Microservice Architectures

Dr. Andreas Schroeder - Munich

Source: https://www.pst.ifi.lmu.de/Lehre/wise-14-15/mse/microservice-architectures.pdf

Agenda

- The Pain
- Therefore, Microservices
- Stable Interfaces: HTTP, JSON, REST
- Characteristics
- Comparison with Precursors
- Challenges
  - With special focus on Service Versioning
- Conclusion

The Pain

Observed problems

- Area of consideration
  - **Web systems**
  - Built collaboratively by several development teams
  - With traffic load that requires horizontal scaling
    (i.e. load balancing across multiple copies of the system)

- Observation
  - Such systems are often built as monoliths or layered systems (JEE)

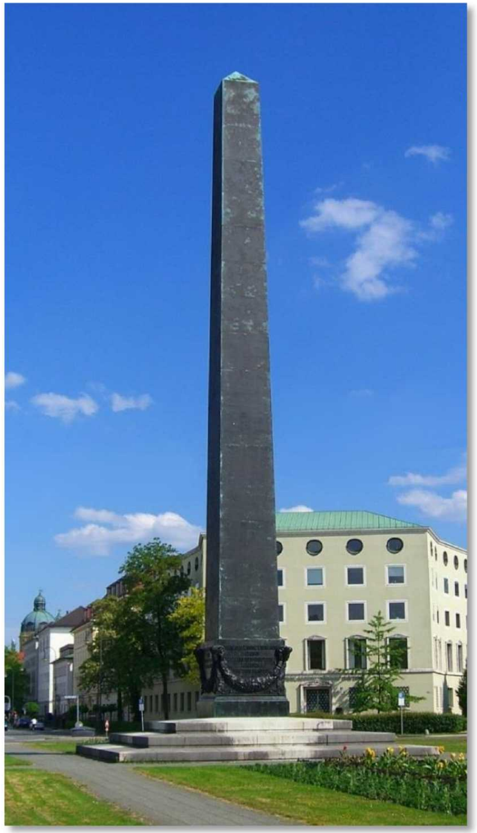
Software Monolith

A Software Monolith

- **One build** and **deployment unit**
- **One code base**
- **One technology stack** (Linux, JVM, Tomcat, Libraries)

Benefits

- **Simple mental model** for developers
  - one unit of access for coding, building, and deploying
- **Simple scaling model** for operations
  - just run multiple copies behind a load balancer



Problems of Software Monoliths

- **Huge** and **intimidating code base** for developers
- **Development tools get overburdened**
  - refactorings take minutes
  - builds take hours
  - testing in continuous integration takes days
- **Scaling is limited**
  - Running a copy of the whole system is resource-intense
  - It doesn’t scale with the data volume out-of-the-box
- **Deployment frequency is limited**
  - Re-deploying means halting the whole system
  - Re-deployments will fail and increase the perceived risk of deployment

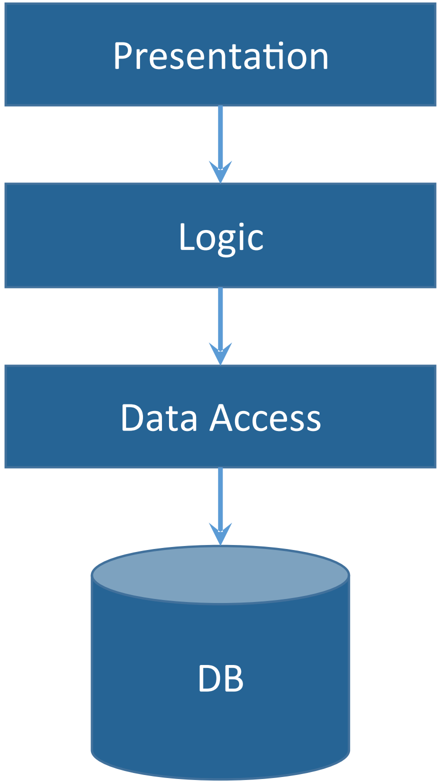
Layered Systems

A layered system decomposes a monolith into layers

- Usually: **presentation**, **logic**, **data access**
- At most one technology stack per layer
  - *Presentation:* Linux, JVM, Tomcat, Libs, EJB client, JavaScript
  - *Logic:* Linux, JVM, EJB container, Libs
  - *Data Access:* Linux, JVM, EJB JPA, EJB container, Libs

