Microservices & Event-getriebene Architektur (MEA)

WASP 1, SS 2025

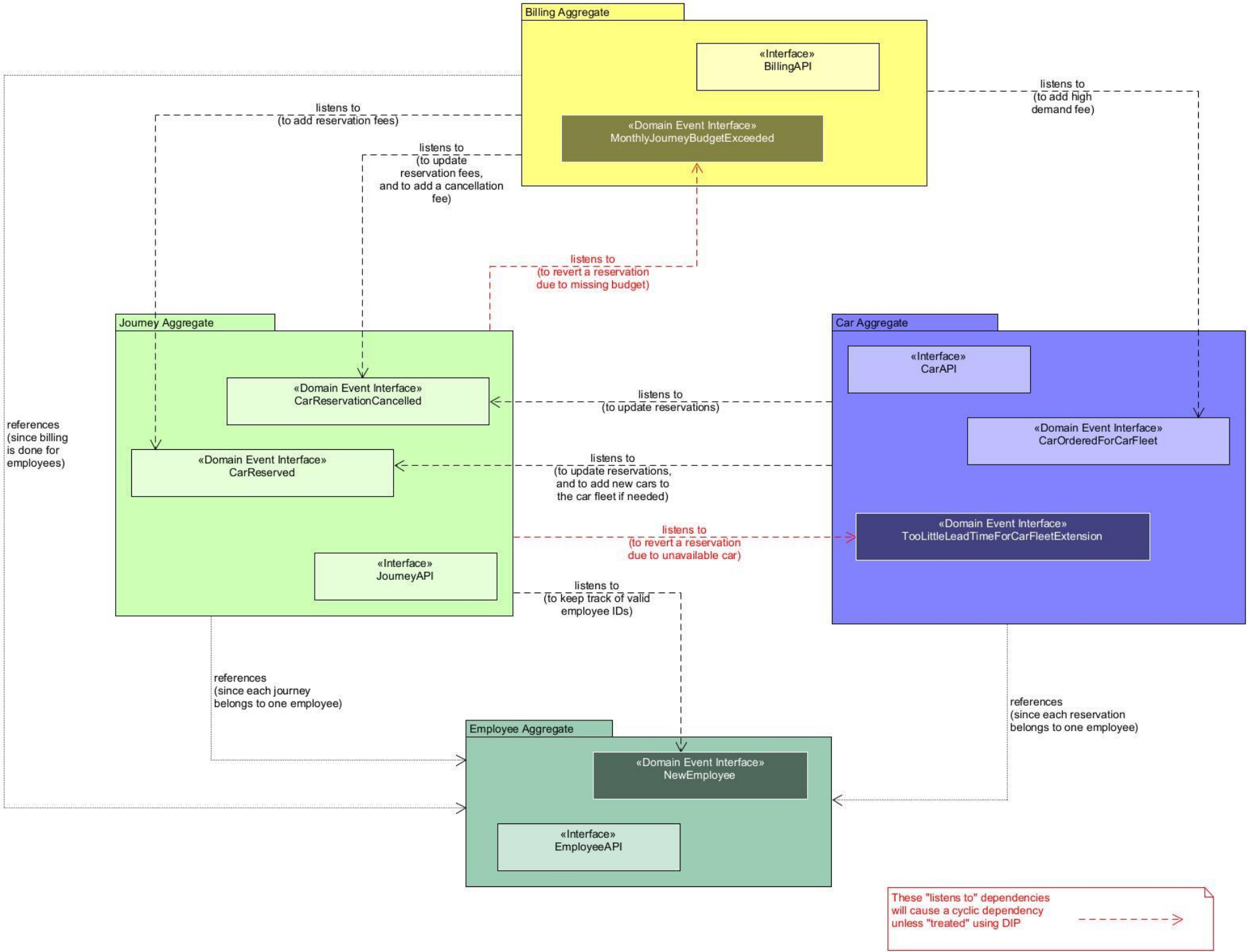
Prof. Dr.-Ing. Stefan Bente

Data Redundancy & Eventual Consistency

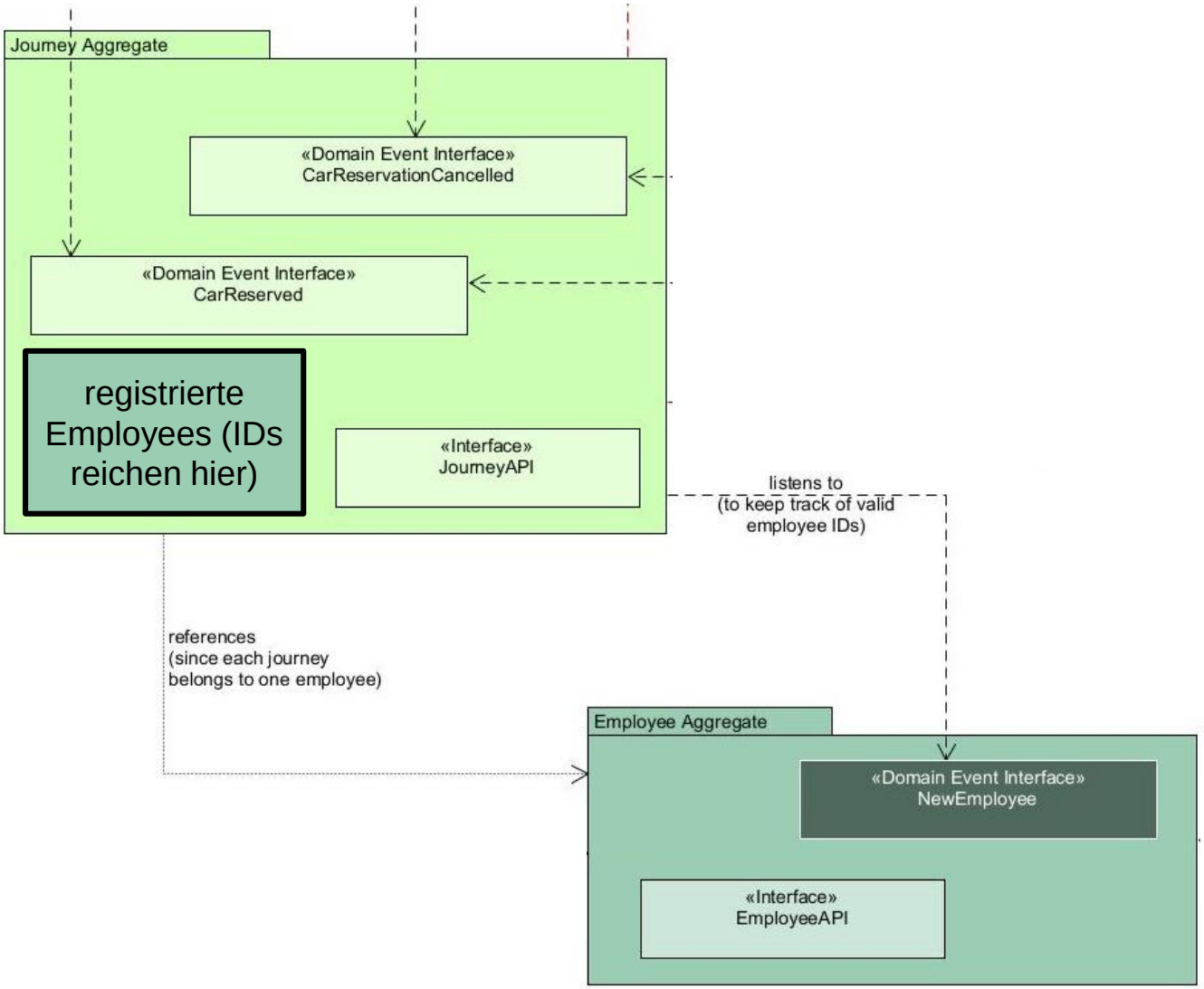
Technology Arts Sciences
TH Köln

Data Redundancy

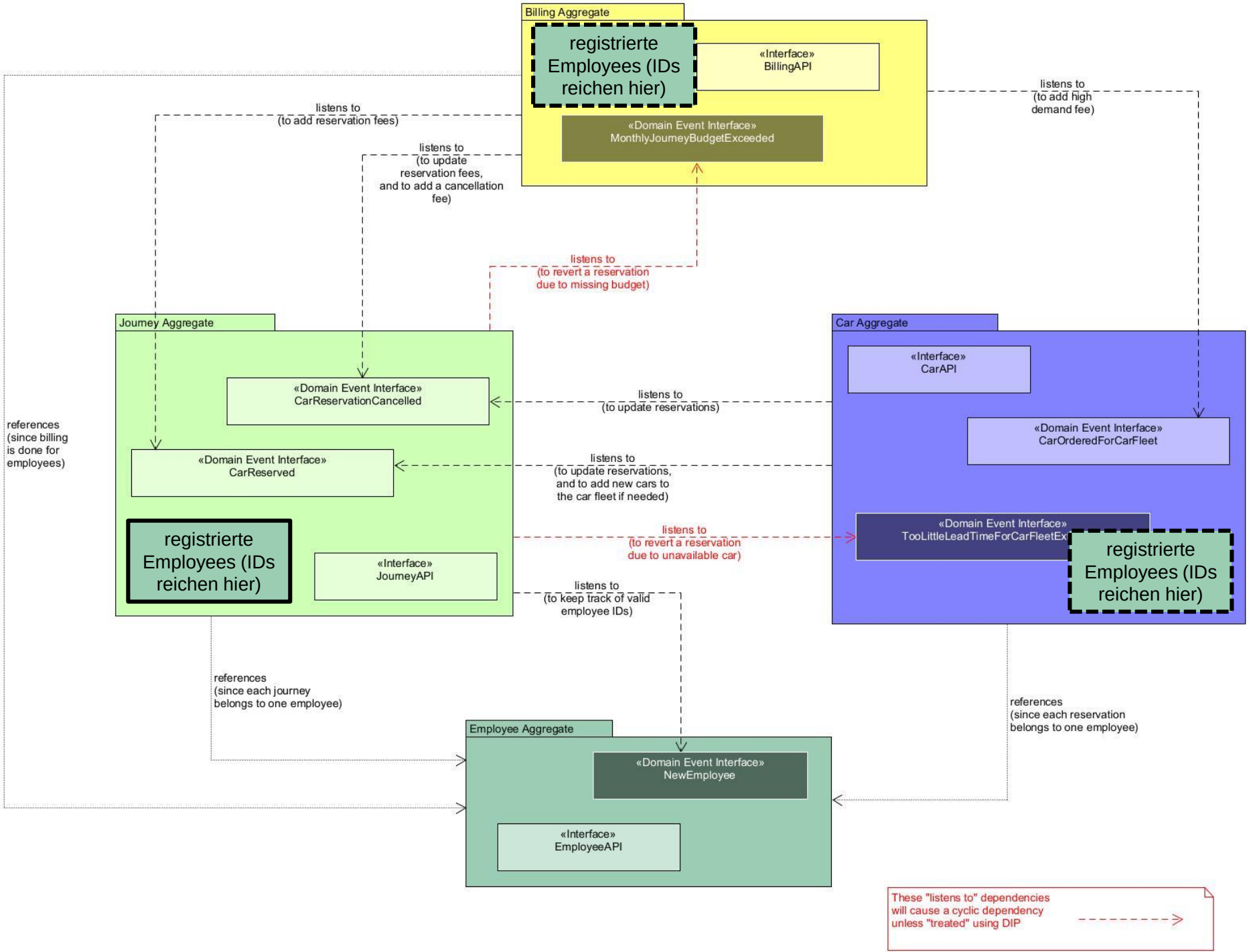
Technology
Arts Sciences
TH Köln


Billing Aggregate
«Interface»
BillingAPI
«Domain Event Interface»
MonthlyJourneyBudgetExceeded
listens to
(to add reservation fees)
listens to
(to update
reservation fees,
and to add a cancellation
fee)
listens to
(to add high
demand fee)
listens to
(to revert a reservation
due to missing budget)
Journey Aggregate
«Domain Event Interface»
CarReservationCancelled
«Domain Event Interface»
CarReserved
«Interface»
JourneyAPI
Car Aggregate
«Interface»
CarAPI
«Domain Event Interface»
CarOrderedForCarFleet
«Domain Event Interface»
TooLittleLeadTimeForCarFleetExtension
listens to
(to update reservations)
listens to
(to update reservations,
and to add new cars to
the car fleet if needed)
listens to
(to revert a reservation
due to unavailable car)
listens to
(to keep track of valid
employee IDs)
references
(since billing
is done for
employees)
references
(since each journey
belongs to one employee)
references
(since each reservation
belongs to one employee)
Employee Aggregate
«Domain Event Interface»
NewEmployee
«Interface»
EmployeeAPI
These "listens to" dependencies
will cause a cyclic dependency
unless "treated" using DIP

Woher weiß Journey, welche Employees registriert sind?



