Keynote Speech on Microservices
– A modern, agile approach to SOA –

Andreas Hausotter
University of Applied Sciences and Arts, Hannover
Faculty of Business and Computer Science
Agenda

Chapter 1  Introduction

Chapter 2  Architectural Tasks and Challenges

Chapter 3  Patterns for Resilience and QoS

Chapter 4  Applications and Examples

Chapter 5  Technology Solutions

Chapter 6  Summary and Conclusions
Microservices – Introduction

Motivation – A common scenario for a web application

Online shop system with basic functionalities:

- Search for products (e.g. by name and/or category),
- view product details (including pictures etc.),
- purchase products (place in basket, proceed to checkout) and
- submit and view product reviews.

Typical Requirements:

- Interoperability: Support a variety of different clients (web browser, mobile applications etc.)
- Maintainability: Enable frequent and rapid changes
- Scalability: Handle sudden increases in user activity
- Availability: Minimise downtime (= financial loss)

→ Traditional Approach: Monolithic Architecture
Microservices – Introduction

Monolithic Architecture – The traditional approach to web applications

Properties:

• Single process
• Single database

Advantages:

• Easy development (for example, communication via simple method calls)
• Easy deployment (deployment of a single artefact)
• Application as a whole is scalable (via load balancer)
Microservices – Introduction

Monolithic Architecture – Challenges

Scenario: The shop is very successful and the project grows steadily

• Number of components and LOC increases as more features are added
• More project members are required for development, QA, design etc.

Challenges:

• Communication overhead between project members
• Decrease in development speed due to increased complexity
• Deployments (and updates) become less frequent

→ Idea: Limit responsibilities of individual project members to individual components instead of entire monolith (e.g. by creating smaller teams).
Microservices – Introduction

Microservice Architecture – Decomposing the Monolith

Concept: Decompose complex applications into smaller units
(usually single tasks or even subtasks)

Properties of a Microservice:

• Self-contained unit providing its own persistence layer etc.
• May be deployed to an arbitrary number of processes
• Clearly defined scope of responsibility (loose coupling; high cohesion)
• Owned by a single team
(responsible for development [and operation])

→ Motto: “You build it, you run it!”
Microservices – Introduction

Microservice Architecture – What is the difference to a SOA?

Microservices are considered a specialisation of SOA.

- All microservice architectures are also service-oriented architectures.
- Microservices introduce additional constraints to SOA:
  - All services must be deployable independent from one another.
  - Size and domain of a microservice are limited (no limitations in SOA).
  - Every service runs in its own process and contains its own storage.
  - No need for an ESB, services handle communication individually.
- A SOA can be comprised of or integrate with multiple microservices.
Advantages:

- Each microservice can be **deployed and scaled independently**
- Ownership by a single, small team (*developer, designer, [administrator] …*) reduces **communication overhead** among project members
- Small size & limited scope allow for **easy replacement** of individual services
- Rapid development lifecycle promotes **continuous integration**

→ **But**: These advantages can quickly turn into challenges!

**Consequence:**

Microservices require **strict adherence** of developers to guidelines provided by architects to **prevent introduction of dependencies**.
Agenda

Chapter 1  Introduction

Chapter 2  Architectural Tasks and Challenges
- Decomposition
- Deployment
- Technology Heterogeneity
- Scalability
- Communication between Microservices
- Monitoring

Chapter 3  Patterns for Resilience and QoS

Chapter 4  Applications and Examples

Chapter 5  Technology Solutions

Chapter 6  Summary and Conclusions
Microservices – Tasks and Challenges

Decomposition – The art of dividing and decoupling

Problem with Monoliths:
• Refactoring is necessary to conform to initial architectural vision

Benefit of Microservices:
• Small enough to replace entire service in case of major changes
• Keeps code rot in check due to limited number of LOC per service

Challenges:
• Small enough, but not too small
  Choosing the correct size for a microservice is important to prevent the overhead from outweighing the benefit.
• Durable Interfaces
  Replacements should not introduce changes to provided interfaces as this would incur additional changes in other services.
Agenda

Chapter 1  Introduction

Chapter 2  Architectural Tasks and Challenges
- Decomposition
- Deployment
- Technology Heterogeneity
- Scalability
- Communication between Microservices
- Monitoring

Chapter 3  Patterns for Resilience and QoS

Chapter 4  Applications and Examples

Chapter 5  Technology Solutions

Chapter 6  Summary and Conclusions
Microservices – Tasks and Challenges

Deployment – What is deployed when and how frequently?

