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

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

*Hasso Plattner Institute at University of Potsdam
P.O.Box 90 04 60, D-14440 Potsdam, Germany
Email: {firstname.lastname}@hpi.uni-potsdam.de

†DB Systel GmbH
Caroline-Michaelis-Straße 5-11, D-10115 Berlin, Germany
Email: {firstname.lastname}@deutschebahn.com

***Abstract*—In contrast to applications relying on specialized and expensive highly-available infrastructure, the basic approach of *microservice architectures* to achieve fault tolerance – and finally high availability – is to modularize the software system into small, self-contained services that are connected via implementation-independent interfaces. Microservices and all dependencies are deployed into self-contained environments called containers that are executed as multiple redundant instances. If a service fails, other instances will often still work and take over. Due to the possibility of failing infrastructure, these services have to be deployed on several physical systems. This horizontal scaling of redundant service instances can also be used for load-balancing. Decoupling the service communication using asynchronous message queues can increase fault tolerance, too. The *Deutsche Bahn AG* (German railway company) uses as system called *EPA* for seat reservations for inter-urban rail services. Despite its high availability, the EPA system in its current state has several disadvantages such as high operational cost, need for special hardware, technological dependencies, and expensive and time-consuming updates. With the help of a prototype, we evaluate the general properties of a microservice architecture and its dependability with reference to the legacy system. We focus on requirements for an equivalent microservice-based system and the migration process; services and data, containerization, communication via message queues; and achieving similar fault tolerance and high availability with the help of replication inside the resulting architecture.**

1. Introduction

The *EPA* (“Elektronische Platzbuchungsanlage”) is a system to reserve and book seats inside trains operated in the intercity railway transportation of the *Deutsche Bahn AG* – a German railway company and the largest railway operator and infrastructure owner in Europe. EPA holds information about available train connections and seat reservations. The first version was deployed in the early 1980s – since then, the requirements for transportation systems changed and the traffic volume increased; simultaneously, customers demand for mobility and flexibility. Currently, EPA processes 1,000,000 requests for available seats, 300,000 bookings, and 50,000 requests for existing bookings per day.

EPA – The Legacy System. EPA is implemented as a set of *Pathway Services* as part of a *HP NonStop* system. Though the system is especially fault-tolerant and highly-available, it has several disadvantageous properties.

Technological constraints. NonStop systems currently support the programming languages C, C++, COBOL, and Java; and the database systems Enscribe, SQL/MPm, and SQL/MX. EPA is mainly written in COBOL and C. When adding new features to the EPA system, the development team is tied to these technologies – but it is not guaranteed that new features can be implemented easily and efficiently with within the given constraints. *Specialized hardware.* To ensure fault tolerance and high availability, the EPA system is tied to the NonStop system with its strongly-coupled hardware and software components. Currently, it is not possible to migrate the EPA system as-is or parts of it to less expensive, supplier-independent hardware and to keep the current dependability at the same time.

Long update cycles. To ensure software dependability and prevent errors and failures during runtime, the whole EPA system is tested extensively after changing the code-base. Hence, it takes a long time before e.g. bug-fixes or new features come to production – possibly multiple months.

Due to these properties, it is difficult to adapt the EPA system to new, unknown needs in an easy and straightforward way. In favor of shorter response times and agile projects for modular applications and data, the out-of-date core systems with manual processes should be replaced. With a microservice architecture, services and their data can be implemented, deployed, and maintained independently by small teams.

Microservice Architectures. We evaluated the use of *Microservice Architectures* in context of the legacy system. Microservices are small, independent, and autonomous services designed for a small, specific range of features and form a complex application in cooperation with other microservices. [1] The size of microservice is not specified precisely: a service should be split into more services if it is too complex and confusing for developers [2] and it should be "small enough" [3]. A microservice encapsulates all of its functions as well as its data in a functionally well-defined domain and provides access to external components via an API. Microservices often provide a ReST (Representational State Transfer) API. In this way, the functionality of a microservice becomes self-contained. The communication between microservices is usually implemented with message queues – each service subscribes to a specific message queue in case it is interested in events of other services. Microservices can be deployed with small configuration overhead to scalable infrastructures such as the cloud.

The aim of this work is to evaluate the general properties of a microservice architecture and its dependability compared to the legacy system.

The paper is structured as follows: section 2 explains the benefits and drawbacks of microservice architectures; in section 3 we describe the requirements for a seat reservation system based on microservices and our definition of the corresponding domains; section 4 provides detailed information about the implementation and operation of our microservice architecture, the containerization with Docker and the message-driven middleware; finally, section 5 evaluates the realization of requirements and focuses on fault tolerance of services, containers, virtual machines, and the communication middleware.

