Understanding Microservices Architecture

Maaike van Putten
Software developer & trainer
www.biltup.com

Y

Session 1

Introduction to microservices

d

Session 2

Comparing microservices and monolithic architecture

d

Session 3

Building and containerizing microservices

E

Session 4

Managing data in microservices

Managing data in microservices

Learning objectives

- Understand how to **handle data persistence** in Docker.
- **Implement** a simple **database service** (like PostgreSQL) as a separate container.
- **Connect** the **database** service to a **Python microservice** for CRUD operations.

Bounded context pattern

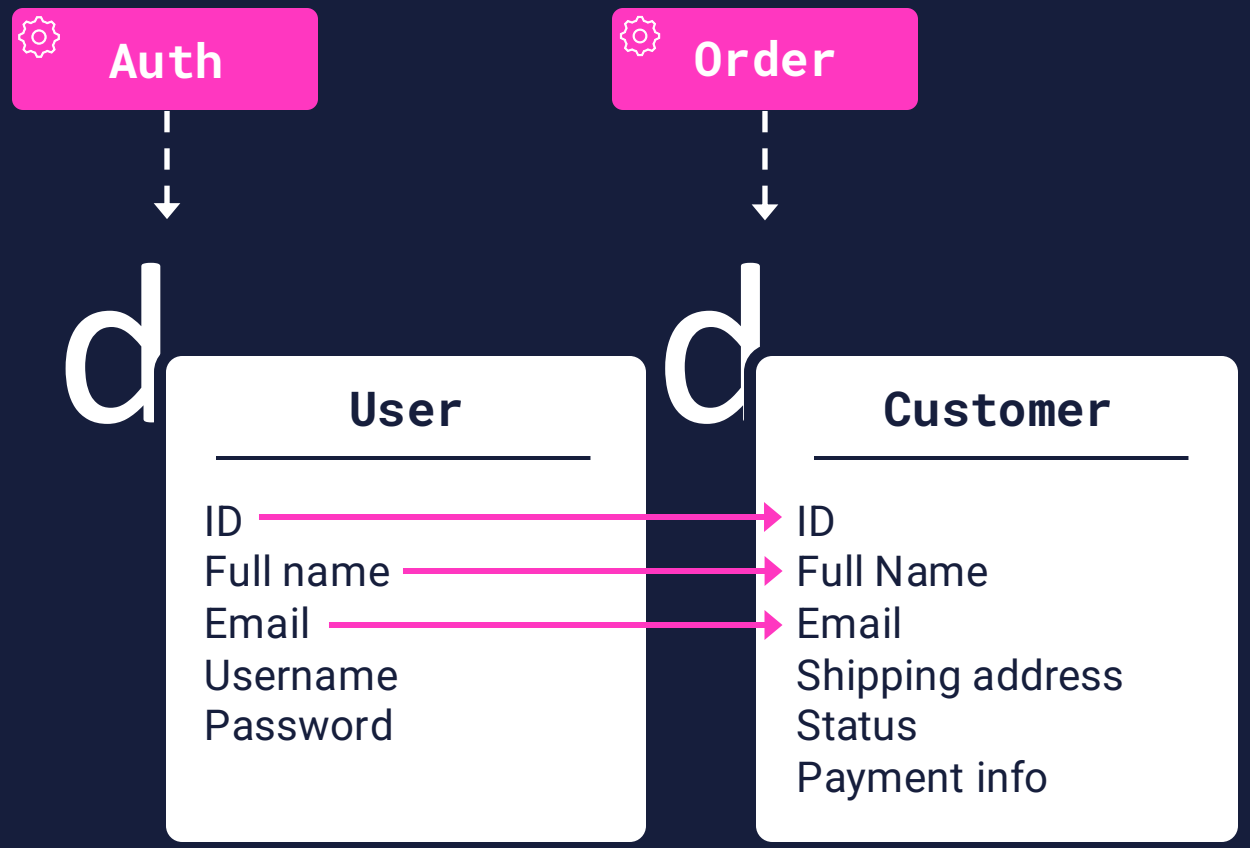
- Splitting up a domain into subdomains
- Each entity has an unambiguous meaning within the subdomain
- Different subdomains can have different properties for same named entities
- Mappers for entities for communication between subdomains
- Bounded contexts in microservices are typically implemented as individual microservices
- They shouldn’t share a common database

Data model

- Different names for entities across subdomains to best match the usage of the entity
- Unique storage
- Consistency and syncing data is a complication
- Complex queries that are sourced from different databases across microservices are harder