Problem with Monoliths: Fixed deployment cycles which may lengthen over time

Benefit of Microservices:

- No fixed deployment schedule (e.g. once per month or quarter)
- Teams may deploy frequently and independently from one another
- New features and changes can be shipped more rapidly

Challenges:

- **Loose Coupling**: A change in one microservice should not (or in practice very rarely) require a change in another microservice.
- **Availability and Continuous Integration (CI)**: There must always be a fully tested version available to all other services, while the diversity of deployed versions should be kept low.
Agenda

Chapter 1  Introduction

Chapter 2  Architectural Tasks and Challenges
  - Decomposition
  - Deployment
  - Technology Heterogeneity
  - Scalability
  - Communication between Microservices
  - Monitoring

Chapter 3  Patterns for Resilience and QoS

Chapter 4  Applications and Examples

Chapter 5  Technology Solutions

Chapter 6  Summary and Conclusions
Advantages of Microservices:

- Every service appears as a **black box** to other services.
- Teams can always use the “best tool for the job” within their own service.
  (e.g. data storage paradigm, programming language, libraries, build chain)

Challenges:

- Overall complexity increases (e.g. licensing, architecture overview)
- Employees cannot easily be reassigned between teams (missing expertise)
- **“Bus factor”**: Can development on a microservice continue when a developer leaves the company?
Microservices – Tasks and Challenges
*Technology Heterogeneity – Advantages and Challenges*

**Examples**

Different microservices may use fundamentally different technology stacks.
Agenda

Chapter 1  Introduction

Chapter 2  Architectural Tasks and Challenges
- Decomposition
- Deployment
- Technology Heterogeneity
- Scalability
- Communication between Microservices
- Monitoring

Chapter 3  Patterns for Resilience and QoS

Chapter 4  Applications and Examples

Chapter 5  Technology Solutions

Chapter 6  Summary and Conclusions
Microservices – Tasks and Challenges

Scalability – Independence vs. communication overhead

Advantages of Microservices:

• Each service runs in a process of its own and provides its own storage.
  → Microservices can be \textit{scaled independently} from each other.
• Modularity allows easy deployment of additional service instances.

Challenges:

• Services must be able to \textit{scale vertically} as well as horizontally.
• Every instance must be able to answer a request, potentially introducing \textit{communication overhead} between instances.
Microservices – Tasks and Challenges

Scalability – Independence vs. communication / synchronization overhead

Scenario 1:
• All services are provided with an equal amount of resources.

Scenario 2:
• B and C continue to share resources.
• A is provided with dedicated resources.

Scenario 3:
• B and C continue to share resources.
• Additional instances of A and C are created with dedicated resources.
Agenda

Chapter 1  Introduction

Chapter 2  Architectural Tasks and Challenges
  Decomposition
  Deployment
  Technology Heterogeneity
  Scalability
  Client-Server Integration
  Communication between Microservices
  Monitoring

Chapter 3  Patterns for Resilience and QoS

Chapter 4  Applications and Examples

Chapter 5  Technology Solutions

Chapter 6  Summary and Conclusions
Microservices – Tasks and Challenges

Communication between Microservices – Patterns and Models

Advantages of Microservices:

• Direct communication between services lifts the requirement for a centralised enterprise service bus.
• Inter-service communication patterns can be chosen as needed.