2. Benefits and Drawbacks of Microservice Architectures

The migration to a microservice architecture implies the introduction of several self-contained services that, combined, deliver the same functionality as the original system. A microservice architecture provides the following **advantages**: [2], [4], [5], [6], [7], [8]

Small and independent services. A microservice represents only a part of the whole systems functionality. The size of a microservice is based on the domain it covers. The classification of specific domains leads to the decoupling and explicit separation of features – intra-component dependencies (e.g. referencing classes and functions in other scopes) are avoided. Because of the self-contained fashion of domains and the communication via explicit interfaces, the inner structure is not relevant for external components; service implementations do not affect other services. Instead of changing, testing, and deploying the entire system, only a subset has to be taken into account. The development of new features can focus on few services and changes usually can be deployed faster than in a system without separated domains.

Free choice of technology. With independent services it does not matter how the services are implemented; it is possible to choose the technology that fits the needs for that specific functionality best. That also applies to the database technology – every service can choose its own data model and database management system. It is not necessary to be tied to a technology that was chosen based on requirements for other features.

Scalability. It is not assumed that one service instance can manage requests of unknown number. To deal with heavy load, microservices usually are designed for horizontal scaling – multiple instances of a service are in use and a load balancer distributes requests to all available service instances. Scalability of microservices requires stateless services that do not save any data outside of a database associated with the service's domain (steady state).

Hardware independence. Microservices usually are independent processes that can be executed as self-contained virtual machines [2] – typically, there is no need for special hardware. Commercial-off-the-shelf hardware can be used.

Replaceability & versioning. Because of loose coupling among themselves, services can be tested and deployed independently. Under the assumption of unchanged (external) interfaces, services can be replaced by other versions. In this way, it is also possible to deploy new versions of a service successively and watch their behavior under real conditions. Due to redundancy, it is possible to deploy a new version only for a part of requests. If the result is satisfactory, the service's old version can be replaced completely. Otherwise, it is also possible to replace new versions with an older but more reliable one. Another effect of stepwise updates is the possibility to run the whole system continuously without downtime.

Automation. The operation of microservice architectures usually contains the management of the underlying infrastructure. Many tools are available for tasks such as running services in virtualized environments, communication between services and parts of the infrastructure, hardening against failures, and monitoring. Many of the steps necessary for operation will be executed multiple time and only differ in some minor configuration options. These steps offer the potential for automation and reduce the probability of human made faults.

DevOps. A term often mentioned in the microservice context is *DevOps*: the combination of developers and operators. Because of the breakdown into small services in separated domains, it is possible to have one single team that is involved in all steps of the development process (such as design, implementation, testing, deployment, maintenance) and architectural layers (such as front-end, back-end, database). The purpose is an efficient collaboration and increased quality of results.

Like other distributed systems, microservice architectures show some **disadvantages**, too:

Complexity. The system's complexity is transferred away from the service implementation to the execution environment. Development and production are more complex due to provisioning and orchestration of services. Every service should be started, stopped, and scaled independently without affecting other services.

Monitoring. In case a faulty service needs to be located and debugged, the log data of several components (such as the service itself, the container, or the infrastructure) has to be found, aggregated, analyzed, and assigned correctly.

Testing. Compared to monolithic applications, new strategies for testing have to be found, where each service for itself, services in combination with other services, the communication between services and the behavior as a whole system have to be examined.

Communication overhead. While monolithic applications mainly send requests within intra-process boundaries, microservices send messages between multiple processes and to remote services. This possibly leads to higher latency and more overhead for communication.

Consistency. Another challenge is to preserve the consistency of shared data across service boundaries. Because of the quasi need for redundantly operating multiple service instances, shared data is not avoidable in most cases. The use of NoSQL database with a relaxed consistency model respectively eventual consistency is a common strategy in microservice architectures.

3. Implementing a Seat Reservation System based on Microservices

Microservice architectures follow another approach than monolithic applications: modularization into self-contained subsystems. Through free choice of technology, hardware independence, and the individual handling of services it is possible to prevent the currently existent inflexibility of the EPA system and to satisfy the requirements for modern and modular software.

