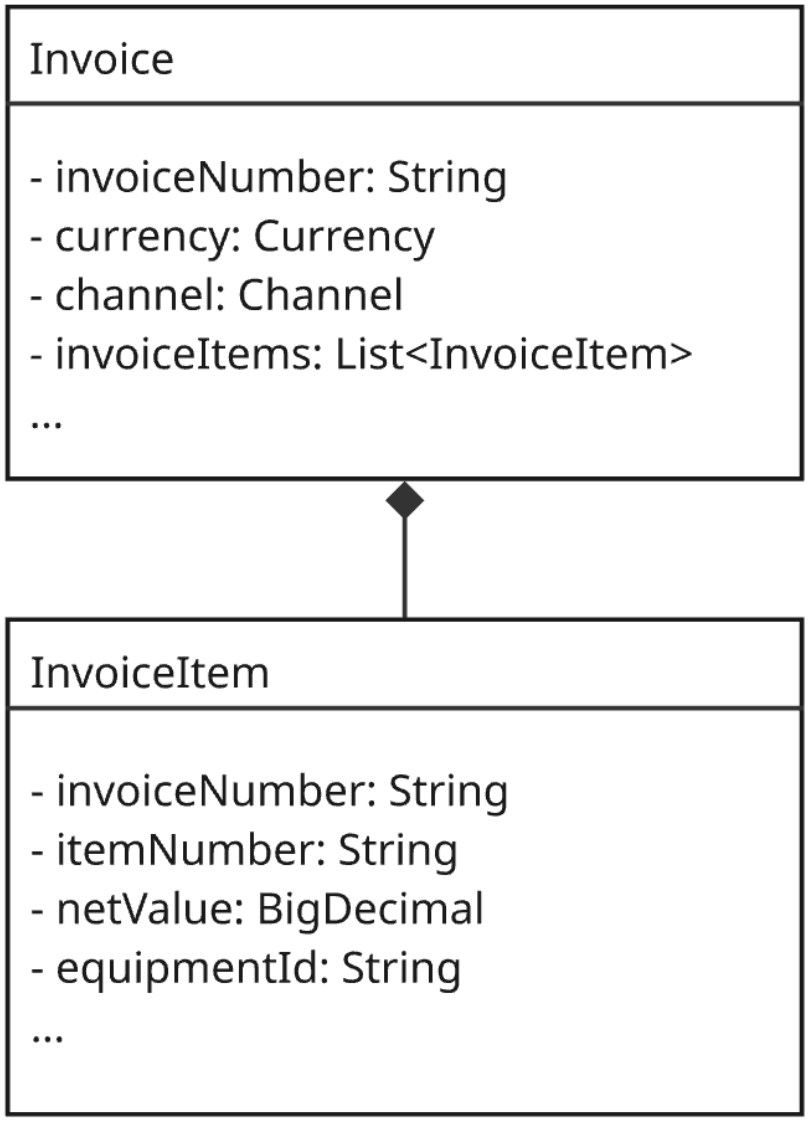
- Frontend: React
- Backend: Spring MVC mit Spring Boot
- Persistenz: Spring Data JPA + Hibernate
- Datenbank: MySQL (historische Gründe)
- Hosting: AWS Fargate

Einführung in die Domäne

High-Level Architektur



Einführung in die Domäne
Datenstrukturen



EquipmentCost

- invoiceNumber: String
- invoiceItemNumber: String

- equipmentId: String
- locationId: String

- currency: Currency
- amount: BigDecimal
- rentalAmount: BigDecimal
- serviceAmount: BigDecimal
...

Einführung in die Domäne
Herausforderungen: Performance

On-the-Fly Transformation

- Normalisiertes Modell anreichern ⇒ Mehrere Joins
- Kostenkategorisierung nicht trivial ⇒ Wird in Java vorgenommen
- Deshalb Aggregationen + Filter auch in Java ⇒ Laden unnötiger Items

⇒Langsam (zweistelliger Sekundenbereich)

Optimierungsversuche

- Datenbank Indizes
- Lazy-Loading
- Code Optimierung

⇒In Summe nicht ausreichend

Einführung in die Domäne
Herausforderungen: Architektur

Keine eindeutigen Zuständigkeiten

- Transformation über mehrere Ebenen verteilt (Domain bis Controller)
- Einzelschritte und Gesamtlogik nicht klar erkennbar
- Gerade bei Support-Anfragen schwer erklärbar

⇒Nicht nachvollziehbare Business-Logik

Einführung in die Domänen

Eine operative und eine analytische Domäne

Rechnungs-Domäne

“Operative” Domäne

- Einsehen von Rechnungen aus der Buchhaltung
- Manuelle Verwaltung von Dritt-Rechnungen

Synchronisation ?

Fahrzeug-Kosten-Domäne

Analytische Reporting Domäne

- Transparenz der Kosten pro Fahrzeug
- Analyse der Kosten pro Standort
- Filteroptionen für analytischen Drill-Down

Einführung in die Domänen

Eine operative und eine analytische Domäne

Rechnungs-Domäne

“Operative” Domäne

- Einsehen von Rechnungen aus der Buchhaltung
- Manuelle Verwaltung von Dritt-Rechnungen

CQRS
!!!!einself

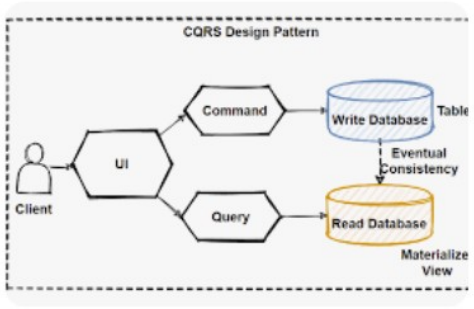
Fahrzeug-Kosten-Domäne

“Analytische” / “Reporting” Domäne

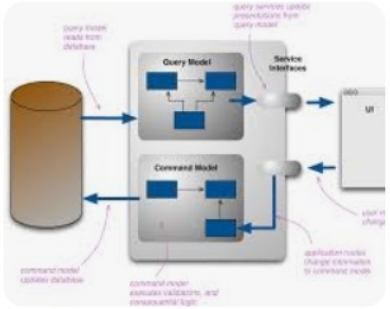
- Transparenz der Kosten pro Fahrzeug
- Analyse der Kosten pro Standort
- Filteroptionen für analytischen Drill-Down

Command Query Responsibility Segregation

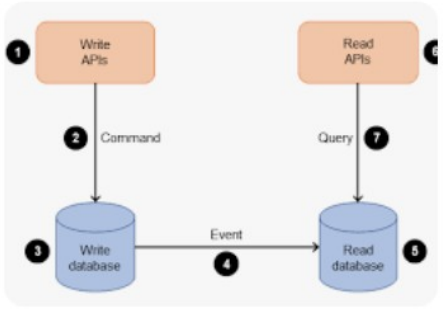
Was die Meisten denken, was es ist



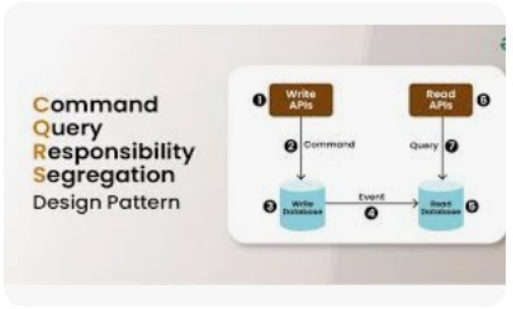
Medium
CQRS Design Pattern in Microservices …



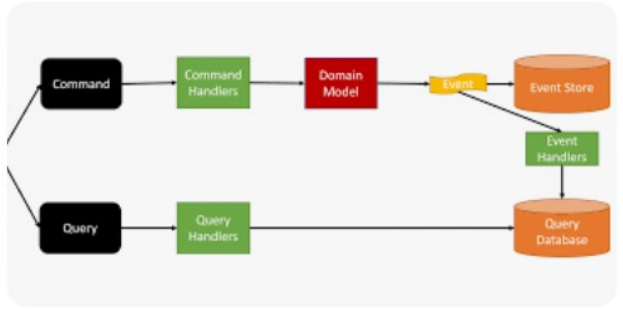
Martin Fowler
CQRS



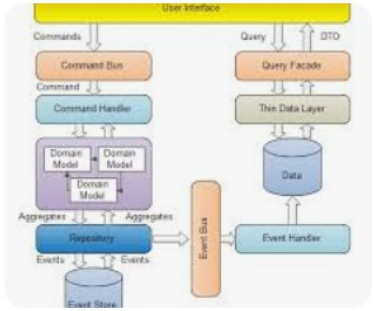
AWS Documentation
CQRS pattern - AWS Prescriptive G…