Challenges:

• Communication between services becomes more complex:
  • Will cross process and potentially even data center boundaries,
  • can no longer be handled via method calls (monolith) and
  • requires (potentially expensive) inter-process communication.
• Interfaces should not be too fine-grained to reduce overhead.
• Calls to other services can not be considered instantaneous and must be handled in a non-blocking manner.
Microservices – Tasks and Challenges

*Communication between Microservices – Patterns and Models*

Examples of Communication Patterns:

- **Request Response**
  - Immediate answer *(e.g. via HTTP using a RESTful API)*
  - Simple, direct and intuitive, but potentially blocking.
  - Requires polling if service A wants to keep track of the state of B
Microservices – Tasks and Challenges

Communication between Microservices – Patterns and Models

Examples of Communication Patterns:

- **Publish Subscribe** (Event-based communication)
  - Spatial Decoupling: Arbitrary number of publishers and subscribers
  - Temporal Decoupling: Messages may be delivered at any time
  - Subscribers are automatically notified on new messages
  - Asynchrony may increase complexity
Microservices – Tasks and Challenges

Communication – Request Response vs. Event-Based

Request - Response

Event
- „A (sudden) occurrence“
Agenda

Chapter 1 Introduction

Chapter 2 Architectural Tasks and Challenges
- Decomposition
- Deployment
- Technology Heterogeneity
- Scalability
- Client-Server Integration
- Communication between Microservices

Chapter 3 Patterns for Resilience and QoS

Chapter 4 Applications and Examples

Chapter 5 Technology Solutions

Chapter 4 Summary and Conclusions
Microservices – Tasks and Challenges
Monitoring – Keeping Track of Key Metrics

Advantages of Microservices:

- Replaceability and small scope of individual services allows for quick reactions and precise localisation of issues.

Challenges:

- Distributed logs etc. need to be collected and aggregated
- Events pertaining to the same, initiating request need to be correlated across all APIs to trace back downstream errors (e.g. using a shared request id).
- Must keep track of various metrics and key performance indicators (KPI)
  - System Level: CPU load, memory consumption, I/O operations, …
  - Application Level: Response times, error rates, …
- Reliable and fail-safe: Monitoring blackouts are a worst-case scenario, as there is no way to tell, how the entire system behaves during that time.
Agenda

Chapter 1  Introduction

Chapter 2  Architectural Tasks and Challenges

Chapter 3  Patterns for Resilience and QoS
- Circuit Breakers
- Chaos Testing
- Canary Environments

Chapter 4  Applications and Examples

Chapter 5  Technology Solutions

Chapter 6  Summary and Conclusions
Microservices – Patterns for Resilience and QoS

Circuit Breakers – Preventing Failures from Cascading

**Problem:**
Performance issues of a downstream service can **impact upstream services**.

**Idea:**
- Monitor services to **detect issues** and potential failure as early as possible
- Provide **fail-fast or fall back mechanism** to prevent upstream cascades

---

Based on: [Newman2015]
Microservices – Patterns for Resilience and QoS

Circuit Breakers – Example: Netflix OSS – Hystrix

Hystrix – An OSS resilience solution for microservices

- Wraps calls to dependencies to track successes, failures, timeouts, …
- Provides a fail fast mechanism to **prevent blocking** requests during high load
- Trips **circuit-breakers** to stop all requests to a particular service (triggered e.g. when error percentage reaches threshold)
- Executes **fall-back logic** in case of failed requests etc.

→ **Goal:** Prevent failures or high latencies in individual services **from cascading** to other parts of the system: *Fail fast, degrade gracefully (if possible).*

Source: [https://github.com/Netflix/Hystrix](https://github.com/Netflix/Hystrix)
Microservices – Patterns for Resilience and QoS

Circuit Breakers – Example: Netflix OSS – Hystrix

Hystrix Dashboard – Key Performance Indicators

Source: https://github.com/Netflix/Hystrix
Agenda

Chapter 1  Introduction

Chapter 2  Architectural Tasks and Challenges

Chapter 3  Patterns for Resilience and QoS
  - Circuit Breakers
  - Chaos Testing
  - Canary Environments

Chapter 4  Applications and Examples

Chapter 5  Technology Solutions

Chapter 6  Summary and Conclusions
Problem:

On microservice level, code tests can identify potential failures and load tests can point out scalability limitations, but neither tests the entire ecosystem.

→ Most production failures are related to issues elsewhere in the ecosystem.

