μ-services

Microservices
Microservices are currently hot!

Source: https://www.google.com/trends/explore?q=Microservices, 2016-10-03
Introduction
Basic Concepts of Microservices
What are Microservices?

Small, autonomous services working together

Single responsibility principle applied to SOA
See also concept of Bounded Context

Best used with DevOps and continuous deployment

Enhance cohesion, decrease coupling, enable incremental evolvement

How small are Microservices?

It depends (e.g. team structure, DevOps maturity, etc.)
“... one agile team can build and run it”, “... can be rebuilt by a small team in two weeks”
Find an individual balance

Autonomous = deploy changes without affecting others

Technology- and platform-agnostic APIs

See also https://en.wikipedia.org/wiki/Microservices
Loose Coupling

Tight Coupling
A change in one module usually forces a ripple effect of changes in other modules
See also Disadvantages of Tight Coupling

Loose Coupling
Components have little or no knowledge of the definitions of other components
Coupling is reduced by e.g. standards, queues, etc.

Microservices = loose coupling wanted
Single change → single deployment
No timing issues (if system A is deployed, system B needs update at the same time)
Cohesion

Highly cohesive systems
Functionality is strongly related
Modules belong together

Microservices = high cohesion wanted
Functions grouped in a services because all contribute to a single well-defined task
Reduce risk that a requirement concerns many different system components
Bounded Context

Microservices often represent bounded contexts

Business-focused design
Less technical-focused design based on technical layers

Source: http://martinfowler.com/bliki/BoundedContext.html
Microservices

Modelled around business concepts
Highly observable
Culture of Automation
Hide implementation details
Decentralized
Independently deployed
Isolate failures

Fundamental ideas

Work alongside many state-of-the-art approaches for software development
Agile development techniques
Continuous Integration/Delivery
DevOps
Cloud Computing
Containers
Why? Why not?
Why Microservices?

Work well in heterogeneous environments
- Right tool for the job
- Available skills of team members
- Grown environment (e.g. M&A, changing policies, changing overall designs)

Easier to test/adopt new technologies
- Reduce risk and cost of failure
- New platforms (e.g. Node.js instead of .NET), new versions (e.g. .NET Core),

Resilience
- Reduce single point of failures
- Support different SLAs for difference modules (costs, agility)
- Separation of services add complexity (e.g. network) → criticism of Micrservices
Why Microservices?

Let people take responsibility
Teams “own” their services
You build it, you run it

Scaling
Fine-grained scaling is possible

Simplify deployment of services
Overall, deployment of many Microservices might be more complex ➔ criticism
Deployment patterns: https://www.nginx.com/blog/deploying-microservices/
Why Microservices?

Composability
   Hexagonal architecture

Ability to replace system components
   Outdated technology
   Changed business requirements
Why Not? (Examples)

Harder to debug and troubleshoot
  Distributed system
  Possible mitigation: Mature logging and telemetry system

Performance penalty
  Network calls are relatively slow
  Possible mitigation: Remote calls for larger units of work instead of chatty protocols

No strong consistency
  We are going to miss transactions!
  Possible mitigation: Idempotent retries
Why Not? (Examples)

Harder to manage
You have to manage lots of services which are redeployed regularly
Possible mitigation: DevOps, Automation

System is too small
For small systems, monolithic approach is often more productive
Cannot manage a monolith (e.g. deployment)? You will have troubles with Microservices!

Environment with lots of restrictions
Microservices need a high level of autonomy
Team Organization
Conway’s Law

“Any organization that designs a system will inevitably produce a design whose structure is a copy of the organization’s communication structure”

Organizational hurdles for Microservices
- Tightly-coupled organizations
- Geographically distributed teams
- Missing tools (e.g. self-service cloud infrastructure, CI/CD tools)
- Unstable or immature service that frequently changes
- Missing culture of taking ownership (need someone to blame)
- Cope with many different and new technologies

Source: Conway, How Do Committees Invent, Datamation magazine, April 1968
Organisational Helpers

Co-locate teams
One team responsible for a single service should be co-located

Embrace open source development style
Works internally, too

Internal consultants, custodians and trusted committers
Quality gateways
Servant leaders

Step-by-step approach

Be clear in communication
E.g. responsibilities, long-term goals, changing roles
Microservices Architects...