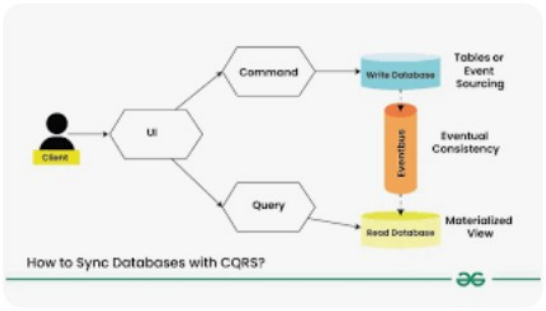
GeeksforGeeks
CQRS - Command Query Responsibility S…



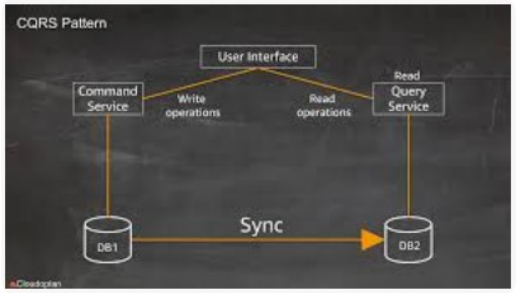
CodeOpinion
CQRS & Event Sourcing Code Walk-Through - Code…



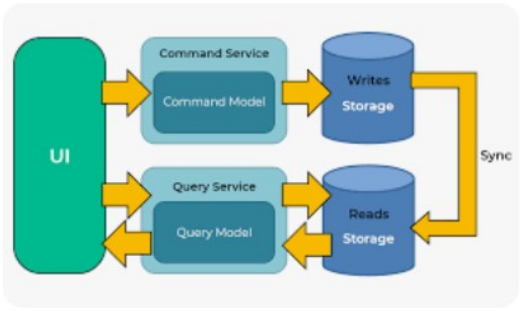
Code Project
Code Project



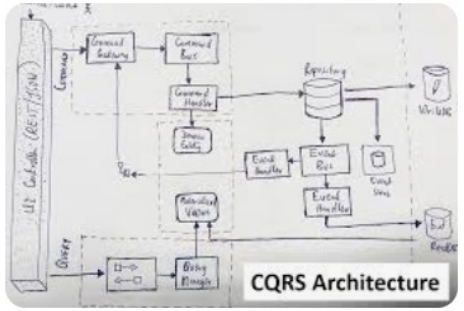
GeeksforGeeks
CQRS - Command Query Responsibility Segr…



Cloudopian
CQRS pattern – Cloudopian



Medium
Revitalizing Transaction-Heavy Systems by …

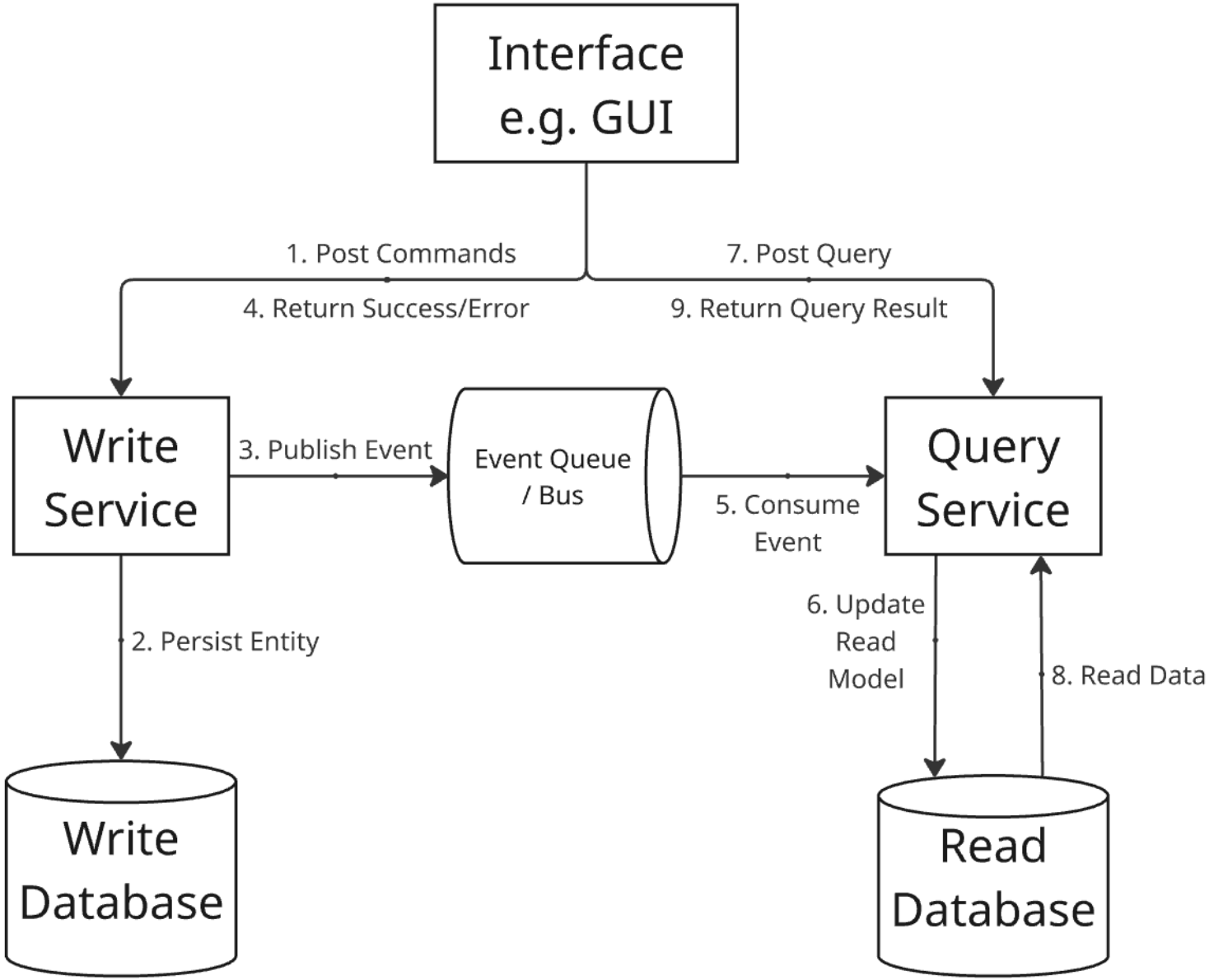


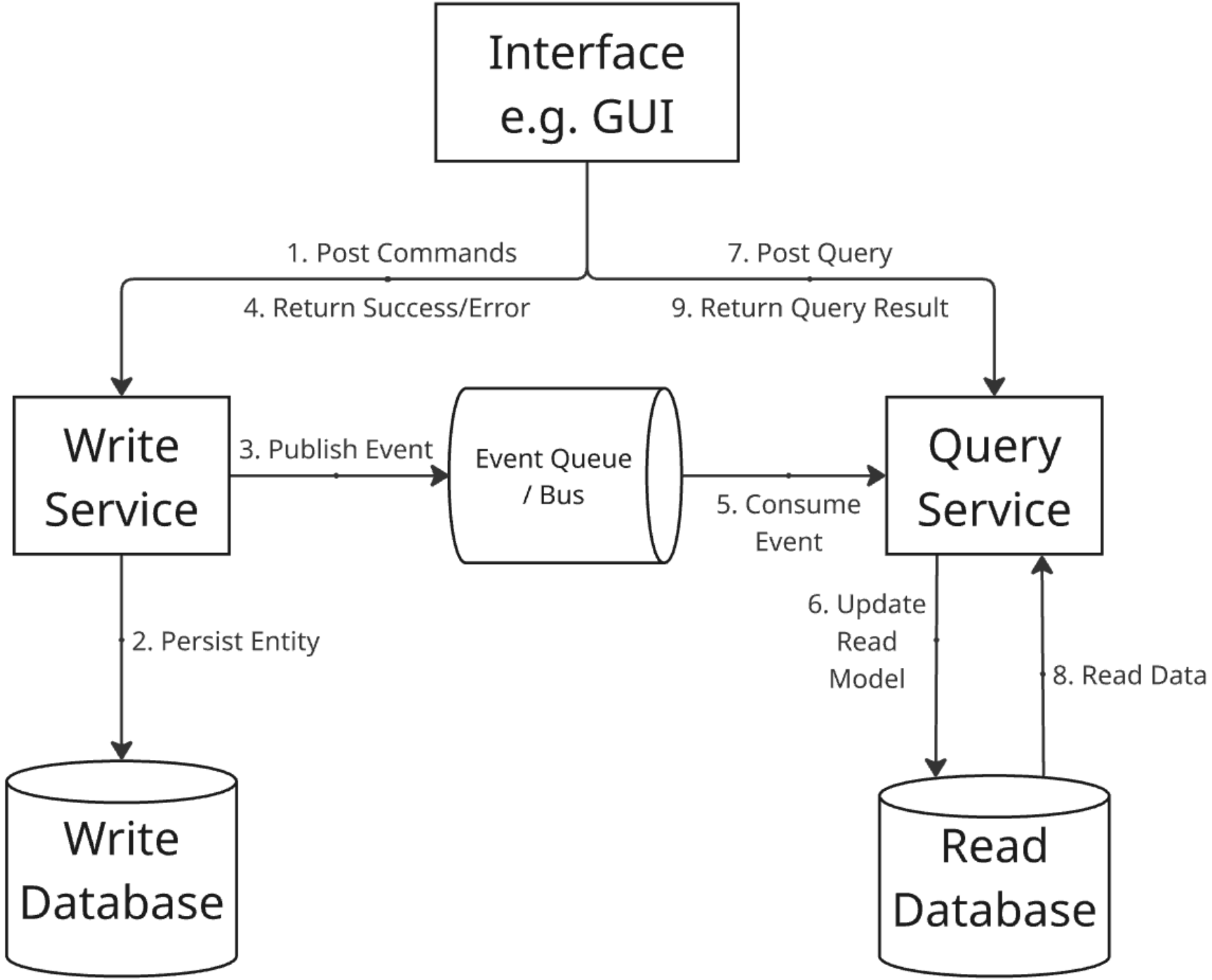
Kindson The Genius
Understanding the Architecture of Eve…