Example bounded context

User and customer, different interesting properties for each microservice



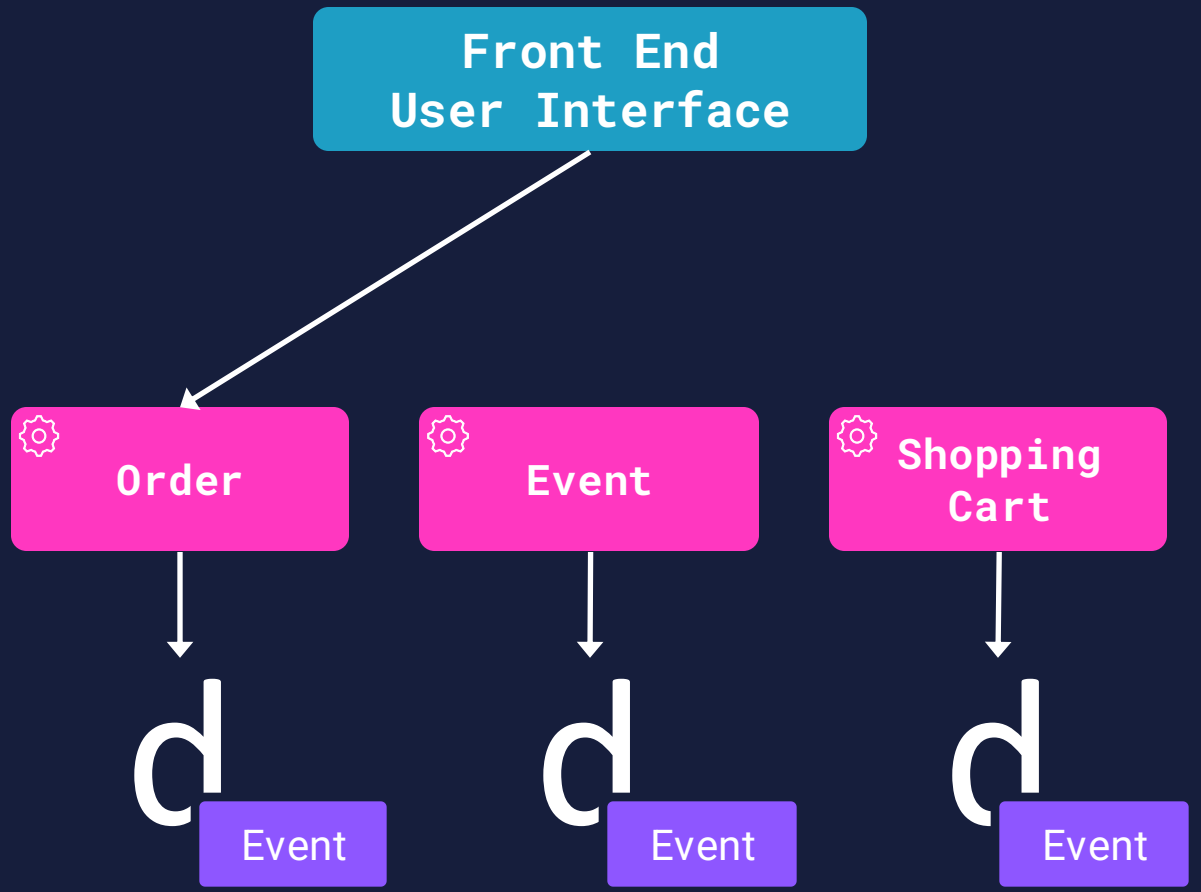
Bounded context and microservice

- Splitting up the landscape into bounded contexts is a great way to start
- Often a bounded context is mapped to a single microservice, but this doesn’t have to be the case
- It also depends on how independent the contexts are in terms of business logic and data ownership
- Also take non-function requirements into account:
  - Be able to independently scale
  - Release frequency and need for changes
  - Separate team for microservice
  - Best choice for the tools (tech stack)
  - Data ownership