...don’t create perfect end products
...help creating “a framework in which the right systems can emerge, and continue to grow”

...understand the consequences of their decisions
...code with the team (“architects should code”, “coding architect”)

...aims for a balance between standardization and freedom
Build skills for a certain technology vs. right tool for the right job

...create guiding principals and practices
Example for principals (largely technology-independent): https://12factor.net/
Example for practices (often technology-dependent): .NET Core Coding Guidelines

Recommended reading: Newman, Sam. Building Microservices, O’Reilly Media
Guidance, Governance

Samples
Small code samples vs. *perfect* examples from real world

Templates, code generators
Examples: Visual Studio Templates, .NET Core CLI, Angular CLI

Shared libraries
Be careful about tight coupling!
Example: Cross-platform libraries based on .NET Standard Library for cross-cutting concerns

Handle and track exceptions from principals and practices
Remember goal of Microservices: Optimize autonomy
→ Exceptions should be allowed
Shift to DevOps

Old World
- Focus on planning
- Compete, not collaborate
- Static hierarchies
- Individual productivity
- Efficiency of process
- Assumptions, not data

New World
- Focus on delivering
- Collaborate to win
- Fluent and flexible teams
- Collective value creation
- Effectiveness of outcomes
- Experiment, learn and respond
DevOps habits and practices

**PRACTICES**
- Automated Testing
- Continuous Integration
- Continuous Deployment
- Release Management

**PRACTICES**
- Usage Monitoring
- Telemetry Collection
- Testing in Production
- Stakeholder Feedback

**PRACTICES**
- Code Reviews
- Automated Testing
- Continuous Measurement

**PRACTICES**
- Application Performance Management
- Infrastructure as Code
- Continuous Deployment
- Release Management
- Configuration Management
- Automated Recovery

**FLOW OF CUSTOMER VALUE**

**TEAM AUTONOMY & ENTERPRISE ALIGNMENT**

**BACKLOG refined with LEARNING**

**EVIDENCE gathered in PRODUCTION**

**MANAGED TECHNICAL DEBT**

**PRODUCTION FIRST MINDSET**

**INFRASTRUCTURE is a FLEXIBLE RESOURCE**

**PRACTICES**
- Enterprise Agile
- Continuous Integration
- Continuous Deployment
- Release Management

**PRACTICES**
- Testing in Production
- Usage Monitoring
- User Telemetry
- Stakeholder feedback
- Feature flags
Technical Aspects
Microservice Interfaces
From Monolith to Microservices

Image Source: Chris Richardson, Microservices – From Design to Deployment, NGINX, 2016
Interfaces

Small number of communication standards
Examples: HTTP/REST, OData, GraphQL, OpenID Connect
Goals: Interoperability, productivity (economy of scope), detect malfunctions

Practices and principles for typical use-cases needed
Status Codes
Data encoding
Paging
Dynamic filtering
Sorting
Long-running operations
...

See also https://speakerdeck.com/rstropek/restful-web-api-design
Interface Technology

Tolerant against changes
See also Breaking Change in Microsoft’s REST API Guidelines

Technology-agnostic

Simple to use and provide
Availability of tools, libraries, frameworks, knowledge

Hide implementation details
Shared Database anti-pattern
Interface Design

Synchronous communication
  Request/response pattern
  Bidirectional communication
  Example: RESTful Web API, WebSockets

Asynchronous communication
  Event-driven
  Examples: Service Bus, RabbitMQ, Apache Kafka, Webhooks

Central orchestration or autonomy?
  Example: Business Process Modelling and Execution

Further reading: https://www.nginx.com/blog/event-driven-data-management-microservices/
Interface Mechanisms

Image Source: Chris Richardson, Microservices – From Design to Deployment, NGINX, 2016
Handling Failures

Partial failures
   Single service must not kill entire system

Outage vs. degradation
   Performance degradation
   Single dependent service not available

Circuit breaker pattern
   Track success of requests
   Stop trying if error rate/performance exceeds threshold
   Regular health check or retry
Versioning

**Semantic Versioning (SemVer)**

*Raise awareness for breaking changes*
Definition of a breaking change is necessary

*Avoid breaking changes*
Discussion point: JSON vs. XAML deserializer in C#

*Offer multiple versions in parallel*
Give consumers time to move
Use telemetry to identify slow movers
Libraries vs. Microservices

Goal: Don’t Repeat Yourself (DRY)
Contraction to Microservices architecture?