Ergebnis einer Google Bilder Suche
“cqrs diagram”

Command Query Responsibility Segregation

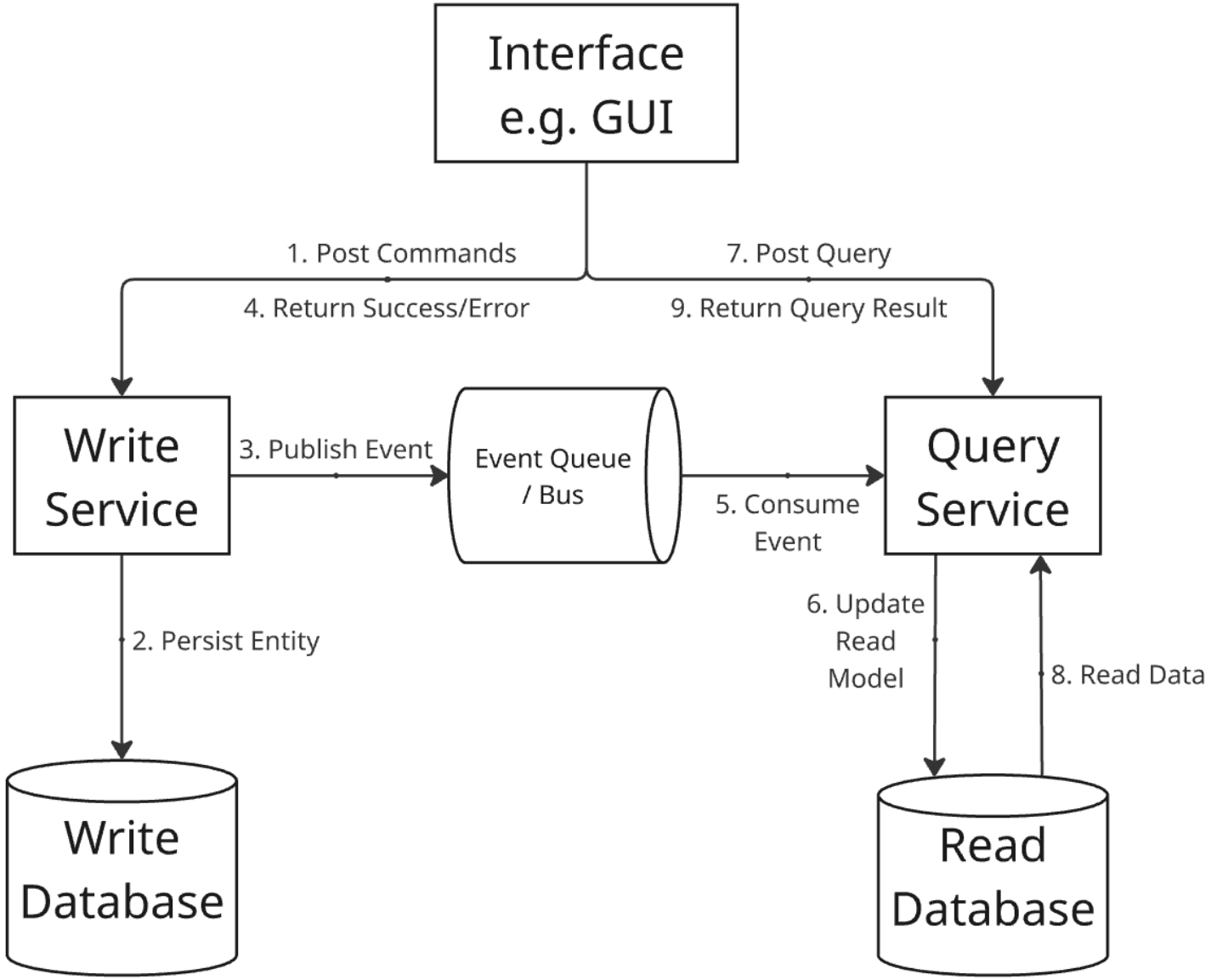
Was die Meisten denken, was es ist




Interface
e.g. GUI
1. Post Commands
4. Return Success/Error
7. Post Query
9. Return Query Result
Write
Service
3. Publish Event
Event Queue
/ Bus
5. Consume
Event
Query
Service
2. Persist Entity
6. Update
Read
Model
8. Read Data
Write
Database
Read
Database

Command Query Responsibility Segregation

Was die meisten denken, was es ist



Alle Vor- und Nachteile eines **verteilten Systems**

- Zwei Deployables
- Entkopplung
- Ausfallsicherheit
- Skalierbarkeit
- Observability
- Maintenance
- Infrastruktur

⇒ Komplex und teuer

JUNGHEINRICH

Command Query Responsibility Segregation

Was es eigentlich ist

“The rationale is that [...], having the **same conceptual model for commands and queries** leads to a more complex model that **does neither well**.”

“[…] **split** [...] **conceptual model into separate models** for update and display, which it refers to as **Command and Query** respectively [...]”

“[…] models, **probably** running in different logical processes, **perhaps** on separate hardware.”