Technology Arts Sciences TH Köln


Billing Aggregate
registrierte Employees (IDs reichen hier)
«Interface» BillingAPI
«Domain Event Interface» MonthlyJourneyBudgetExceeded
listens to (to add reservation fees)
listens to (to update reservation fees, and to add a cancellation fee)
listens to (to add high demand fee)
listens to (to revert a reservation due to missing budget)
Journey Aggregate
«Domain Event Interface» CarReservationCancelled
«Domain Event Interface» CarReserved
registrierte Employees (IDs reichen hier)
«Interface» JourneyAPI
Car Aggregate
«Interface» CarAPI
«Domain Event Interface» CarOrderedForCarFleet
«Domain Event Interface» TooLittleLeadTimeForCarFleetEx
registrierte Employees (IDs reichen hier)
listens to (to update reservations)
listens to (to update reservations, and to add new cars to the car fleet if needed)
listens to (to revert a reservation due to unavailable car)
listens to (to keep track of valid employee IDs)
references (since billing is done for employees)
references (since each journey belongs to one employee)
references (since each reservation belongs to one employee)
Employee Aggregate
«Domain Event Interface» NewEmployee
«Interface» EmployeeAPI
These "listens to" dependencies will cause a cyclic dependency unless "treated" using DIP

Eventual Consistency

Was passiert hier?

```java
@Test
public void testNaiveHandlingOfEventualConsistency() {
    // given
    carAPI.extendCarFleet( d0 ); // just 1 car in the fleet

    // when
    idAlfred = employeeAPI.add( "Alfred" );
    journeyAPI.planJourney( idAlfred, d1, d3,  carNeeded: true );

    // then
    long cost = billingAPI.getTotalCostFor( idAlfred );
    assertEquals(  expected: 3 * COST_PER_DAY.getFee(), cost );
}
```

Definition *Eventual Consistency*

- Eventual = „letztendlich“, „schließlich“
  - sog. **false friend** (heißt NICHT „eventuell“, „vielleicht“ …)

- Eventual consistency is a model that guarantees that updates will propagate through the system and eventually be applied to all nodes, given enough time.
  https://www.baeldung.com/cs/eventual-consistency-vs-strong-eventual-consistency-vs-strong-consistency

- **ACID** vs. **BASE**

Verteilte, asynchrone Systeme

Monolithische, synchrone Systeme (oft basierend auf einer zentralen relationalen DB)

Technology Arts Sciences TH Köln

ACID

- **A**tomicity
  - Ensures that all changes we make to the data as part of a transaction, we make them as a single entity and operation.
  - This effectively means that either we perform **all the changes or none of them**.
- **C**onsistency
  - Consistency ensures that we execute all the data changes while maintaining a consistent state at the start and the end of a transaction.
  - A consistent state of data must conform to all the constraints that we define for data.
- **I**solation
  - ensures that we keep the intermediate states of a transaction **invisible to other transactions**.
  - This gives concurrently running transactions an effect of being **serialized**.
- **D**urability
  - ensures that when a transaction completes, we persist changes to the data
  - any other transaction doesn't revert those changes.

http://baeldung.com/cs/transactions-intro#1-acid-properties

BASE

- **B**asically **A**vailable
  - The database ensures that all data is available by ensuring that it's stored **across multiple nodes**.
  - This means that even if some nodes are unavailable, it's much more likely that all of the data can still be accessed, but at the cost that it takes a non-zero amount of time for the data to spread across the nodes.
- **S**oft State
  - because the same data is stored across multiple nodes, and because this data may take time to get to every node, this means that the **database can't strongly enforce data integrity**.
  - Instead, it's assumed that the **client application will be doing this.**
- **E**ventually Consistent
  - because the data is stored across multiple nodes, there is no guarantee that reading from one node will immediately reflect writes to another node.
  - These reads will be consistent **in time but not immediately.**

Wie kann ich das besser testen?

```java
@Test
public void testNaiveHandlingOfEventualConsistency() {
    // given
    carAPI.extendCarFleet( d0 ); // just 1 car in the fleet

    // when
    idAlfred = employeeAPI.add( "Alfred" );
    journeyAPI.planJourney( idAlfred, d1, d3, carNeeded: true );

    // then
    long cost = billingAPI.getTotalCostFor( idAlfred );
    assertEquals( expected: 3 * COST_PER_DAY.getFee(), cost );
}
```

Wartezeit!