3.1. Requirements

Besides the apparent functional requirements for a seat reservation system with the three main responsibilities *display of available seats*, *booking of a seat reservation*, and *overview of existing bookings* (we omit details here), the formerly defined properties of microservice architectures can be transferred to the following non-functional requirements:

1) Consistency: Avoidance of same available seats offered to different customers. Prohibition of multiple bookings for the same seat.
2) Fault Tolerance: Tolerance of failure of several service instances, virtual machines, or infrastructure components. Asynchronous communication between services.
3) Scalability & efficiency: Horizontal scalability of services, virtual machines, and servers. Usage of the minimal amount of necessary resources in case of low degree of capacity utilization (one instance per component); no upper boundary for extensive load.
4) Load balancing: Steady distribution of external and internal service requests to all available service instances.
5) Portability: Possibility to execute a service on a different server, on a different operating system, and on infrastructures of different cloud providers.
6) Deployment & maintainability: Minimization of need for human intervention and maximization of automation for deployment, start, stop, and scaling of components.
7) Changeability: Free choice of used technologies, tools, and programming languages.
8) Replacement & versioning: Update of services in operation without downtime. Operation of different versions of a service at the same time without affecting other versions.
9) Interfaces: Standardized communication between services via HTTP and JSON. [9] Testing of interfaces – a service that does not fulfill its interface definition is not allowed to be deployed.

3.2. Definition of Domains

One of the key aspects of microservices is the partitioning of the whole system into functional connected domains that help to introduce loose coupling. Each domain contains self-contained services with limited scope of operation; services and all dependencies are deployed into self-contained environments (*containers*) and are connected via implementation-independent interfaces to a complete software system. A domain's services and data can be implemented, deployed, and maintained independently by small teams.

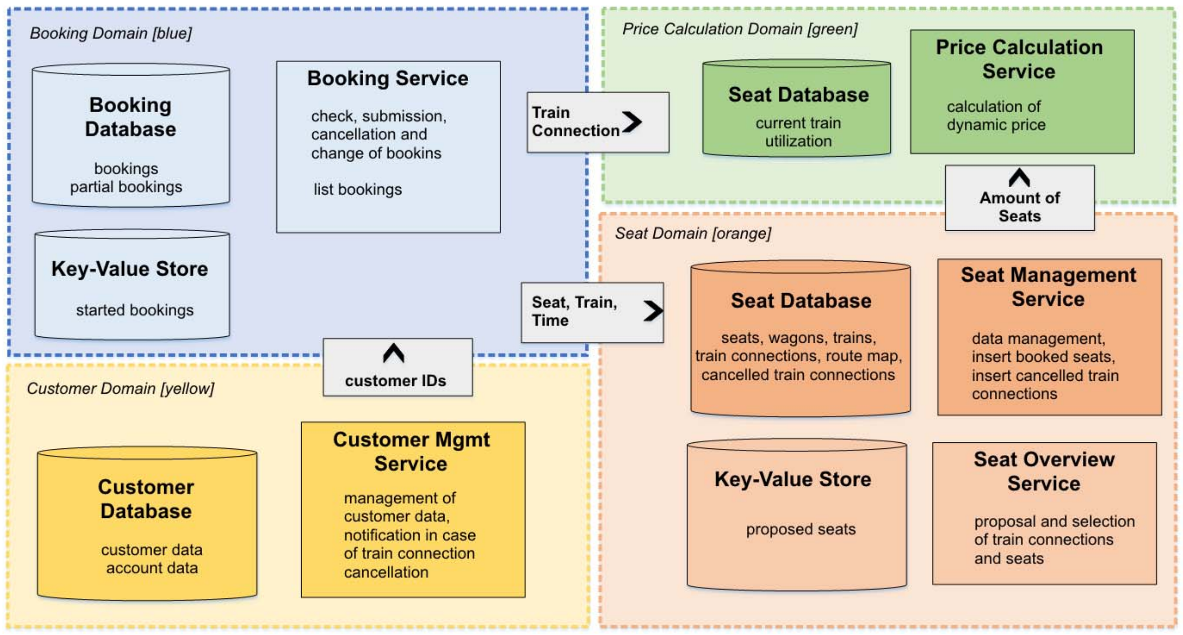
Based on the functional requirements for the seat reservation system and a customer's view, we identified the following domains and their contained services and data (see Figure 1):

3.2.1. Seat Management Domain. An integral part of the planned software system are seats in trains. One main function is to list all available seats in a train for a specific connection and to offer seat reservations to customers. To satisfy these needs, the seat management service can create, read, update, and delete entries of the seat database – this services provides the data source for seat reservations. Each seat is unique and assigned to a specific railway carriage that is related to a specific train connection at a specific date.

After successful seat booking, the corresponding booking service sends a notification to the seat management service about then unavailable seats and the seat database is updated.

3.2.2. Seat Overview Domain. The seat overview domain is responsible for all necessary data related to a seat reservation. Customers can request available seats for specific train connections and apply for a reservation. This domain is tied to the display of data – it is not possible to change available data. For this purpose the seat overview service works on a read-only copy of the seat management domain's database.