Quelle: https://martinfowler.com/bliki/CQRS.html
Hervorhebungen von mir

Unser Lösungsansatz

Leichtgewichtige, iterative Adaption

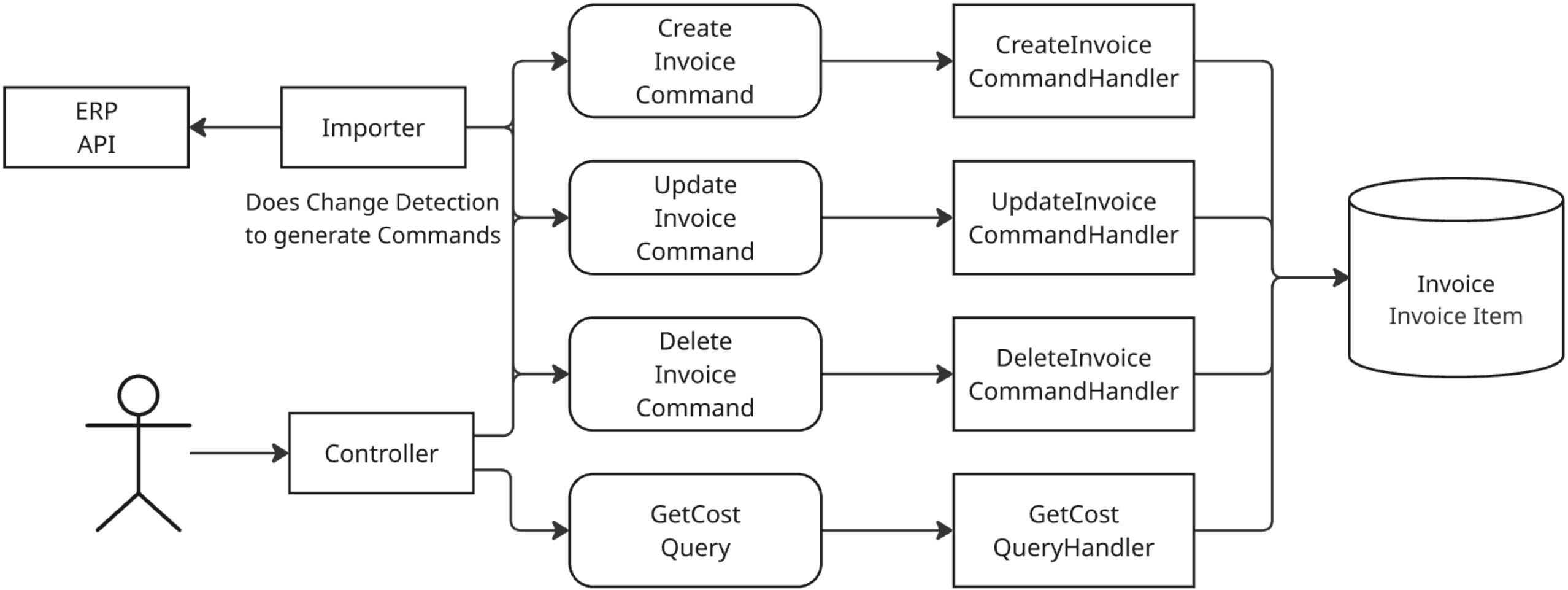
JUNGHEINRICH

1. Iteration

Separate Commands and Queries

1. Iteration

Separate Commands and Queries



JUNGHEINRICH

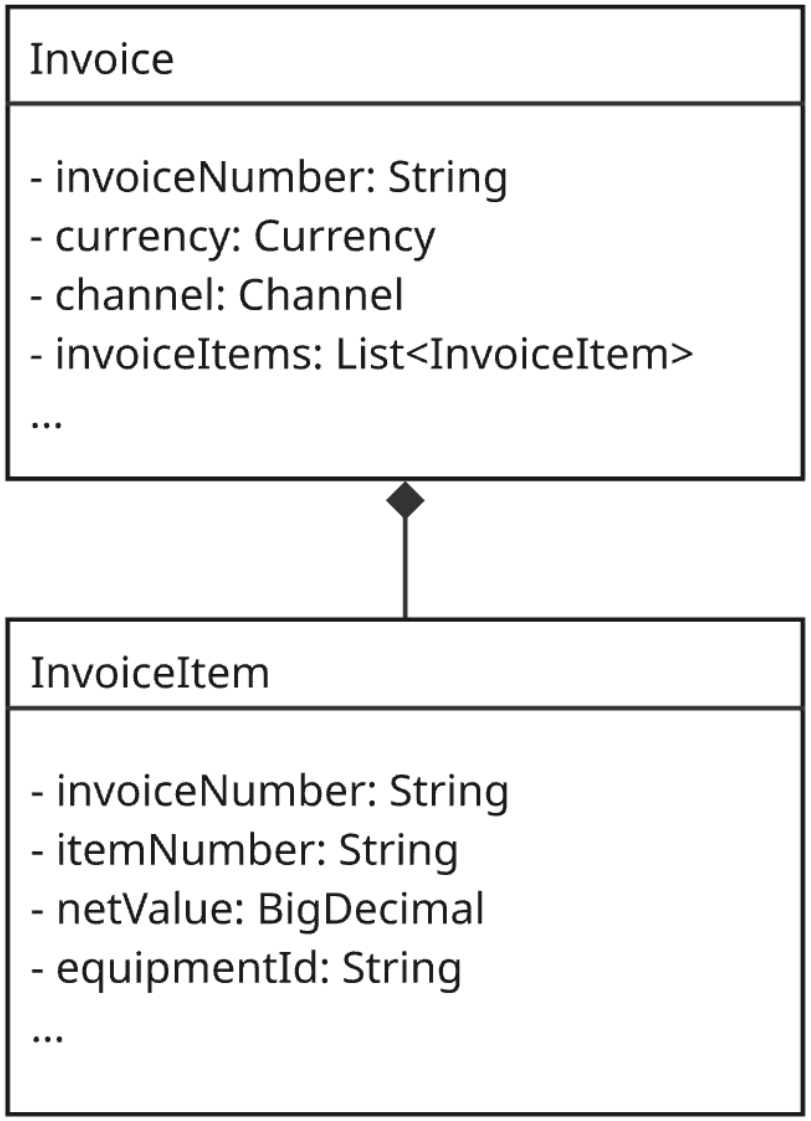
1. Iteration
Separate Commands and Queries

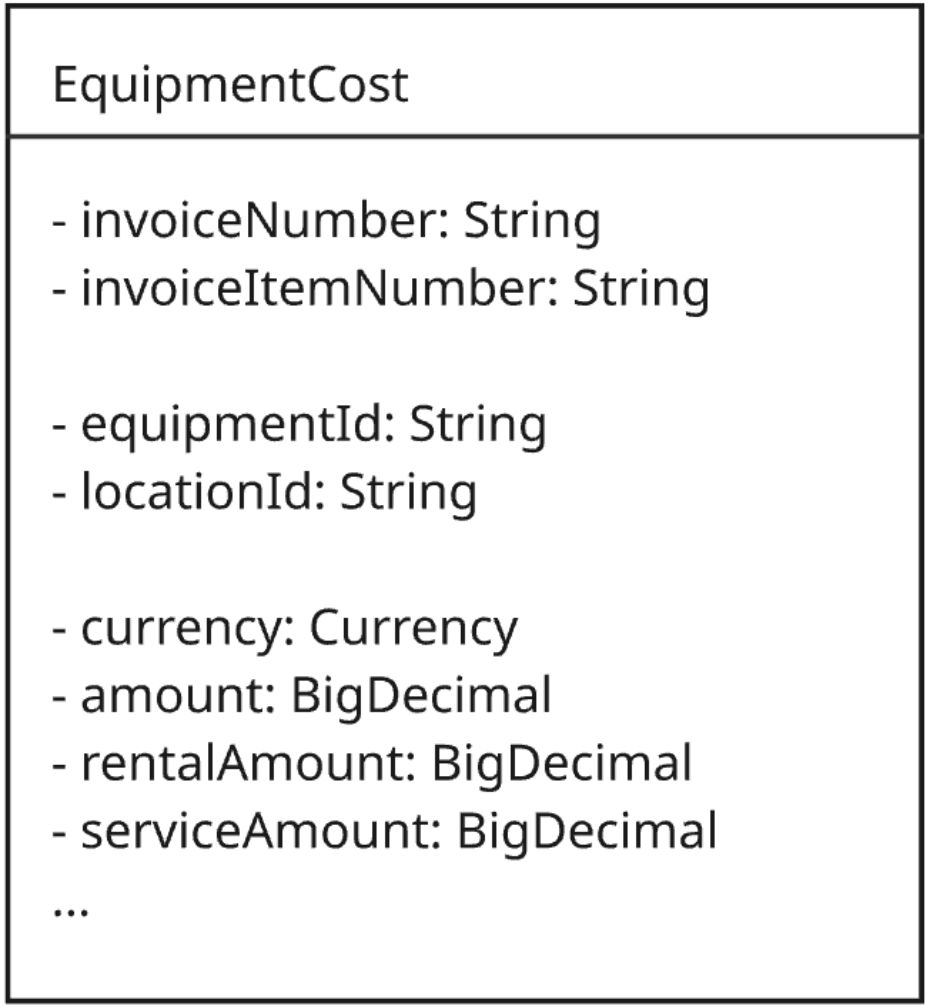
Zwischenergebnis:

- Klare Trennung der Verantwortlichkeiten
- Sauberer Ausgangspunkt für weitere Iterationen
- Keinerlei Änderung der Business Logik