Technology Arts Sciences TH Köln

`scenario.stimulate( … )` zum Testen mit Wartezeit

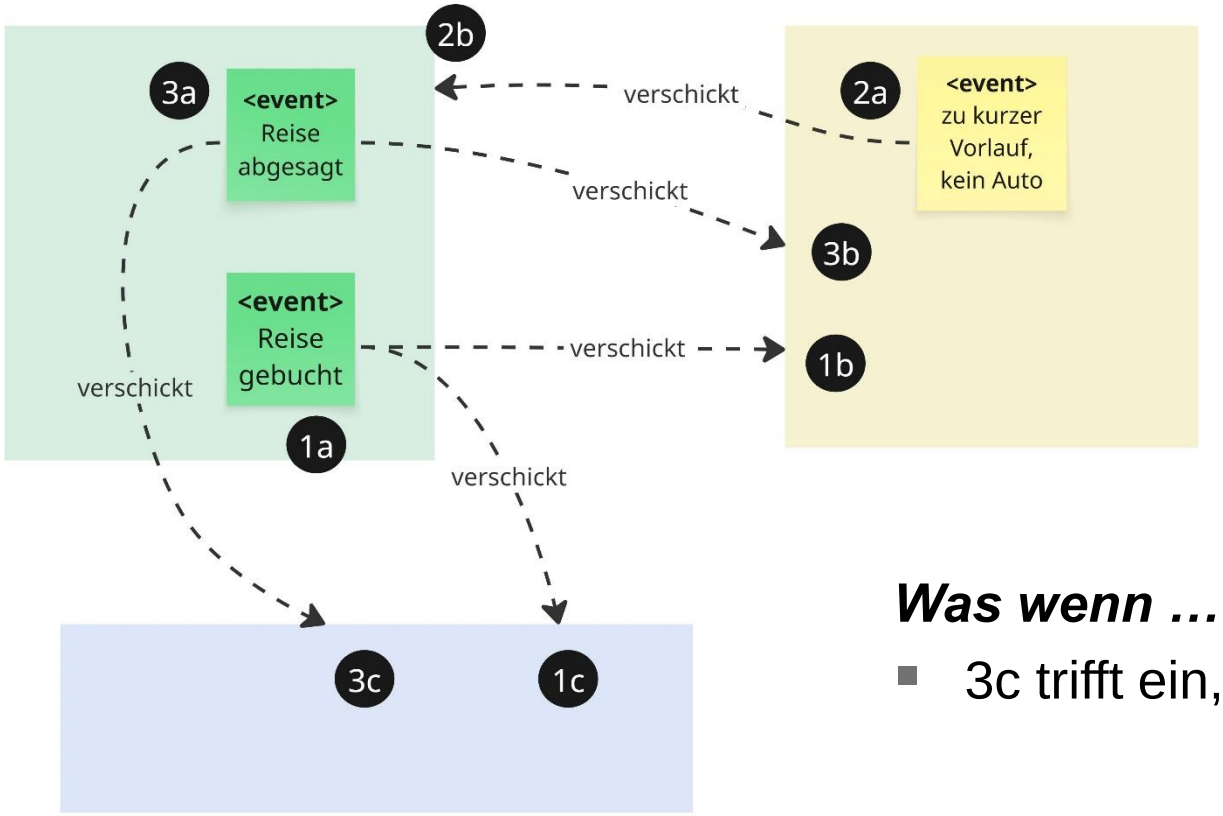
```
@Test
public void testBetterHandlingOfEventualConsistency( Scenario scenario ) {
    // given
    carAPI.extendCarFleet( d0 ); // just 1 car in the fleet
    LocalDate sixDaysAfter = d7.plusDays( daysToAdd: 6 );

    // when
    idAlfred = employeeAPI.add( "Alfred" );
    scenario.stimulate( () -> journeyAPI.planJourney( idAlfred, d1, d3, carNeeded: true ) )
            .andWaitForStateChange( () -> billingAPI.getTotalCostFor( idAlfred ),
                    Long cost -> cost > 0 );

    // then
    long cost = billingAPI.getTotalCostFor( idAlfred );
    assertEquals( expected: 3 * COST_PER_DAY.getFee(), cost );
}
```