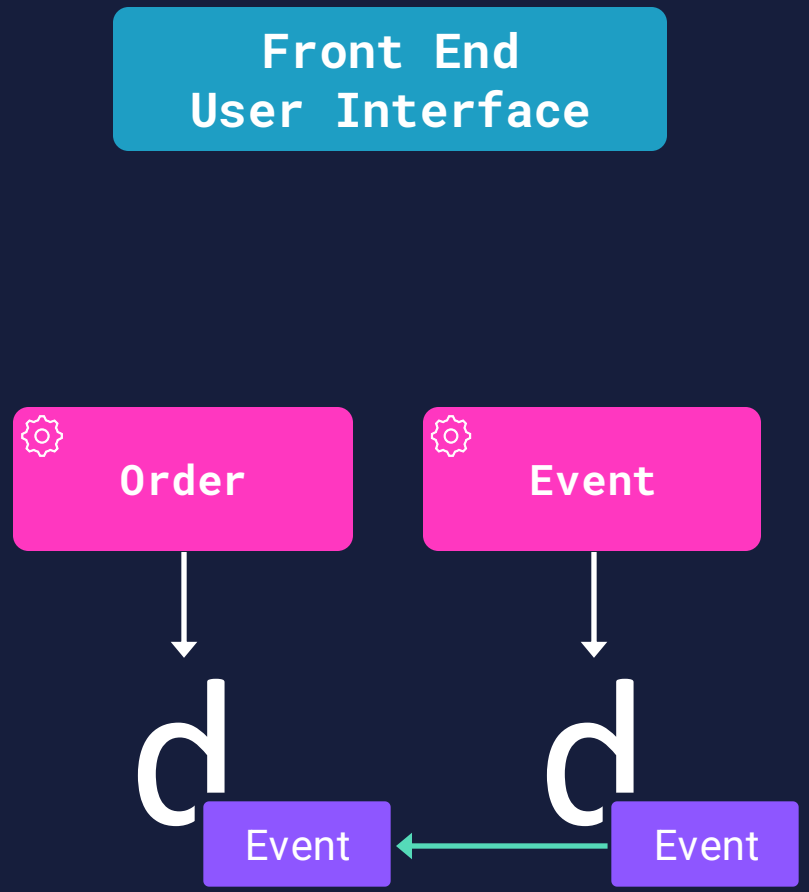
Data design considerations

- E Microservices should manage their own data
- d They should not share a database
- h APIs can be used to get data from the database of another microservice
- f BUT: API calls are costly and they shouldn’t be constantly happening
- l Async calls and service bus for delivering the messages can help (also with resilience if the other service is down for a short time)
- p If two microservice need realtime data from each other and delay cannot be tolerated, and/or have a lot of data that they share in general >> one microservice instead of two