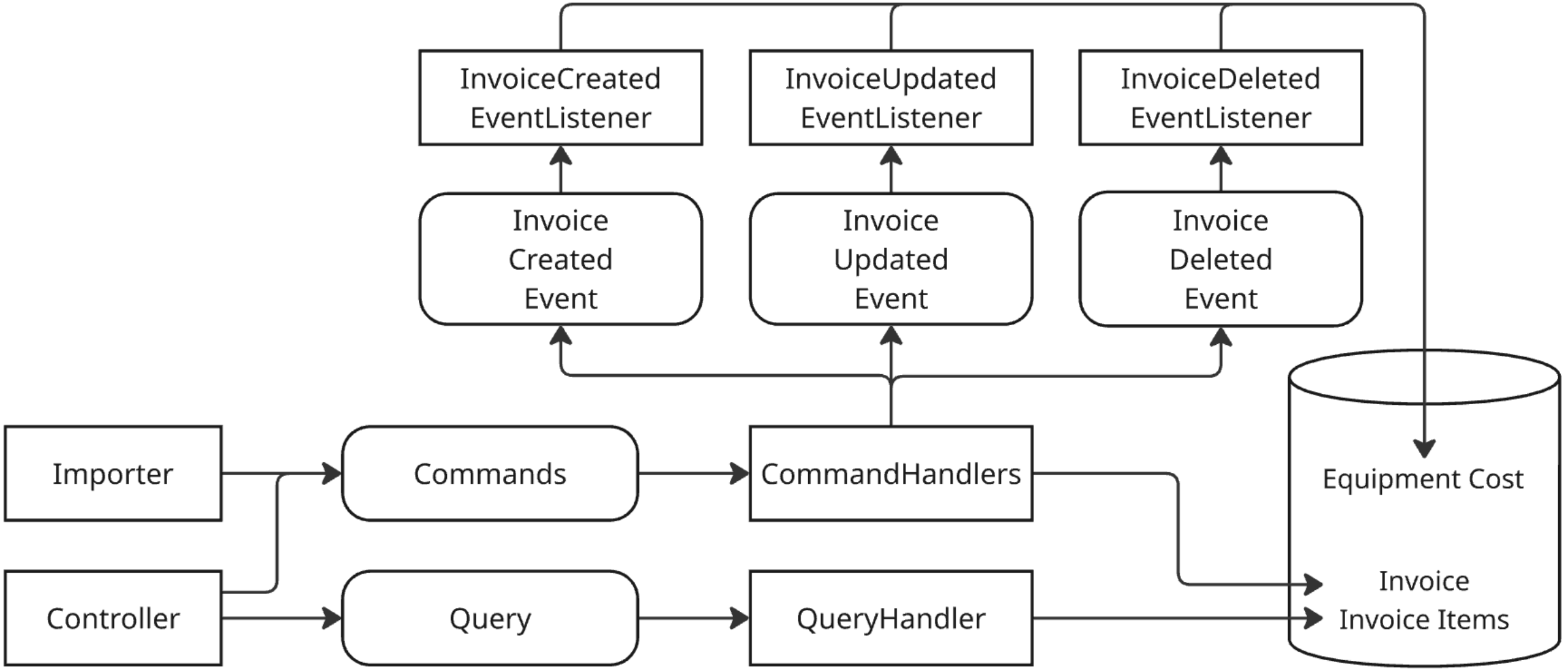
2. Iteration
Cost Model Schreiben

2. Iteration
Cost Model Schreiben





2. Iteration
Cost Model Schreiben



2. Iteration
Cost Model Schreiben

```
@Component
public class CreateInvoiceCommandHandler {

    @Autowired
    private final ApplicationEventPublisher eventPublisher;

    @Transactional
    public void handle(CreateInvoiceCommand command) {

        var invoice = invoiceFrom(command);
        invoiceRepository.save(invoice);

        var event = new InvoiceCreatedEvent(invoice);
        eventPublisher.publishEvent(event);
    }
}
```

2. Iteration
Cost Model Schreiben

CommandHandler verwaltet Transaktion:

- Sollte der Listener eine Exception werfen, wird das Command zurückgerollt
- Benötigt, um Invoices und Equipment Cost in Sync zu halten
- Als Konsequenz: ApplicationEventListener muss synchron ausgeführt werden

2. Iteration
Cost Model Schreiben

```
public record InvoiceCreatedEvent(
        Invoice invoice
) {}
```

Event enthält Invoice Referenz:

- Einfachste Lösung
- Alternativen
  - Rechnungsnummer / ID
  - Relevante Felder
- Prinzipiell Design- und Kapselungsentscheidung

2. Iteration

Cost Model Schreiben

```
@Component
public class EquipmentCostInvoiceCreatedEventListener {

    @EventListener
    public void listen(InvoiceCreatedEvent event) {

        var equipmentCost = fromInvoice(event.invoice);
        equipmentCostRepository.save(equipmentCost);

    }
}
```

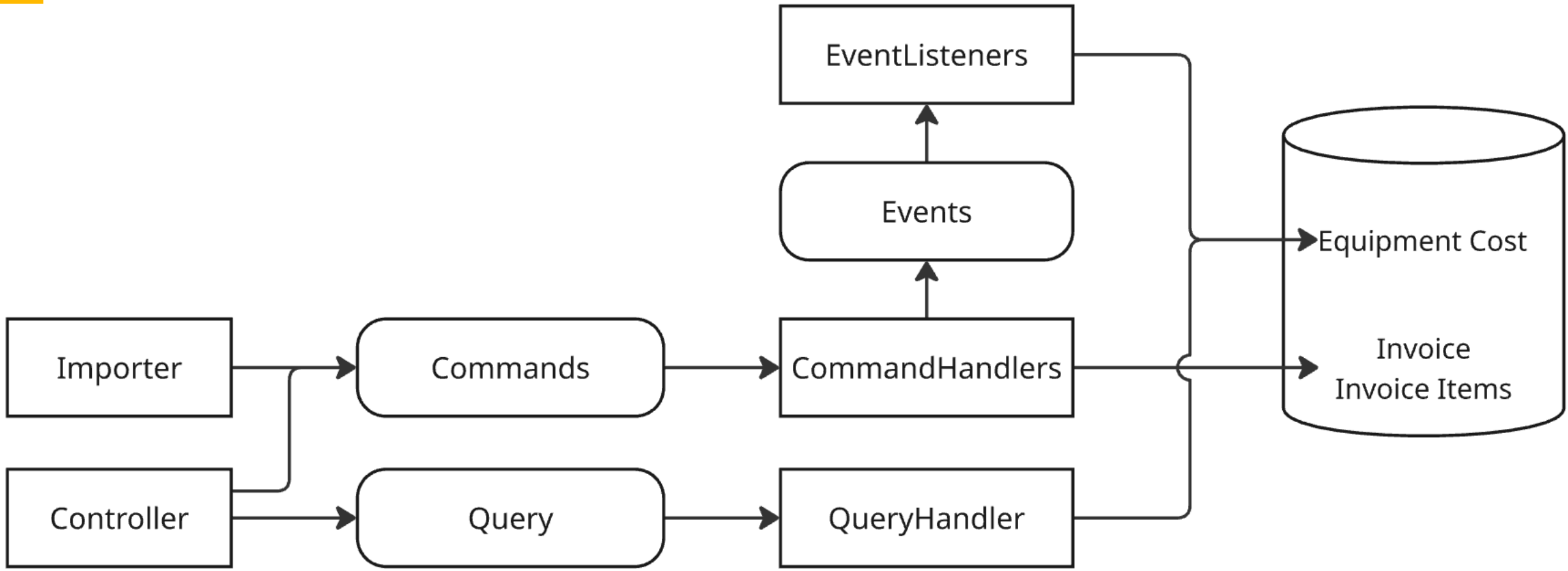
2. Iteration
Cost Model Schreiben

Zwischenergebnis:

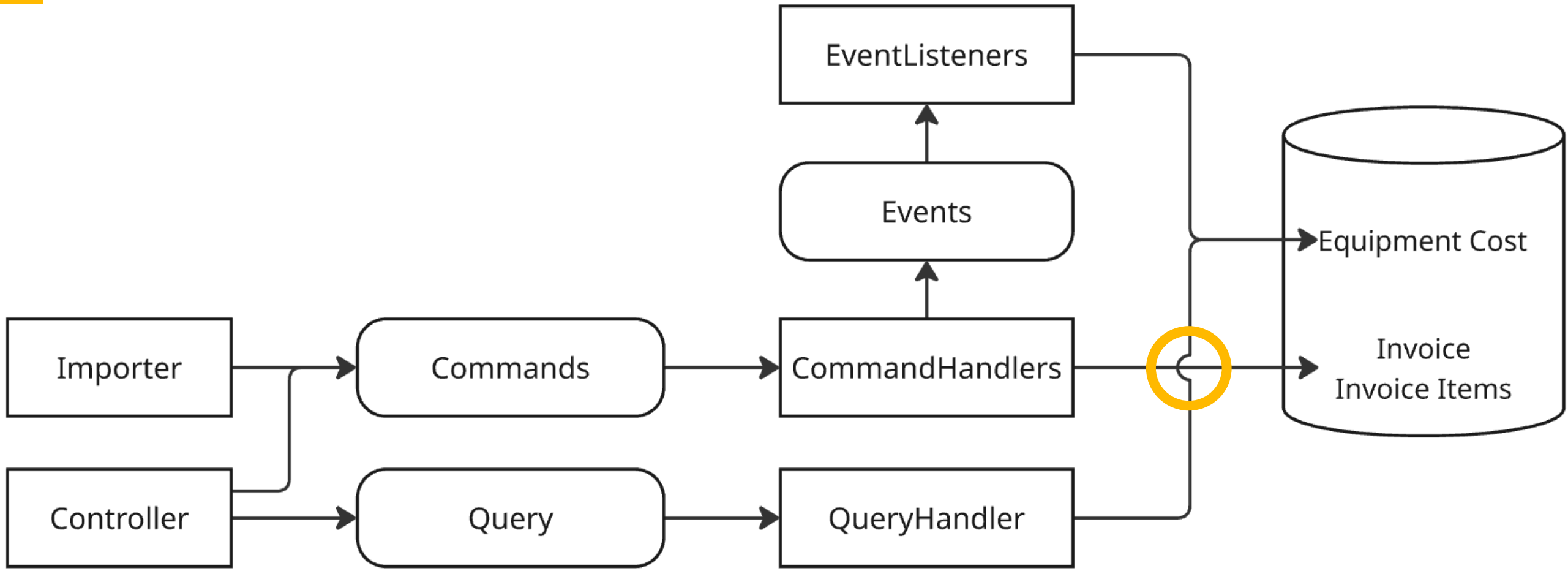
- Änderungen nur auf Schreib-Seite
  - Konzentration auf Korrektheit des Modells
  - Konzentration auf Korrektheit der Technik (z. B. Transaktionen)
- Lese-Seite unbeeinflusst
  - Weder Code- noch Test-Änderungen notwendig

