# Highly-Available Applications on Unreliable Infrastructure: Microservice Architectures in Practice

Daniel Richter, Marcus Konrad, Katharina Utecht, and Andreas Polze

Operating Systems & Middleware Group
Hasso Plattner Institute at University of Potsdam, Germany

# Motivation

- EPA – the legacy system
  - reserve and book train seats operated by Deutsche Bahn (German railway)
  - 1 mio seat requests & 300,000 bookings
  - first version: 1980s
  - set of *Pathway Services* as part of *HP NonStop* system
  - especially fault-tolerant and highly-available

DB

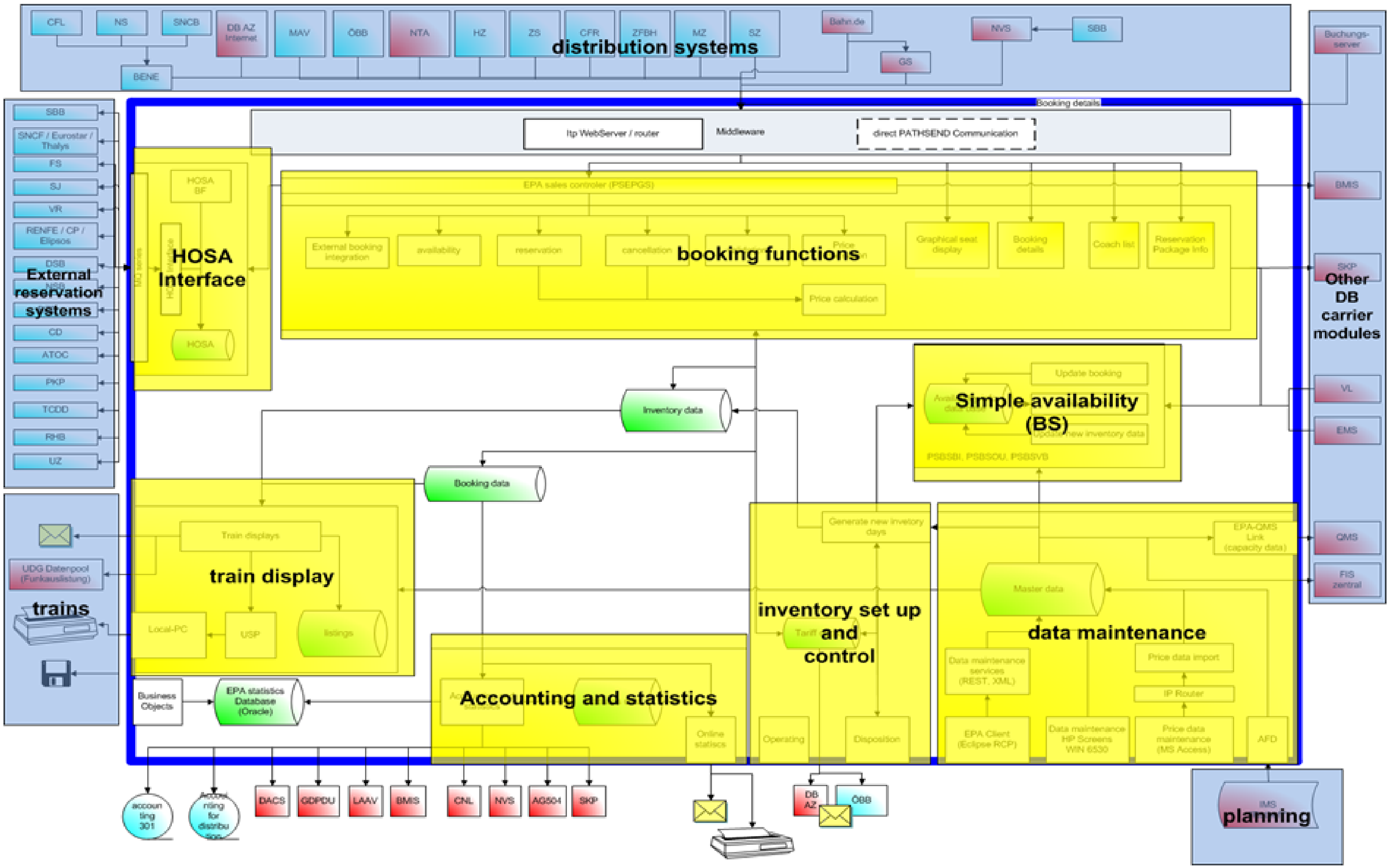
# Motivation

but: difficult to adapt to new, unknown needs

- technological constraints
  - programming languages: C, C++, Cobol, Java
  - DBMS: Enscribe, SQL/MPm, SQL/MX
- specialized hardware
  - tied to *HP NonStop* system
- long update cycles
  - possibly multiple months

**Highly-Available Applications on Unreliable Infrastructure…**

# Motivation



CFL
NS
SNCB
DB AZ Internet
MAV
ÖBB
NTA
HZ
ZS
CFR
ZFBH
MZ
SZ
distribution systems
Bahn.de
NVS
SBB
GS
BENE
Buchungs-server
Booking details
Itp WebServer / router
Middleware
direct PATHSEND Communication
HOSA Interface
booking functions
EPA sales controler (PSEPGS)
External reservation systems
SBB
SNCF / Eurostar / Thalys
FS
SJ
VR
RENFE / CP / Elipsos
DSB
CD
ATOC
PKP
TCDD
RHB
UZ
BMIS
Other DB carrier modules
VL
EMS
QMS
FIS zentral
Inventory data
Simple availability (BS)
Booking data
train display
UDG Datenpool (Funkauslastung)
trains
inventory set up and control
data maintenance
Accounting and statistics
Business Objects
EPA statistics Database (Oracle)
DACS
GDPDU
LAAV
BMIS
CNL
NVS
AG504
SKP
ÖBB
planning

# Motivation

Hasso Plattner Institut

## …Microservices in Practice

- small, independent, autonomous services
- small, specific range of features
- encapsulates all its functions *and* data
- cooperation with other microservices (usually ReST & message queues)
- DevOps

# Motivation

**Aim**: evaluate general properties of a microservice and its dependability compared to the legacy system

1. Benefits & Drawbacks of MSAs
2. Implementing a Seat Reservation System based on Microservices