To guarantee the scaling in case of higher load without having to deal with increasing synchronization overhead



Booking Domain [blue]
Booking Database
bookings
partial bookings
Booking Service
check, submission, cancellation and change of bookins
list bookings
Key-Value Store
started bookings
Train Connection
Price Calculation Domain [green]
Seat Database
current train utilization
Price Calculation Service
calculation of dynamic price
Amount of Seats
Seat Domain [orange]
Seat, Train, Time
Seat Database
seats, wagons, trains, train connections, route map, cancelled train connections
Seat Management Service
data management, insert booked seats, insert cancelled train connections
customer IDs
Customer Domain [yellow]
Customer Database
customer data
account data
Customer Mgmt Service
management of customer data, notification in case of train connection cancellation
Key-Value Store
proposed seats
Seat Overview Service
proposal and selection of train connections and seats

Figure 1. The chosen architecture consists of for domains that contain services and data: Seat Overview Domain (orange) containing a service to manage train & seat data and a service for fast identification of available seats; the Booking Domain (blue) containing a service to book a seat and all data concerning booking; Customer Management Domain (yellow) containing a service to manage customer data; and the Price Computation Domain (green) containing data about the available seat capacity and a service for dynamic price computation.

between the domains, the consistency guarantees have been relaxed. Hence, the read-only copies will be synchronized only in specific time intervals. To avoid the offering of the same available seats to different customers at the same time, there is a special temporary lock mechanism that works with a high-performance key-value store.

3.2.3. Booking Domain. The booking domain with its booking service and booking database stores and manages successful seat reservations. Similarly to the seat overview domain, a special lock mechanism prevents the booking of the same seat for different customers.

3.2.4. Customer Management Domain. Customer data – such as address and payment information – is stored into the customer database that belongs to the customer management domain. Also, the customer management service is responsible for the authentication of users.

3.2.5. Price Computation Domain. To complete a booking, the price for a seat reservation has to be calculated. The price computation service is used by the seat overview service and calculates the actual price bases on the chosen train connection and the capacity of available seats.

3.2.6. Front-end. The front-end is a web application that implements graphical controls to e.g. search for train connections, display available seats, perform a seat reservation, and display existing bookings. It transforms the user's input into HTTP requests containing JSON objects, interacts with corresponding services, and displays received responses.

After the assembling of the domain's designed services and its databases, a process for a complete booking as shown in Figure 2 is possible.

3.3. Monitoring

The collection and evaluation of logging information is important for the operation of microservice architectures. The goal is to gather as much information as possible to have a good overview about the system's current state as well as the possibility for a fast root cause analysis in case of failures. Our project uses a combination of tools such as *LogStash*,[1] *Elasticsearch*,[2] and *Kibana*.[3] LogStash extracts relevant information out of diverse logging sources. Extracted data will be indexed and stored in Elasticsearch. Kibana is used to access the stored data, process it, and create visualizations. The monitoring of virtual machines is done with the help of *Sensu*,[4] which gathers information such as processor utilization periodically.

4. Operation of Microservice Architectures

After their implementation, the microservices, their databases and the front-end have to be deployed and operated. The

1. https://www.elastic.co/products/logstash
2. https://www.elastic.co/products/elasticsearch
3. https://www.elastic.co/products/kibana
4. https://sensuapp.org/