3. Iteration
Cost Model Lesen

3. Iteration
Cost Model Lesen



3. Iteration
Cost Model Lesen



3. Iteration
Cost Model Lesen

Vorgehen:

- Erstellen zusätzlicher Endpunkte + QueryHandler, die aus dem Cost Model lesen
- Anlegen von Regressionstests auf Endpunktebene
  - Jest-basiert (Node)
  - Ruft beide Endpunkte auf und erwartet das exakt gleiche Ergebnis
- Initialisierung des Equipment Cost Models: Für jede Rechnung ein InvoiceCreatedEvent absetzen
- Frontend-Feature Toggle für Product Owner
- Performance-Evaluierung via k6

3. Iteration
Cost Model Lesen

Zwischenergebnis:

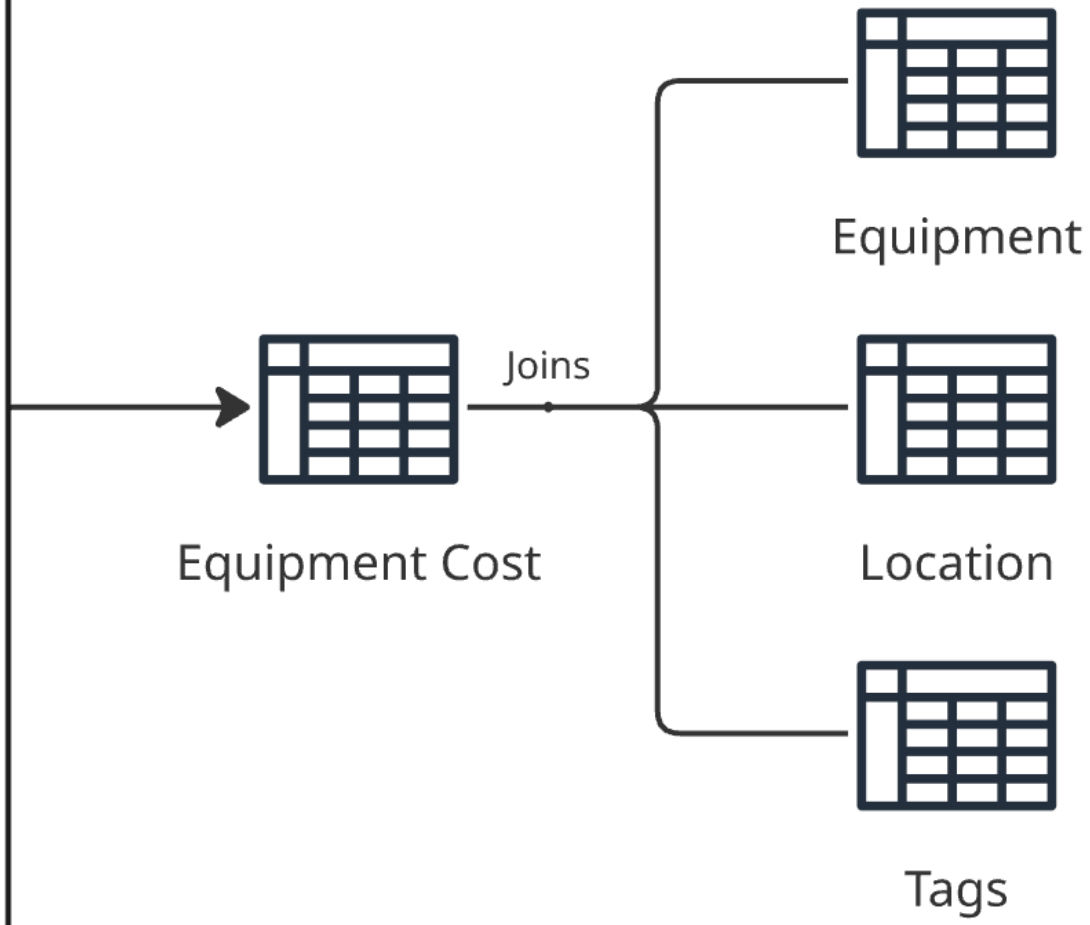
- Performance-Vorteil kommt zum Tragen
- Fokus auf Umstellung der Lese-Seite
- Aber: Komplexe Filter-Optionen erlauben weitere Optimierung

4. Iteration
Denormalisierung

4. Iteration

Denormalisierung

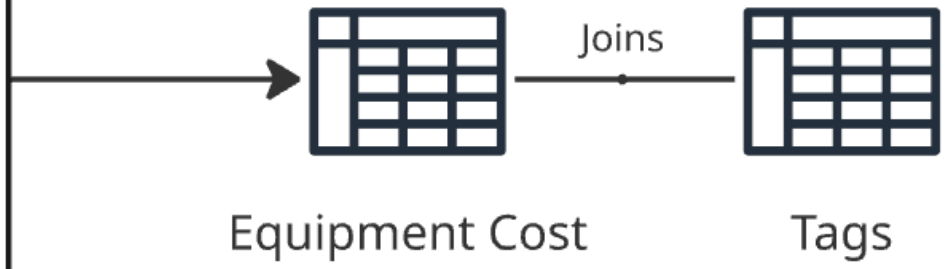
```
EquipmentCostQuery

filters: {
  timeSpan: {
    start: '2022-01-01',
    end: '2023-01-01'
  },
  locations: [...],
  segments: [...],
  tags: [...]
},
sort: {
  field: "location.city",
  order: "asc"
}
```



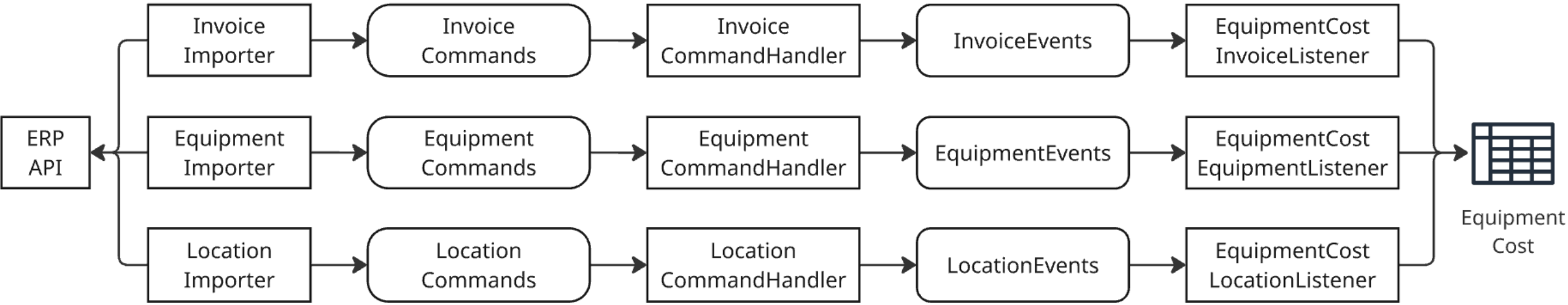
4. Iteration
Denormalisierung

EquipmentCost
- invoiceNumber - invoiceItemNumber - currency - amount - rentalAmount - serviceCost ... - locationCity - locationStreet - equipmentSegment - equipmentManufacturer - equipmentYearOfConstruction ...