   - Requirements, Definition of Domains
3. Operation of Microservice Architectures
   - Containerization with Docker, Message-Driven Communication Middleware
4. Evaluation: Dependability & Fault Tolerance

# Benefits and Drawbacks of Microservice Architectures

introduction of self-contained services that deliver, combined, the same functionality as the original system

# Advantages

- small and independent services
  - classification of domains
  - decoupling & explicit separation of features
- free choice of technology
  - use the technology that fits the needs best
  - functionality *and* data
- scalability
  - designed for horizontal scaling – multiple instances
  - requires stateless services
- hardware independence
  - usually self-contained virtual machines

# Advantages

- replaceability & versioning
  - loose coupling among microservices
  - independent testing & deployment
  - redundancy: multiple versions at the same time
- automation
  - many steps for operation only differ in some minor configuration options
- DevOps
  - one single team involved in development (design, implementation, testing, deployment, maintenance) and architectural layers (frontend, backend, database)

# Disadvantages

- complexity
  - from implementation to execution environment
  - provisioning & orchestration of many services
- monitoring
  - service vs. container vs. infrastructure
- testing
  - single service vs. combined services, communication
- communication overhead
  - inter-process & remote
- consistency
  - shared data across service boundaries

# Implementing a Seat Reservation System based on Microservices

modularization into self-contained subsystems with free choice of technology

# Requirements

- functional:
  - display available seats, book a seat reservation, overview of existing bookings
- non-functional
  - consistency, scalability & efficiency, load balancing, portability, deployment & maintainability, changeability, replacement & versioning, interfaces
  - **fault tolerance**
    - tolerate failure of several service instances, virtual machines, or infrastructure components
    - asynchronous communication between services