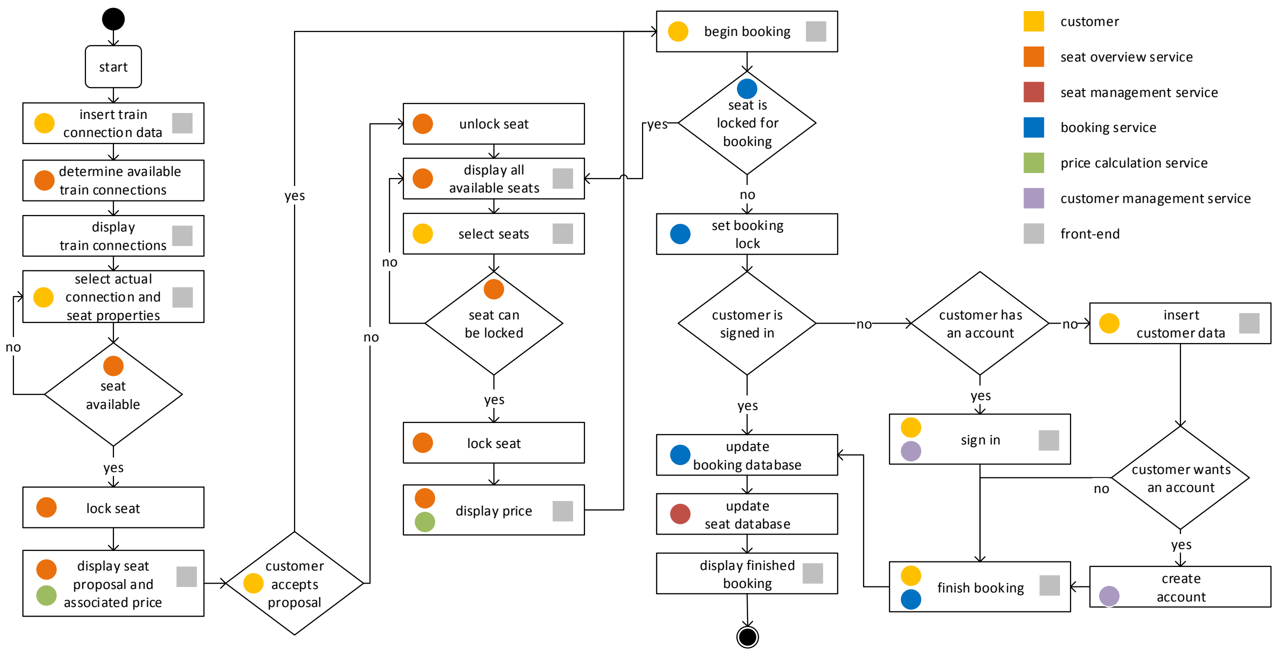


Figure 2. The process for seat booking associated with the corresponding services and the front-end.

challenge of the execution environment is to deal with the non-functional requirements such as portability, load-balancing, fault tolerance, or maintainability.

As a foundation for our execution environment, we decided to use *Amazon Web Services (AWS)*,[5] a collection of offerings for cloud computing, and used *Elastic Compute Cloud (EC2)*[6] instances, where scalable virtual machines with several hardware configurations and dynamic resources are available. To decrease the amount of time needed to bring a virtual machine into service, a pre-configured virtual machine image with all necessary software dependencies and configurations is used.

4.1. Containerization with Docker

The execution environment should be able to start services with as little prior configuration as possible. Thus, we use containerization – each service with its data and dependencies is wrapped into a separate container. These containers need less space than a virtual machine. It is possible to deploy multiple containers at the same time to a EC2 instance without the necessity to deploy software dependencies to the virtual machine itself. Besides the smaller memory footprint, these containers can be started and stopped faster than virtual machines – instead of a whole operating system only isolated processes have to be started and stopped. [3], [10], [11], [12]

5. https://aws.amazon.com/de/
6. https://aws.amazon.com/de/ec2/

Several tools are available for containerization. Prominent examples are *Docker*,[7] *CoreOS*,[8] and *LXD*;[9] these tools are based on the Linux kernel and offer features to create, operate, and maintain containers. At the time of writing, *Docker* was the most popular containerization tool, had good documentation, and extensions for operation and administrations such as *Docker Compose*[10] and *Docker Swarm*[11] were available.

After installing Docker, the host system (*Docker Host*) containing a so called *Docker Daemon* is active. This component is able to start containers by request, based on an container image. The container image itself is configured with the help of a *Dockerfile* – a script file containing instructions (e.g. copy files, execute commands) to define the environment in which the desired application is executable. Every container can get an additional, individual configuration.

Docker Hosts can contain several network interfaces to run containers isolated from other networks. In this way, each container is assigned an own IP address. The Dockerfile can define internal ports to communicate with other containers inside the network. To grant access for external services, an internal port has to be exposed as an external port.

Docker implements its own DNS (Domain Name System) server with integrated load-balancing to address containers and services; Multiple Docker Hosts can be connected via a so called *Overlay Network*. This eases the communication, because for requests it is not necessary to know the exact

7. https://www.docker.com/
8. https://coreos.com/
9. https://linuxcontainers.org/lxd/
10. https://docs.docker.com/swarm/overview/
11. https://docs.docker.com/compose/overview/