Vorteil: Performance Gewinn
Herausforderung: Daten aktuell halten



4. Iteration
Denormalisierung



Transaktions-Scope

Race Conditions?
Optimistic locking!

Re-Cap
Was wir bis hier haben

- Getrennte Read- und Write-Models
  - Read-Model optimiert für OLAP-Anforderungen*
- ApplicationEvent und in-process Kommunikation
  - Kein zusätzliches Deployment, keine zusätzliche Infrastruktur, …
  - Transaktionsgarantie durch synchrone Events und optimistic Locking
- Guter Ausgangspunkt für Erweiterungen

⇒ Performance im Griff

⇒ Architektur / Business Logik im Griff

⇒ Leichtgewichtig und flexibel erweiterbar

* Online Analytical Processing

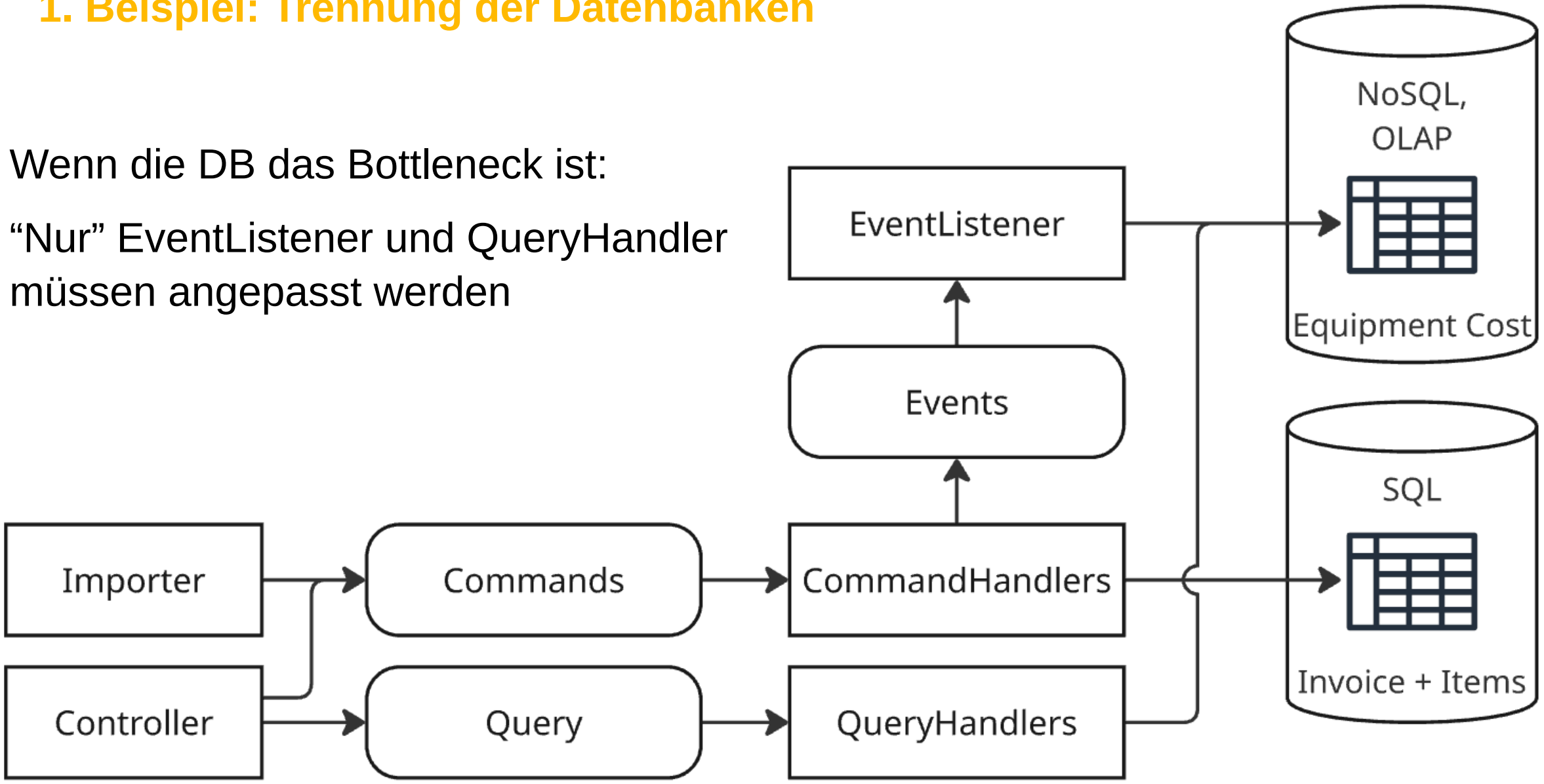
Erweiterungsmöglichkeiten
Vier Beispiele

Erweiterungen

1. Beispiel: Trennung der Datenbanken

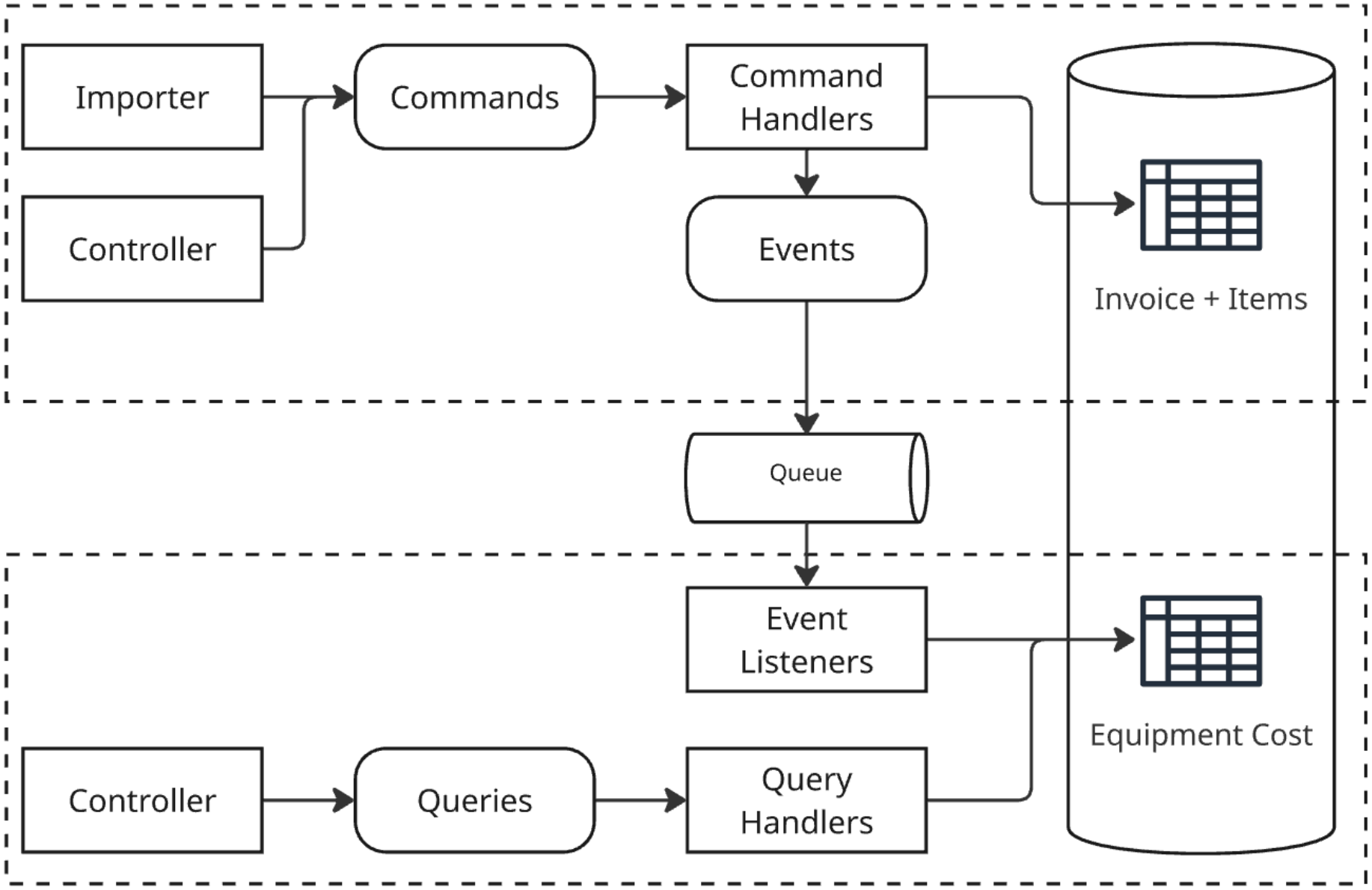
Wenn die DB das Bottleneck ist:

“Nur” EventListener und QueryHandler müssen angepasst werden



Erweiterungen

2. Beispiel: Trennen Lesen + Schreiben in zwei Deployables



Wenn die CPU das Bottleneck ist:

ApplicationEvents mit Queue und Listener mit z. B. @SqsListener ersetzen

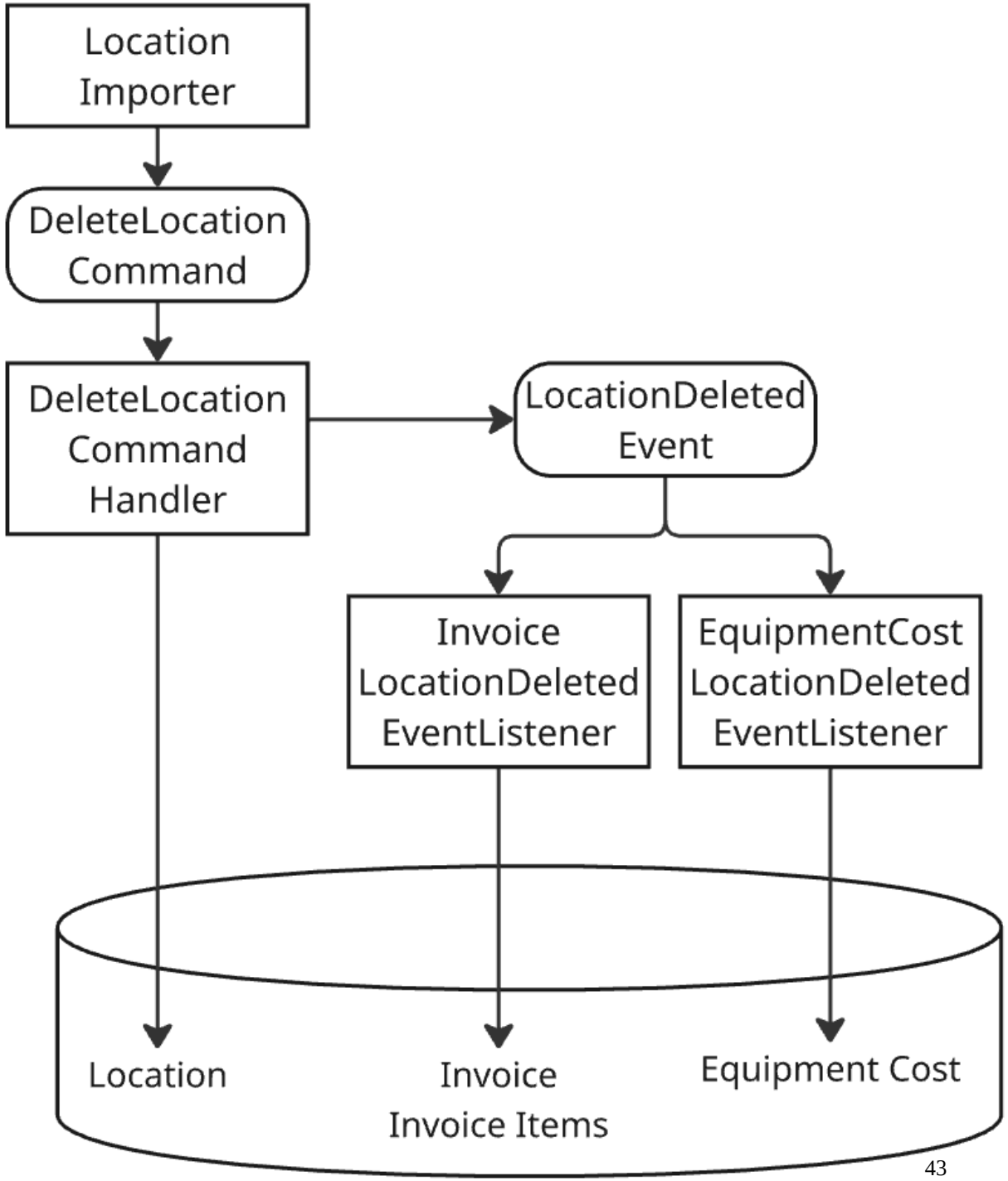
Verfügbare Controller und Handler Beans über Spring Profile steuern

Erweiterungen
3. Beispiel: Datenlöschung

Kaskadierendes Löschen:

Über Events lösbar, selbst wenn Business Events nicht aus Quellsystem kommen

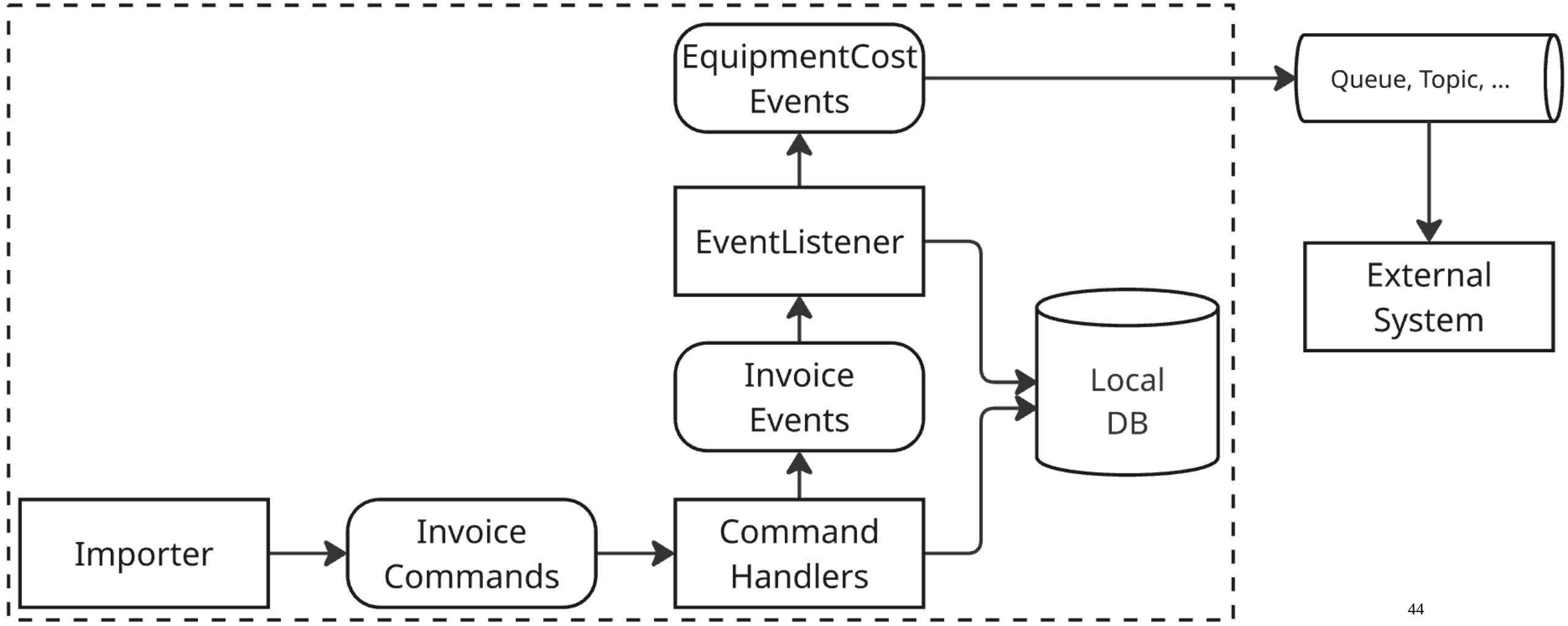
Business-Logik bleibt im Modell:
z. B. bei abweichenden Anforderungen



Erweiterungen

4. Beispiel: Data Sharing / Events für externe Systeme

Anforderung: Kosten mit Drittsystemen teilen



Fazit

- Gedanklich einen Schritt zurück machen:

  Neubetrachtung der Domäne **mit etwas Abstand** kann erhellend sein

- Mit kleinen Schritten voran:

  Iteratives Vorgehen immer empfehlenswert

- Nimm nur das mit, was du brauchst:

  CQRS ist **nicht** gleichbedeutend mit einem **verteilten System**

- “Langweilig” ist cool:

  Etablierte Features können große Probleme lösen

Vielen Dank an mein Team!

Vielen Dank für eure Aufmerksamkeit!

Fragen und Antworten

Gerne jetzt, später oder über meine Kontaktdaten

www.simonkerler.de

github.com/namelessvoid

linkedin.com/in/simon-kerler

simon.kerler@jungheinrich.de

simon_kerler@web.de


JUNGHEINRICH