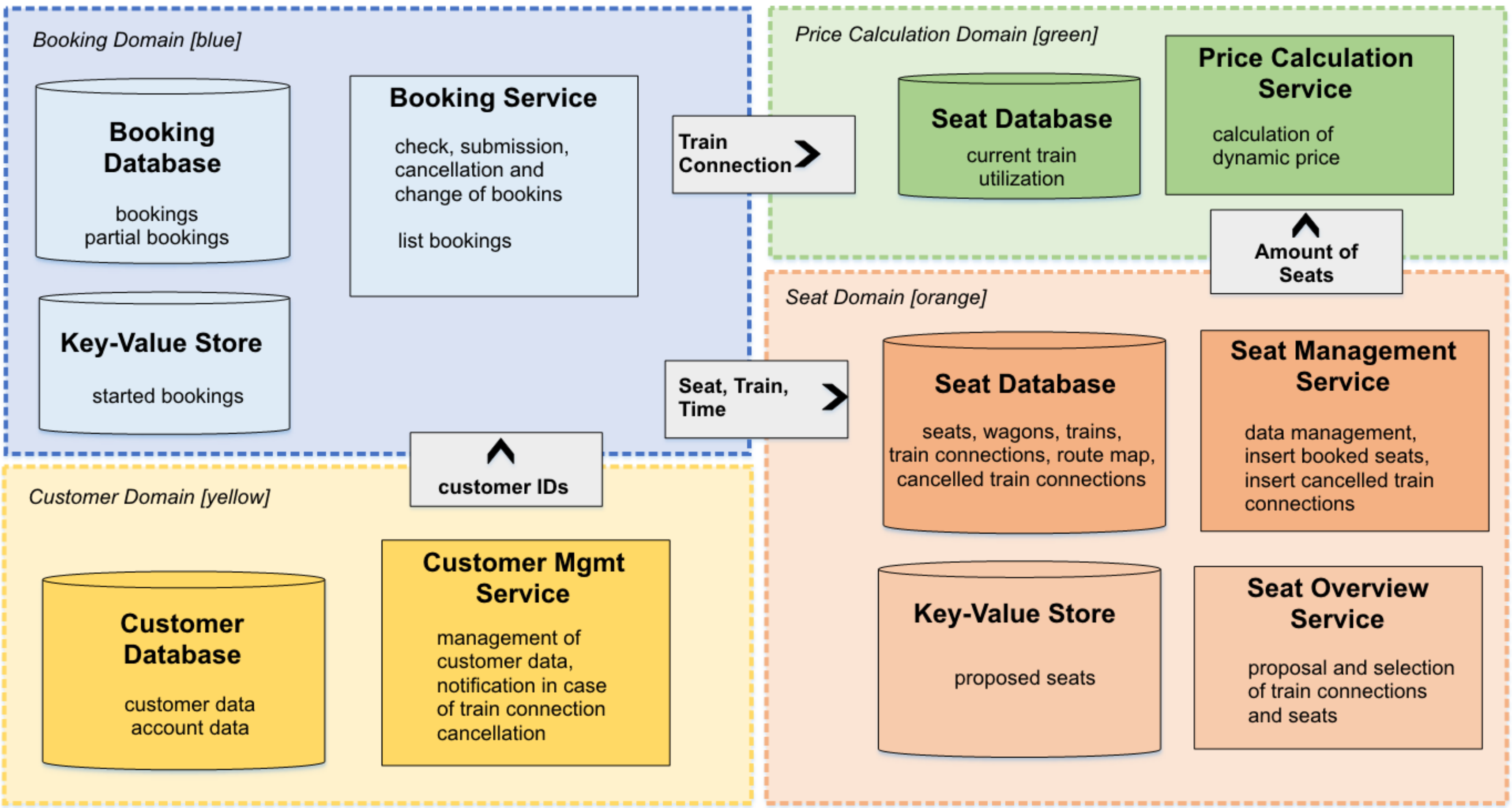
# Definition of Domains

Hasso Plattner Institut

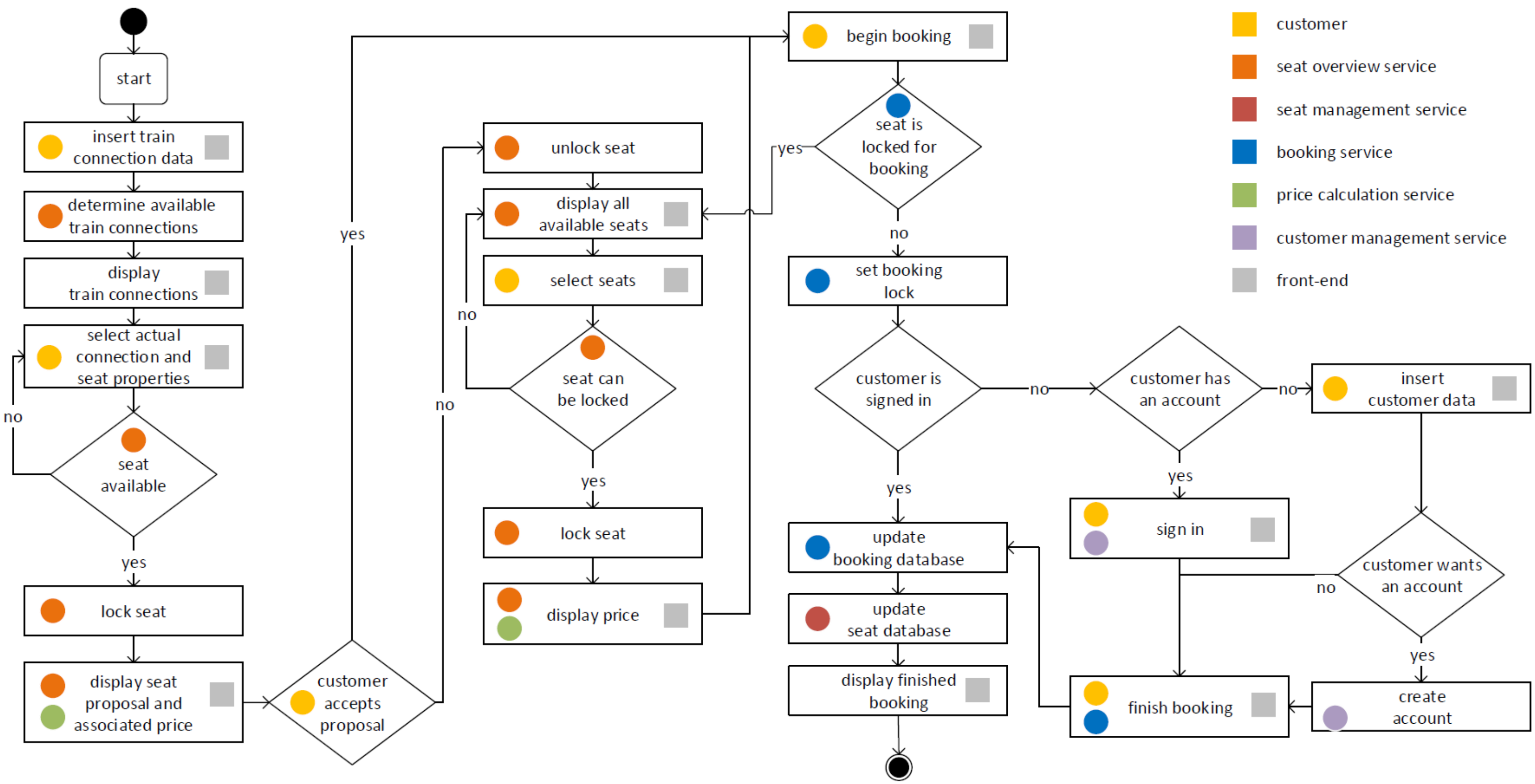
partitioning into functionally connected domains, each domain contains self-contained services with limited scope of operation