Technology
Arts Sciences
TH Köln

Wenn wir nicht bei Modulith, sondern bei Microservices wären …

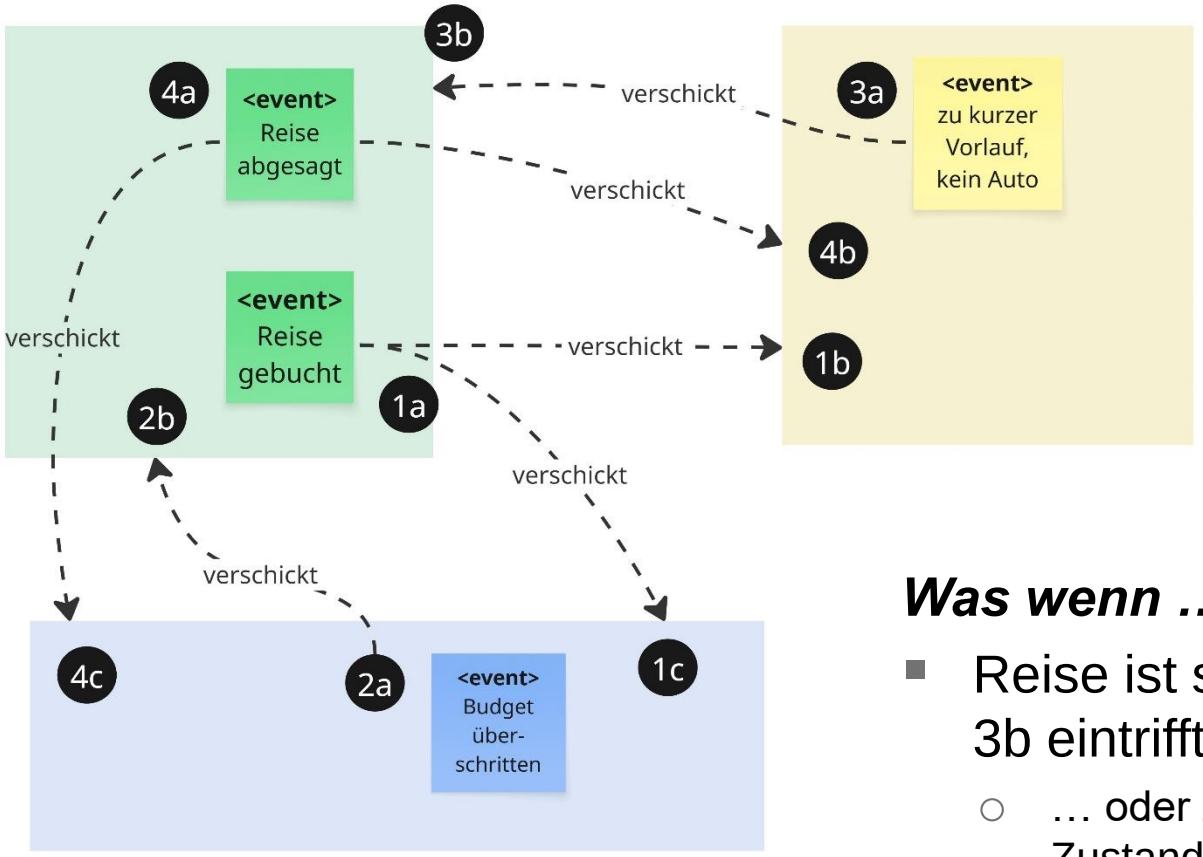


Was wenn …

- 3c trifft ein, bevor 1c fertig?

Technology
Arts Sciences
TH Köln

Wären wir nicht bei Modulith, sondern bei Microservices …



Was wenn …

- Reise ist schon gelöscht, wenn 3b eintrifft?
  - … oder zumindest in „falschem“ Zustand
- 1c später als 4c?

Technology
Arts Sciences
TH Köln