Duplicate data

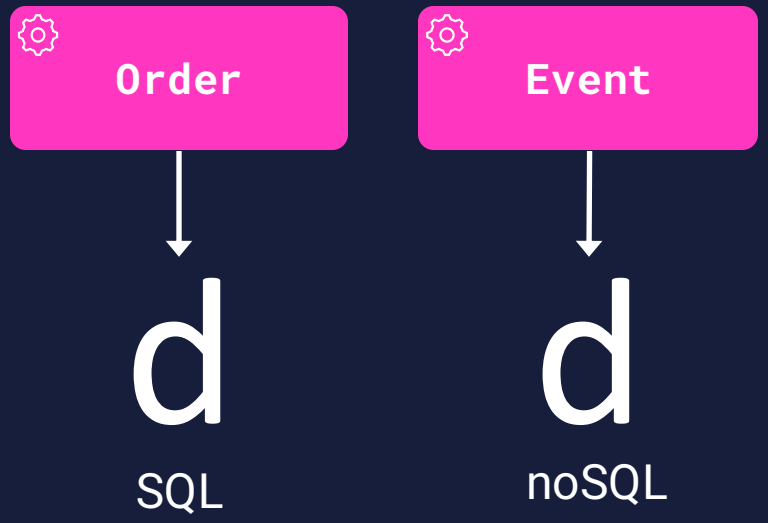


Duplicate data



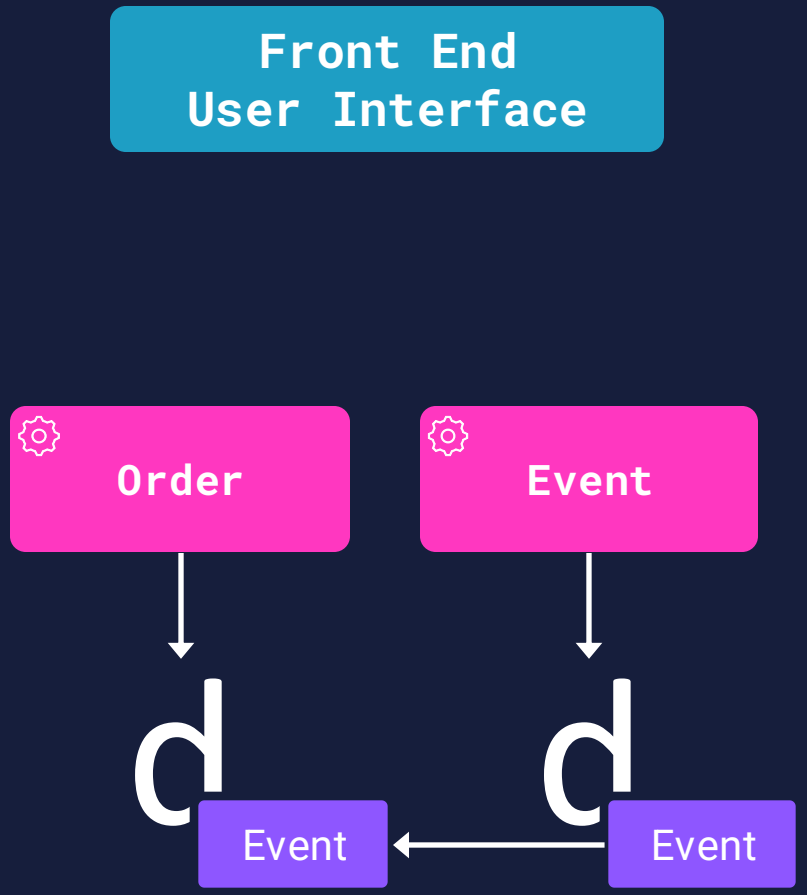
Polygot persistence

- Type of data storage might differ per microservice (for example mixing noSQL and SQL)
- This mixing of data storage techniques is called “polygot persistence”



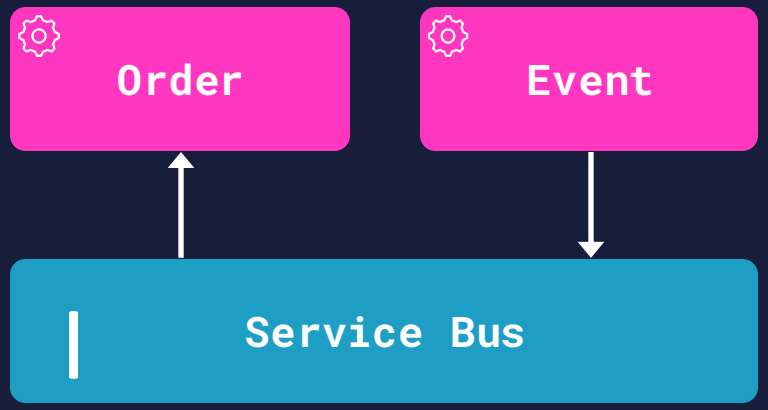
Eventual consistency

- Related data across microservices has to be synced
- Eventual consistency is the name for the most common strategy
- Changes are synced but this takes a little bit (no blocking of the microservices during the update)



Implementing eventual consistency

- Event driven communication with publishers and subscribers
- Async
- System needs to tolerate that not all data is immediately consistent