- Seat Management Domain
- Seat Overview Domain
- Booking Domain
- Customer Management Domain
- Price Computation Domain
- Front-end

# Definition of Domains



# Domains + Booking Process



# Operation of Microservice Architectures

after their implementation, the microservices, their databases, and the front-end have to be deployed into self-contained environments

# Execution Environment

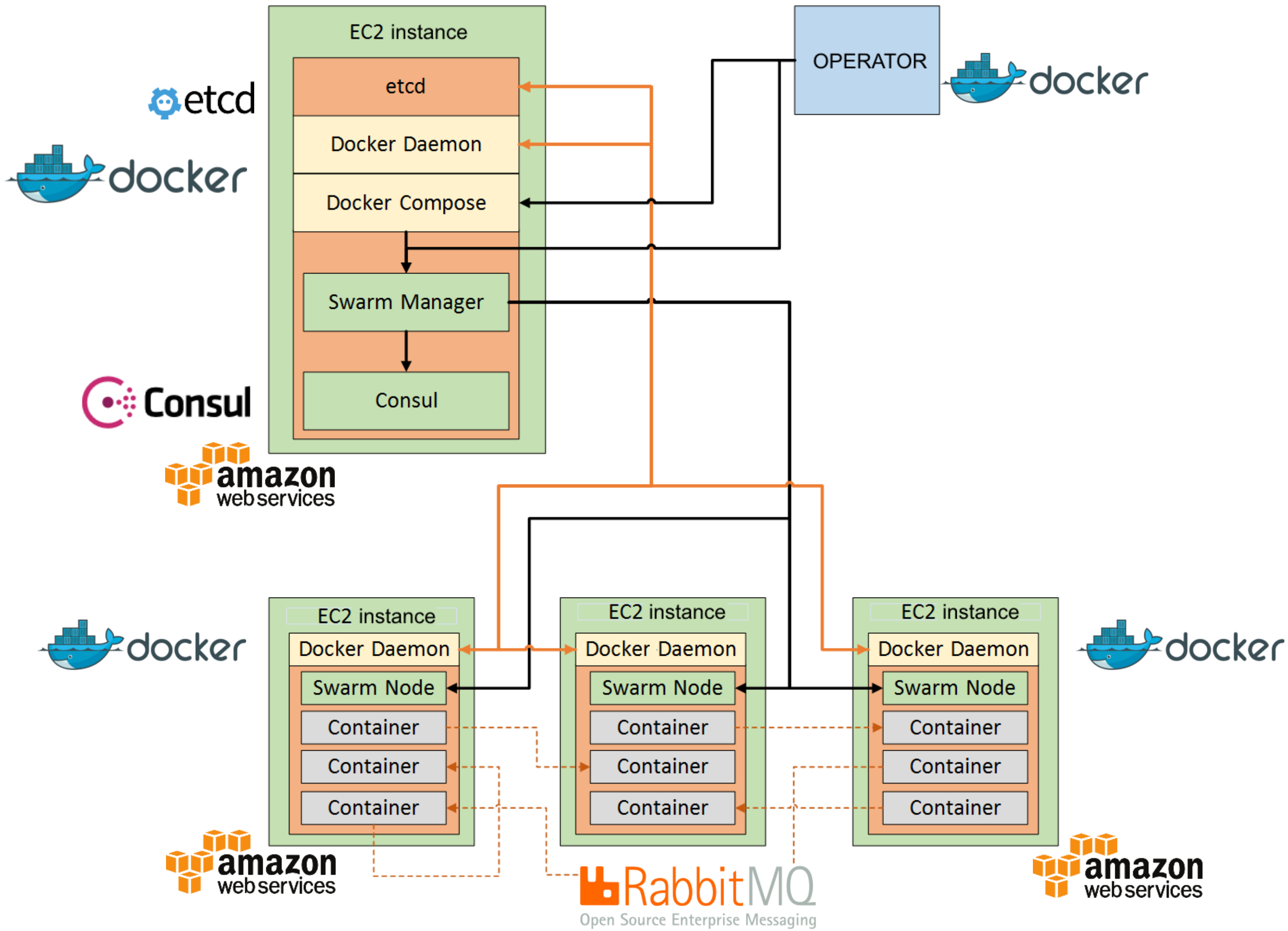


**requirements**: portability, load balancing, fault tolerance, maintainability

- virtualized infrastructure
  - *AWS/EC2* Ubuntu 14.4
- containerization with *Docker* 1.11
  - *Docker Compose*
  - *Docker Swarm*
  - *Overlay Network*
- message-driven communication middleware
  - *RabbitMQ* 3.6.2