Idea:

• **Push** microservices **to fail** in production:
  Make it **fail all of the time** and in every way possible.

• Run **scheduled** tests as well as **random test**:
  Catch developers off guard as well as in prepared states of readiness.

• Provide chaos testing **as a service**:
  **Dedicated team**, no ad hoc cooperation across multiple teams.

• Break **every microservice** and **every piece of infrastructure** (*multiple times!*).

Based on: [Fowler2017]
Microservices – Patterns for Resilience and QoS

**Chaos Testing – Because Chaos is Closer to Reality**

Example:

- Block individual APIs, stop single services, introduce network latency, break entire hosts, disconnect entire regions or datacentres …

→ Even though it is called Chaos Testing, it has to be **well controlled** to prevent it from bringing down the entire ecosystem or go rogue!
Agenda

Chapter 1  Introduction

Chapter 2  Architectural Tasks and Challenges

Chapter 3  Patterns for Resilience and QoS
  Circuit Breakers
  Chaos Testing
  Canary Environments

Chapter 3  Applications and Examples

Chapter 5  Technology Solutions

Chapter 6  Summary and Conclusions
Microservices – Patterns for Resilience and QoS
Canary Environments – The Last Stage before Full Release

Problem:

Even after passing all tests, actual production traffic may still cause unexpected failure which may bring down the entire production environment.

Idea:

• Do not switch the entire production traffic over to the new version at once.
• Deploy new versions to a Canary Environment, which servers only about 5 – 10 % of the production traffic.
• Once the canary survived an entire traffic cycle (interval after which traffic patterns repeat), deploy it to the entire production platform.

→ If a canary fails, only a small number of clients will be affected and the deployment can be rolled back easily.

Based on: [Fowler2017]
Microservices – Patterns for Resilience and QoS

Canary Environments – The Last Stage before Full Release

Example:

- Rollout of a new version for service A to the canary environment
- New canary environment only serves a small portion of production traffic
Agenda

Chapter 1  Introduction
Chapter 2  Architectural Tasks and Challenges
Chapter 3  Patterns for Resilience and QoS

Chapter 4  Applications and Examples
  Service Granularity
  Case Study

Chapter 5  Technology Solutions
Chapter 6  Summary and Conclusions
Cost-based definition of service granularity

Source: [Gouigoux2017]
Agenda

Chapter 1  Introduction
Chapter 2  Architectural Tasks and Challenges
Chapter 3  Patterns for Resilience and QoS

Chapter 4  Applications and Examples
  Service Granularity
  Case Study

Chapter 5  Technology Solutions
Chapter 6  Summary and Conclusions
Microservices – Applications and Examples

Case Study – Danske Bank

Foreign Exchange (forex, FX):

- Exchange of one currency for another or the conversion of one currency into another currency.
- Encompasses the conversion of currencies at an airport kiosk to payments made by corporations, financial institutions and governments.
- Largest financial market in the world

Danske Bank FX System

- Mission critical system of the Danske Bank, implements FX
- Gateway between the international markets and the Danske Bank clients

Source: [Dragoni2017]
Microservices – Applications and Examples

Case Study – Danske Bank

Problems with the FX System system:

• Large Components with little cohesion and tight coupling

• Multiple communication and integration paradigms (RPC, messaging)

• Complex and manual deployment

• No global monitoring and logging

• Technology dependencies (MS .NET)

→ Great expense with respect to maintenance, quality assurance, and deployment

Idea:

Migration of the FX system from a monolithic to a microservice architecture.

Source: [Dragoni2017]
Microservices – Applications and Examples

Case Study – Danske Bank

Approach:

• Shift business logic in dedicated services
• Provide “foundation services” for system management tasks
• Provide infrastructure services
• Use Docker and Docker Swarm for deployment, load balancing, and fail over
• Introduce Continuous Integration

Source: [Dragoni2017]
Agenda

Chapter 1  Introduction
Chapter 2  Architectural Tasks and Challenges
Chapter 3  Patterns for Resilience and QoS
Chapter 4  Applications and Examples

Chapter 5  Technology Solutions
  Spring Cloud
  Netflix OSS

Chapter 6  Summary and Conclusions
Microservices – Technology Solutions

Spring cloud – Overview of an Ecosystem

Source: https://jaxenter.de/cloud-native-anwendungen-42976
Agenda

Chapter 1  Introduction
Chapter 2  Architectural Tasks and Challenges
Chapter 3  Patterns for Resilience and QoS
Chapter 4  Applications and Examples

Chapter 5  Technology Solutions
  - Spring Boot
  - Netflix OSS

Chapter 6  Summary and Conclusions
Netflix has open-sourced a great number of their tools and services.