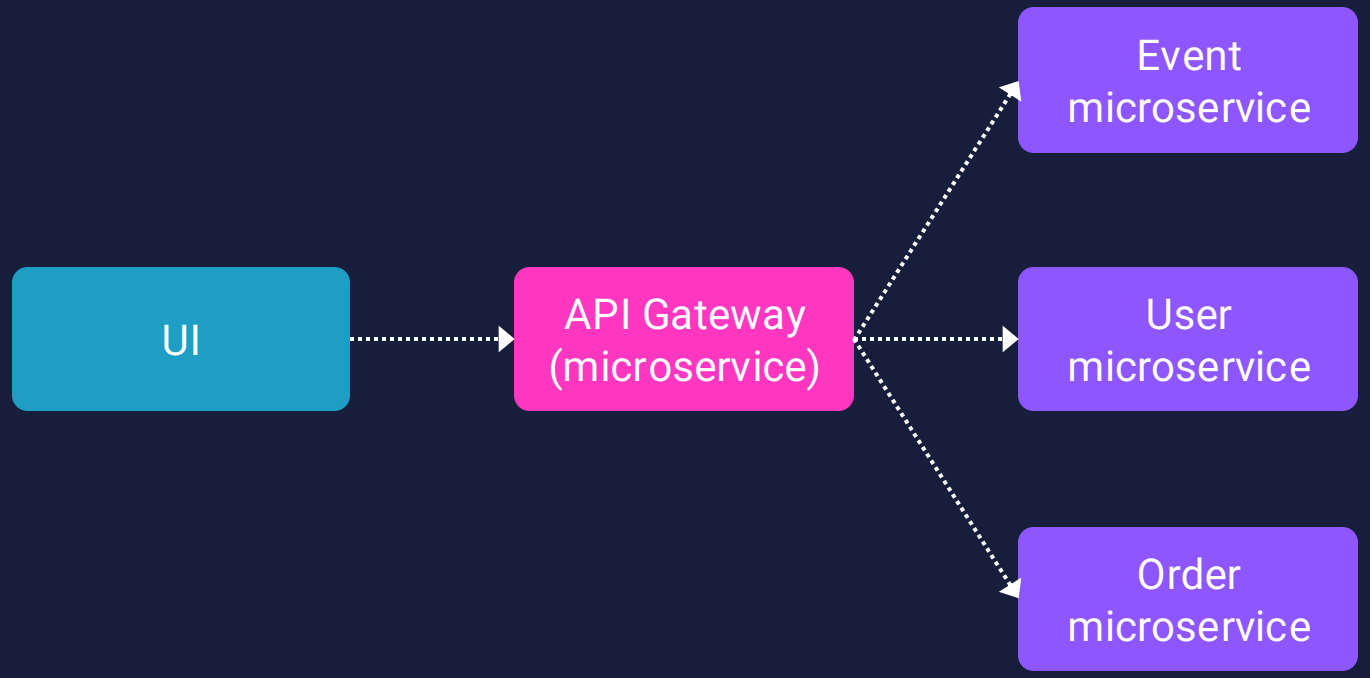


Queries with distributed data

Three common optins:

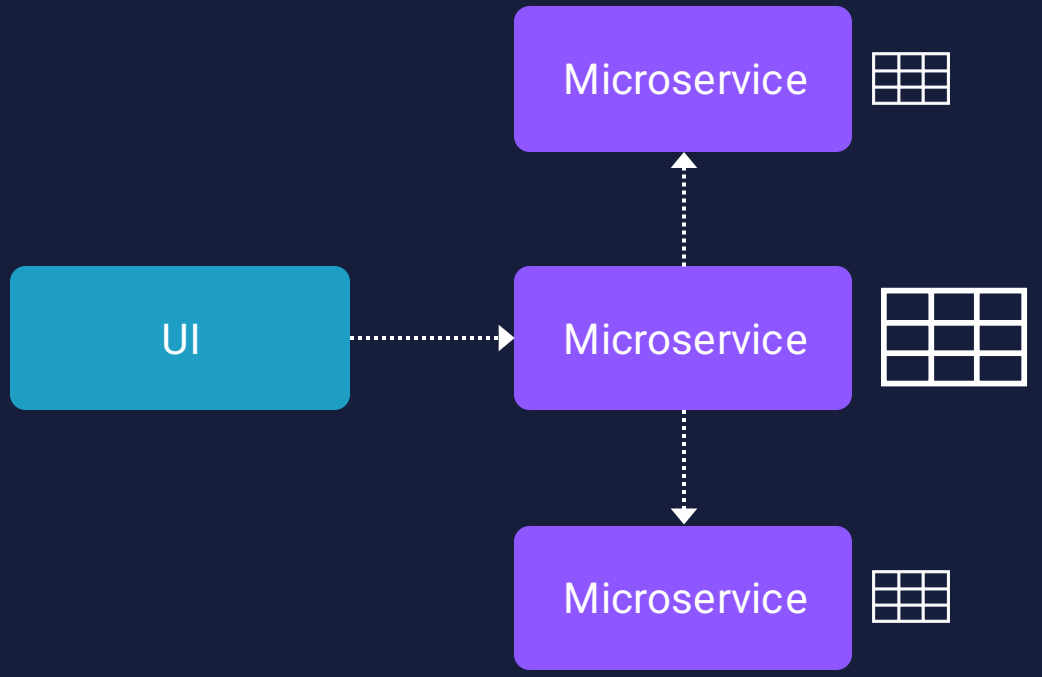
- API gateway
- Materialized views
- Cold data in central databases

API gateway (another microservice itself)



Materialized view

- View of the data in one microservice
- Could give consistency issues



Central reporting database

O

Cold data (data that is not being changed but only read) can be stored in a central database, such as a data warehouse

O

This is used for reporting and not for changing data

Y

Again, consistency issues and eventual consistency approach

Data consistency in distributed systems

Failures can always occur, distributed architecture has multiple pain points

It’s possible that calls between services fail, typically circuit breaker pattern used for this

Service bus also helps to mitigate the issues

Integration events for eventual consistency

What we'll be doing:

- **d** Run a database in a container
- **p** Persist the data in a volume
- **U** Interact with the containerized database from our microservices

Exercise

Adding data persistence to a database service

Exercise

Connect microservices from last time to the database

Questions?

maaikejvp@gmail.com

www.biltup.com