# Execution Environment



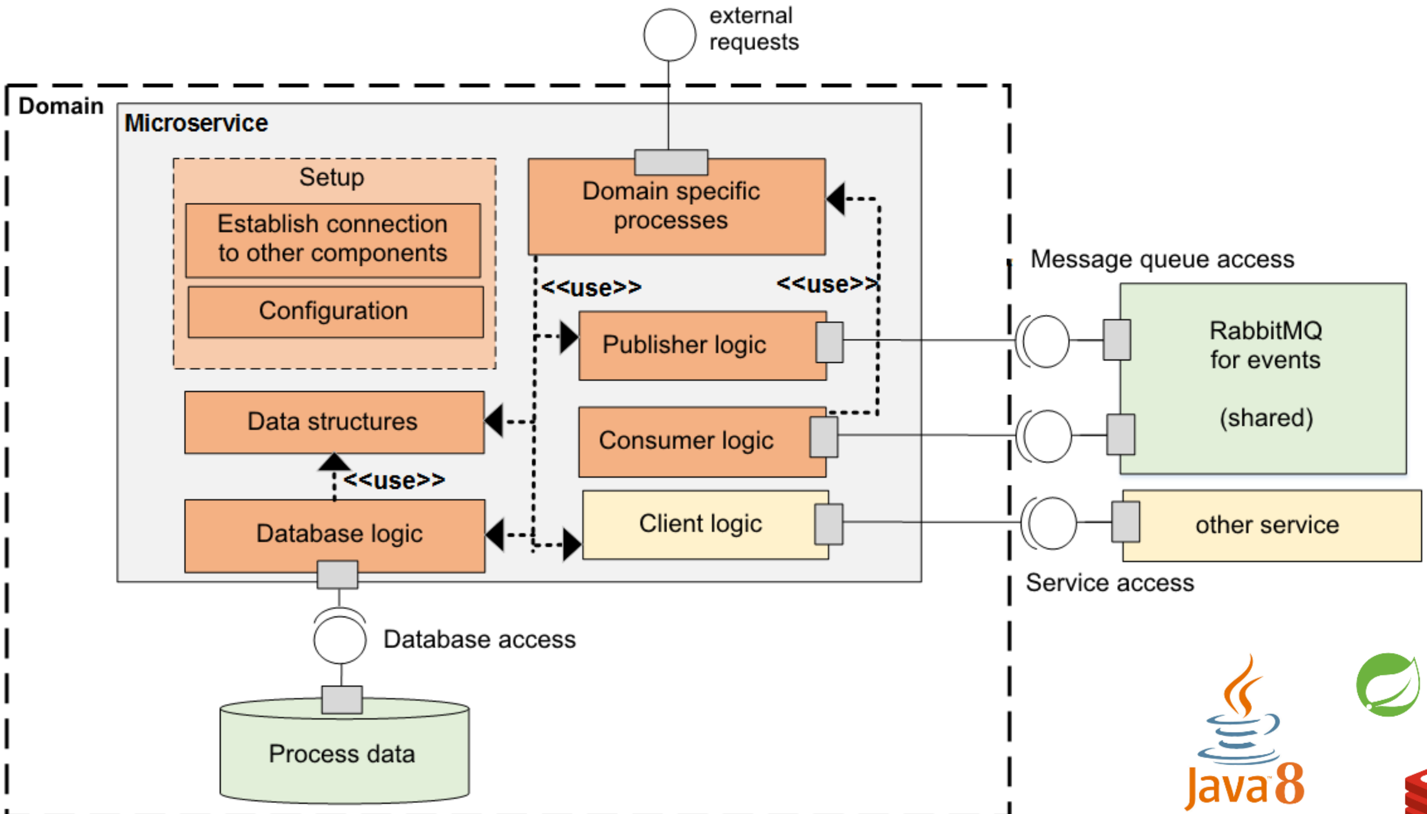


# Execution Environment



- services for seat reservation
  - *Java* 8
  - *Spring Boot* 1.3
  - *MySQL* 5.7
  - *Redis* 3.2
  - *Cassandra* 3.4

# Basic Set-Up of a Microservice




Java 8, spring by Pivotal, redis, MySQL, cassandra

# Evaluation

modularized software system consisting of self-contained services published as containers and executed as multiple redundant instances

# Recap: Requirements

- functional:
  - display available seats, book a seat reservation, overview of existing bookings
- non-functional
  - consistency, scalability & efficiency, load balancing, portability, deployment & maintainability, changeability, replacement & versioning, interfaces

# Dependability & Fault-Tolerance

- instead of relying on specialized (and expensive) highly-available infrastructure:
  modularize the software system into self-contained services published as containers and execution as multiple redundant instances

**Redundancy**

- replicas of services, containers, virtual machines
- communication middleware
- service logic and databases

# Replicas of…

Hasso Plattner Institut

## …services, containers, and virtual machines

- Overlay Network
  - uniform host name, arbitrary number of replicas
  - if service instance, RabbitMQ server, or even EC2 instance fails – redirect to another instance
- Docker Swarm
  - “High Availability” feature: primary manager instance + multiple replica that will take over
  - data storage (etcd, Consul) can be scaled and connected

# Replicas of…

## …services, containers, and virtual machines

- services
  - state-less (state is stored into domain's database)
  - can be replaced by other instances