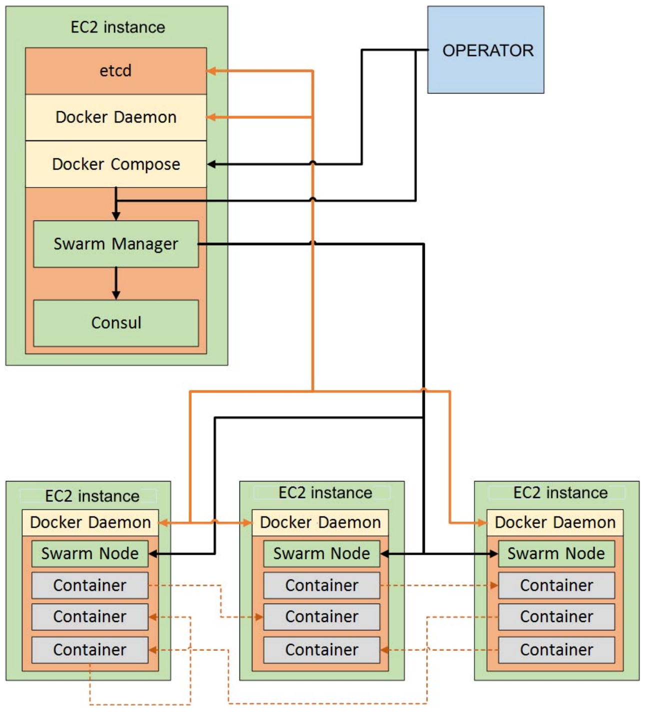


Figure 3. The resulting architecture with Overlay Network, Docker Swarm, and Docker Compose. Each EC2 instance belongs to the swarm (green). Docker Daemons use a connection to etcd to create an Overlay Network (orange, solid), so services can communicate with each other (orange, dashed). The Swarm Manager provides an interface (via Swarm Manager) to the operator, so Docker Compose can distribute containers to Swarm Nodes or get information about the swarm. Status information is stored into a Consul instance.

location of a target container or service (as long as they belong to the same Overlay Network). So dedicated service discovery tools are not required. The Overlay Network uses *etcd*[12] as external key-value store.

With Docker Swarm, it is possible to manage multiple Docker Hosts. A Swarm contains two roles: *Swarm Managers* and *Swarm Nodes*. A Swarm Manager orchestrates and schedules containers on the entire Swarm; it stores status data in a *Consul*[13] server instance running inside a container. Nodes are additional Docker Hosts. Adding or removing Docker Hosts to or from a Swarm enables a dynamic scaling of computing power – invisible to external components. The distribution of containers to members of the swarm is transparent, too.

Docker Compose helps do establish complex applications that consist of multiple containers. A Docker Compose definition describes required containers and configurations; multiple instances with the same configuration are possible. Thereby, all running instances form a unit that is addressable with the same host name.

The resulting architecture is shown in Figure 3.

12. https://github.com/coreos/etcd
13. https://www.consul.io/

4.2. Message-Driven Communication Middleware

To ensure a reliable communication between loosely coupled services that is fault-tolerant against failures of software, hardware, and the network without losing messages, we use a message-driven middleware. [13], [14], [15]

RabbitMQ[14] is an open source solution that fits these needs. RabbitMQ implements the *Advanced Message Queueing Protocol (AMQP)*. AMQP consists of a functional and a transportation layer; the functional layer defines the components *Exchange*, *Message Queue*, and *Binding*. An Exchanges accepts messages and distributes them based on Bindings to a Message Queue. [16], [17]

A *RabbitMQ Server* is a service that accepts client connections and that implements AMQP and methods for message forwarding. A *RabbitMQ Client* is an application that connects to the RabbitMQ Server and publishes or receives messages. A RabbitMQ Server can contain virtual hosts that orchestrate Exchanges, Message Queues, and associated objects (such as messages) and create a boundary to other virtual hosts on the same host. RabbitMQ can be deployed as Docker Container and connected to a cloud via the service *Cloud AMQP*[15]. The resulting architecture using RabbitMQ looks as follows: RabbitMQ Servers are started via Docker Compose on each Docker Host. Internally, each RabbitMQ Server has a unique host name. External services can address all RabbitMQ servers via a common name (rabbit), so internal host names are not relevant for communication.

5. Evaluation

The developed microservice architecture with its execution environment using Amazon Web Services (with Ubuntu 14.4), Docker 1.11, RabbitMQ 3.6.2 and services for the seat

14. https://www.rabbitmq.com/
15. https://www.cloudamqp.com/

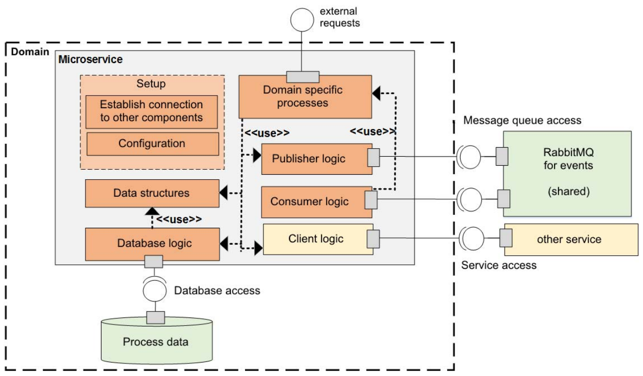


Figure 4. Basic set-up of a microservice – A shared message queue is used for communication. Data is contained by the domain. A service needs a connection to the message queue for sending messages (publish logic), data (data logic), and other services (client logic). Messages are received via client logic and processed in domain specific logic.

reservation using technologies such as Java 8 with Spring Boot[16] 1.3, MySQL[17] 5.7, Redis[18] 3.2 and Cassandra[19] 3.4 are available online.[20].