Some examples taken from their open-source ecosystem:

**Runtime Services & Libraries**
- Archaius
- Eureka
- Hystrix
- Zuul

**Build and Delivery Tools**
- Nebula
- Animator
- Spinnaker

**Insight Reliability & Performance**
- Atlas
- Chaos Monkey
- Edda
- Spectator
- Vector

**Other Areas**
- Security
- User Interface
- Data Persistence
- Content Encoding
- Big Data

Source: [https://netflix.github.io](https://netflix.github.io)
Microservices – Technology Solutions

Netflix OSS – Zuul: The Edge Service – Component Overview

Source: http://techblog.netflix.com/2013/06/announcing-zuul-edge-service-in-cloud.html
Microservices – Technology Solutions
Netflix OSS – Zuul: The Edge Service

Zuul – The Gatekeeper

• Provides various filters to enable dynamic routing, monitoring, resiliency and security.

• Uses a number of other services to perform certain tasks, e.g.:
  • **Hystrix** – Real time metrics and resilience
  • **Ribbon** – Routing and load balancing
  • **Eureka** – Service and instance location
  • **Turbine** – Server-Sent Event (SSE) stream aggregation
  • **Archaius** – Thread-safe configuration management

Source: http://techblog.netflix.com/2013/06/announcing-zuul-edge-service-in-cloud.html
Microservices – Technology Solutions
Netflix OSS – Ribbon: Routing and Load Balancing

Ribbon – The rule based load balancer

- Zone-based load balancing in the cloud (*avoids cross zone traffic*)
- Capable of dynamically discovering services in its zone (*using Eureka*)
- Filters servers based on:
  - **Availability** – determined via ping interface
  - **Broken Circuits** – provided by Hystrix
- Dynamic configuration for load balancers via Archaius
- Commonly used balancing rules:
  - **Round Robin** – default or fallback for more complex rules
  - **Availability Filtering** – uses tripped (broken) circuits
  - **Weighted Response Time** – longer response time, less weight in selection

Source: [https://github.com/Netflix/ribbon/wiki/Features](https://github.com/Netflix/ribbon/wiki/Features)
Microservices – Technology Solutions

Netflix OSS – Eureka: Service and Instance Discovery

Eureka – The Service Registry

- Used to locate services in an AWS cloud environment
- Additional **load balancing and failover mechanism** for middle-tier servers
- Automated service removal via **registration renewal heartbeat**

Source: [https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance](https://github.com/Netflix/eureka/wiki/Eureka-at-a-glance)
Agenda

Chapter 1  Introduction
Chapter 2  Architectural Tasks and Challenges
Chapter 3  Patterns for Resilience and QoS
Chapter 4  Applications and Examples
Chapter 5  Technology Solutions

Chapter 6  Summary and Conclusions
Microservices – Summary and Conclusions

• The *microservices paradigm* is a new *promising approach* in provisioning software:
  • *Small* services, *self-contained*, *high cohesion* and *loose coupling*
  • Runs in a separate *process*
  • Maybe *deployed* and *scaled independently* from each other
  • Owned by a *single team* – “You build it, you run it”
  • *Continuous integration – continuous delivery* (CICD)
• Efficient *OSS frameworks* for development & delivery are available
  • Spring Boot / Cloud, Netflix OSS, Docker Swarm, Kubernetes, …
  • BUT: High frequency of change
• Some *success stories*: Amazon, Netflix, Google, Danske Bank, Otto …
• Is the microservices paradigm just a *hype* – or is it the *silver bullet*
  which will solve all our problems in the software industry?
References & Additional Reading