- messages
  - distributed among all RabbitMQ servers
  - conflict-free merging of message nodes (via master-node)

# Communication Middleware

- message queue is one of the most important parts of the architecture
- tolerated faults: network failure, RabbitMQ server fault, infrastructure failure, malformed messages
- clients can connect do different RabbitMQ servers
- virtual hosts, exchanges, and message queues are synchronized between servers by default

# Service Logic & Databases

- services are state-less – the critical part is the database
- use relaxed consistency guarantees (e.g. NoSQL)
  - Cassandra with multiple replicas
  - MySQL in master-slave-replication mode

# Conclusion

- prototypical architecture and implementation
- freedom to choose any technology is bigger than before; several tools and frameworks for execution environment. but: tied to Docker
- no hardware dependency – fully virtualized infrastructure by AWS
- bring service modifications into production within minutes; architectural changes last a few days
- experience for multiple tools have to be gained; tools, libraries, and frameworks are still in development *and change quickly*

# Conclusion

HPI Hasso Plattner Institut

The results show a potential for microservice architectures and the possibility for flexible implementation, deployment, and advancement of services. In terms of non-functional requirements, the is no evidence that the new solution perform better, though.

We would like to thank Lena Feinbube, Maxi Fischer, Jonas Bounama, Nils Hennings, Timo Traulsen, Henry Hübler, Dr. Stephan Gerberding, Dr. Clements Gantert, Wolfgang Schwab, and Ingo Schwarzer for their support and assistance with this project.