5.1. Non-Functional Requirements

Revisiting subsection 3.1, the stated non-functional requirements are satisfied as follows:

1) Consistency: Due to a special lock mechanism should the same available seat should not be displayed to different customers and a booking of the same seat for different customers is prevented.
2) Fault Tolerance: See subsection 5.2 in detail.
3) Scalability: Due to the usage of Docker Compose it is possible to scale all implemented services and databases. The parts of the infrastructure – such as the key-value stores etcd and Consul (used to store the state of the Swarm), as well as Swarm Managers and Swarm Nodes can be scaled and connected to each other.
4) Load Balancing: The used Overlay Network has integrated load-balancing. Currently, the algorithm in use is implemented internally by Docker and cannot be changed.
5) Portability: The only existing hard dependency is Docker. As long as the target system supports Docker, containers can be deployed to any hardware, operating system, or cloud provider.
6) Deployment & maintainability: To reduce the need for human intervention and to increase the automation, virtual machine images with all dependencies installed and several scripts for starting and stopping virtual machines, containers, and services are available. With Docker Compose and Docker Swarm services can be deployed automatically.
7) Changeability: The free choice of technology is achieved only partially: The current architecture is tied to RabbitMQ (or rather ASQP) as communication middleware. As a consequence of the requested portability property, the execution environment currently is bound to Docker.
8) Replacement & maintainability: An uninterruptible operation while deploying new versions as well as the simultaneous operation of different versions of services at the same time is currently not implemented.
9) Interfaces: Services communicate asynchronously via JSON over HTTP. However, the conformance to this requirement affects other requirements like the free choice of technologies.

16. http://projects.spring.io/spring-boot/
17. https://www.mysql.com/
18. https://redis.io/
19. https://cassandra.apache.org/
20. https://github.com/dbsystel/ReservierungMicroservicesDienste, https://github.com/dbsystel/ReservierungMicroserviceSetup

5.2. Dependability & Fault-Tolerance

In contrast to applications relying on specialized (and expensive) highly-available infrastructure, the basic approach of microservice architectures to achieve fault tolerance – and finally high availability – is to modularize the software system self-contained services published as containers and executed as multiple redundant instances. If a service fails, other instances will still work and take over. This horizontal scaling of redundant service instances can also be used for load-balancing. Decoupling the service communication using asynchronous message queues can increase fault tolerance, too.

5.2.1. Replicas of Services, Containers, and Virtual Machines. On the strength of the Overlay Network, different instances of the same component are addressed by one uniform host name. The number of replicas can be arbitrarily chosen and instances can be substituted. In case one or more service instances, the RabbitMQ Server, or even an EC2 instance fail, other services do not notice the failure – they are redirected to another instance associated with the host name (as long as there is at least one operational instance).

In Docker Swarm, the Swarm Manager is responsible for the entire cluster. The *High Availability* feature allows a Docker Swarm create a single primary manager instance and multiple replica instances that will take over if the primary manager fails. The used key-value stores etcd and Consul (storage of the state of the Swarm) can be scaled and connected to each other.

The services themselves are implemented state-less – state is stored into a domain's database. In this way, service instances can be replaced by other service instances without data loss.

Messages are distributed and replicated among all RabbitMQ Servers, so that messages do not get lost. During runtime, services may connect to other available RabbitMQ Servers. In the event of a network partition, RabbitMQ offers the probability of conflict-free merging of message queues by choosing a master node – child nodes have to synchronized with the master node before they can act as a message broker. In case a master node fails, only a fully-synchronized child node can become a master.

Before services start executing, all dependent components are checked for their availability. If a component is not available, the process is not executed. In case the connection cannot be established over a longer time period, a monitoring component will be notified.

However, fault tolerance of the communication middleware and the services is limited: When a service cannot connect to the message queue, the service is not state-less for a small period of time since messages are stored into a thread-local storage until delivery. If the service fails within this time period, the message is lost.

5.2.2. Communication Middleware. The message queue is one of the most important parts of the architecture – if the message queue is not available, neither the publishing nor the receipt of messages is possible.

The following faults should be tolerated: network failure (no way to communicate with RabbitMQ servers or clients), the failure of a RabbitMQ Server (internal fault), and infrastructure failure (e.g. crash of an EC2 instance), and malformed messages. Based on this fault model, in a RabbitMQ-based architecture multiple RabbitMQ Servers must be available and distributed amongst different EC2 instances. RabbitMQ servers must be connected and gain access to message queues via multiple network interfaces.