Good for...
...cross-cutting concerns (use existing, wide-spread libraries)
...sharing code inside a service boundary

Client libraries
Hide complexity of communication protocol
Implement best practices (e.g. retry policy)
Example: Azure Active Directory Authentication Libraries

UI components
Service provides UI fragments (e.g. WebComponents)
Mobile app CI and CD

Code Repository → Build + Deploy → Automated Testing → User Testing

Backlog

Monitor and improve
CI/CD

One code repository and CI/CD build per service
   Possible: Common infrastructure for economy of scope and scale

Build and deployment pipeline
   Compile and fast tests (unit tests)
   Slow tests
   UAT (manual tests, explorative tests)
   Performance testing (e.g. cloud load testing)

Separate deployment from release
   E.g. Azure App Service stages with swapping

Canary releasing
   Direct portion of your traffic against new release and monitor stability
Monitoring

System-wide view of our system’s health
- Contains data from all services
- Logging
- Telemetry (e.g. CPU and memory usage, response times, dependent requests, etc.)

Microsoft’s solutions
- Visual Studio Application Insights
- Hockeyapp

3rd party solutions
- Log analysis with Elastic Stack
- Dynatrace (leader in Gartner Magic Quadrant)
Manual Testing

Manual testing: try the program and see if it works!

Tester plays the role of a user
   Checks to see if there is any unexpected or undesirable behavior

Test plans with specified test cases

Drawbacks
   Slow
   Requires lots of resources → expensive
   Cannot be performed frequently

Heavy manual testing is a showstopper for Microservices
Testing Level

Unit Test
Test single function or class

Service Tests
Bypass UI and test service directly
Stubs or mockups for dependent services/resources (e.g. Mountbank)

End-to-End Tests
Hard in a Microservice environment (e.g. which versions to test?)
Tend to be flaky and brittle
Good approach: Test a few customer-driven “journeys”
Deployment
Deployment Strategies

Single service instance per host
  Inefficient

Multiple service instances per host
  Efficient in terms of resource usage
  No isolation $\Rightarrow$ no resource limitation, no isolated environments, no sandboxes

Service instance per VM
  Based on a common image
  Complete isolation
  Uses resources less efficient $\Rightarrow$ expensive
  Requires mature virtualization environment
Deployment Strategies

Service instance per container
Based on a common image (automatically created)
High level of isolation (like VMs if you use e.g. Windows Hyper-V Container)
Requires running container environment (e.g. Docker Cloud, Azure Container Services)

Serverless deployments
E.g. Azure App Service, Azure Functions
Reduce operations to a minimum
Service Discovery

Dynamically assigned addresses

Changing environment
 Failures
 Scaling
 New versions

Tools
 DNS (e.g. Azure DNS)
 Load Balancer (e.g. Azure LB)
 Discovery and config tools (e.g. Consul)

Image Source: Chris Richardson, Microservices – From Design to Deployment, NGINX, 2016
Client vs. server-side discovery

Image Source: Chris Richardson, Microservices – From Design to Deployment, NGINX, 2016
Deployment

Architecture
Old: Each node contains entire system
New: Unrelated modules behind load balancer/reverse proxy

API Gateways
Marshal backend calls
Aggregate content
Example: Azure API Management

Source: How we ended up with microservices
Data Management
Data Management

Each Microservice has its own data
   No transactions
   No distributed queries
   Duplicated data to a certain extent

Event-driven architecture
   Requires service bus or message broker (e.g. Service Bus, RabbitMQ, Apache Kafka)
   Option: Use DB transaction log

Event sourcing and CQRS
   Read more in MSDN, Martin Fowler
Transactions

Question and avoid **ACID** transactions across services
   Perfectly fine inside service boundaries
   Has consequences on API design (e.g. Azure Storage Entity Group Transactions)

Idempotent retry
   Gather data, try again later

Use **compensating transactions**
Further Readings
Further Readings

Martin Fowler on Microservices
Newman, Sam. Building Microservices, O'Reilly Media

NGINX
Tech Blog
Microservices: From Design to Deployment
Q&A

Thank you for coming!