Services and consumers can be connected to different RabbitMQ server – it does not matter to which server messages are published. Virtual hosts, Exchanges, and Message Queues are distributed and synchronized between multiple servers by default.

5.2.3. Service Logic and Databases. Services are state-less and can be replicated arbitrarily. The critical component is the database. Modern database management systems support replication of databases. To ensure availability, an approach could be to relax consistency guarantees and e.g. use NoSQL databases which apply eventual consistency.

The seat management system uses a Cassandra database deployed with three replicas. The relation MySQL database is operated in master-slave-replication mode.

6. Conclusion

Compared to the original EPA system, a seat reservation system based on a microservice architecture has several advantages: The freedom to choose any preferred technology considerably bigger than before, particularly, the opportunities for the implementation of domain logic and databases. For the execution environment, several tools and frameworks are available, too. A complete unrestricted choice of technology could not be achieved, though.

During the project, no dependencies to hardware were detected – the infrastructure was fully virtualized by Amazon Web Services.

The timespan to bring new features into production is also presumably lower than before. Service modifications could be brought into deployment within minutes; architectural changes last usually few days, because experiences with the used tools had to be gained before. However, the adoption of a microservice architecture caused bigger effort in context of the used tools and frameworks – every small part of the whole system had to be examined individually. A lot of tools, libraries, and frameworks are still in development and change quickly, which makes development difficult, too.

We have shown that fault tolerance can be achieved through redundancy on many levels, without the need for special and expensive infrastructure. Within the scope of this project the designed architecture and implementation is only prototypical. An extensive evaluation of the capability and performance could not be performed. Nevertheless, 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.

Acknowledgments

The authors 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.

References

[1] Farcic, Viktor, "The DevOps 2.0 Toolkit - Automating the Continuous Deployment Pipeline with Containerized Microservices." dpunkt.verlag, 1 2016, pp. 121–143.

[2] E. Wolff, *Microservices: Grundlagen flexibler Softwarearchitekturen*, 1st ed. dpunkt.verlag GmbH, 2015.

[3] K. T. Seo, H. S. Hwang, I.-Y. Moon, O.-Y. Kwon, and B.-J. Kim, "Performance comparison analysis of linux container and virtual machine for building cloud," *Advanced Science and Technology Letters*, vol. 66, no. 105, p. 2, 2014.

[4] C. H. Gammelgaard, *Microservices in .NET Core*. Shelter Island, NY: Manning, Sep. 2016.

[5] Fowler, Martin. (2014, Mar.) Microservices - a definition of this new architectural term. [Online]. Available: http://martinfowler.com/articles/microservices.html

[6] Newman, Sam, *Building Microservices*, 1st ed. O'Reilly and Associates.

[7] S. J. Fowler, *Production-Ready Microservices: Building Standardized Systems Across an Engineering Organization*. O'Reilly UK Ltd., Dec. 2016.

[8] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, *Site Reliability Engineering: How Google Runs Production Systems*, 1st ed. O'Reilly UK Ltd., Apr. 2016.

[9] N. Nurseitov, M. Paulson, R. Reynolds, and C. Izurieta, "Comparison of JSON and XML Data Interchange Formats: A Case Study." *Caine*, vol. 2009, pp. 157–162, 2009.

[10] Merkel, Dirk, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014.

[11] Mouat, Adrian, "Using Docker - Developing and Deploying Software with Containers, Chapter 3.3: Building Images from Dockerfiles." O'REILLY, 1 2016, p. 25.

[12] H. Kang, M. Le, and S. Tao, "Container and Microservice Driven Design for Cloud Infrastructure DevOps," in *2016 IEEE International Conference on Cloud Engineering (IC2E)*, Apr. 2016, pp. 202–211.

[13] O. Levina and V. Stantchev, "Realizing event-driven SOA," in *Internet and Web Applications and Services, 2009. ICIW'09. Fourth International Conference on*. IEEE, 2009, pp. 37–42. [Online]. Available: http://ieeexplore.ieee.org/abstract/document/5072495/

[14] M. Eisele, *Developing Reactive Microservices*. O'Reilly Media, Inc., Jun. 2016. [Online]. Available: http://proquest.tech.safaribooksonline.de/9781491975640

[15] J. Bonr, *Reactive Microservices Architecture*. O'Reilly Media, Inc., Apr. 2016. [Online]. Available: http://proquest.tech.safaribooksonline.de/9781491975664

[16] Vinoski, Steve, "Advanced message queuing protocol," *IEEE Internet Computing*, vol. 10, no. 6, p. 87, 2006.

[17] Pivotal Software, Inc. (2008, Nov.) Amqp advanced message queuing protocol. [Online]. Available: https://www.rabbitmq.com/resources/specs/amqp0-9-1.pdf