Benefits

- **Simple mental model**, simple dependencies
- **Simple deployment** and **scaling model**



Problems of Layered Systems

- Still **huge codebases** (one per layer)
- ... with the same impact on development, building, and deployment
- **Scaling** works better, but **still limited**
- **Staff growth is limited**: roughly speaking, one team per layer works well
  - Developers become specialists on their layer
  - Communication between teams is biased by layer experience (or lack thereof)

Growing systems beyond the limits

- Applications and teams **need to grow beyond the limits** imposed by monoliths and layered systems, and they do – **often in an uncontrolled way**.
- Large companies end up with landscapes of layered systems that often **interoperate in undocumented ways**.
- These landscapes then often **break in unexpected ways**.

How can a company grow and still have a working IT architecture and vision?

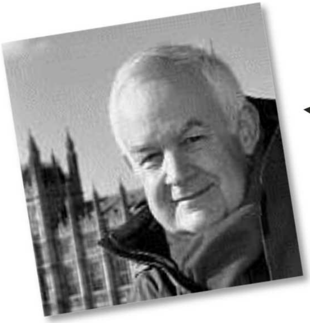
- Observing and documenting successful companies (e.g. Amazon, Netflix) lead to the **definition of the MICRO-SERVICE architecture principles**.

Therefore, Microservices

History

- 2011: First discussions using this term at a software architecture workshop near Venice
- May 2012: microservices settled as the most appropriate term
- March 2012: "Java, the Unix Way" at 33rd degree by James Lewis
- September 2012: "µService Architecture" at Baruco by Fred George
- All along, Adrian Cockroft pioneered this style at Netflix as "fine grained SOA"

James Lewis


Fred George

Adrian Cockroft

http://martinfowler.com/articles/microservices.html#footnote-etymology

Underlying principle

On the logical level, microservice architectures are defined by a

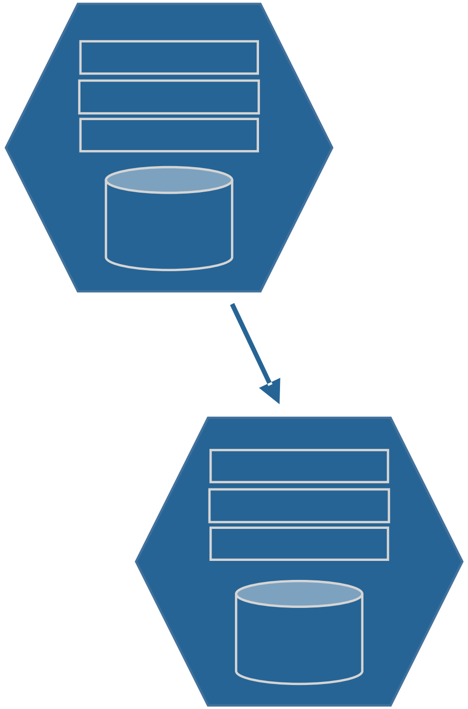
> functional system decomposition into manageable
> and independently deployable components

- The term “micro” refers to the sizing: a microservice must be **manageable by a single development team** (5-9 developers)
- **Functional system decomposition** means vertical slicing
  (in contrast to horizontal slicing through layers)
- **Independent deployability** implies no shared state and inter-process communication (often via HTTP REST-ish interfaces)

More specifically

- Each microservice is functionally complete with
  - **Resource representation**
  - **Data management**
- Each microservice handles one resource (or verb), e.g.
  - Clients
  - Shop Items
  - Carts
  - Checkout

Microservices are **fun-sized services**, as in
"still fun to develop and deploy"



Independent Deployability is key

It enables separation and independent evolution of

- **code base**
- **technology stacks**
- **scaling**
- and **features**, too

Independent code base

Each service has its **own software repository**

- Codebase is maintainable for developers – it fits into their brain
- Tools work fast – building, testing, refactoring code takes seconds
- Service startup only takes seconds
- No accidental cross-dependencies between code bases

Independent technology stacks

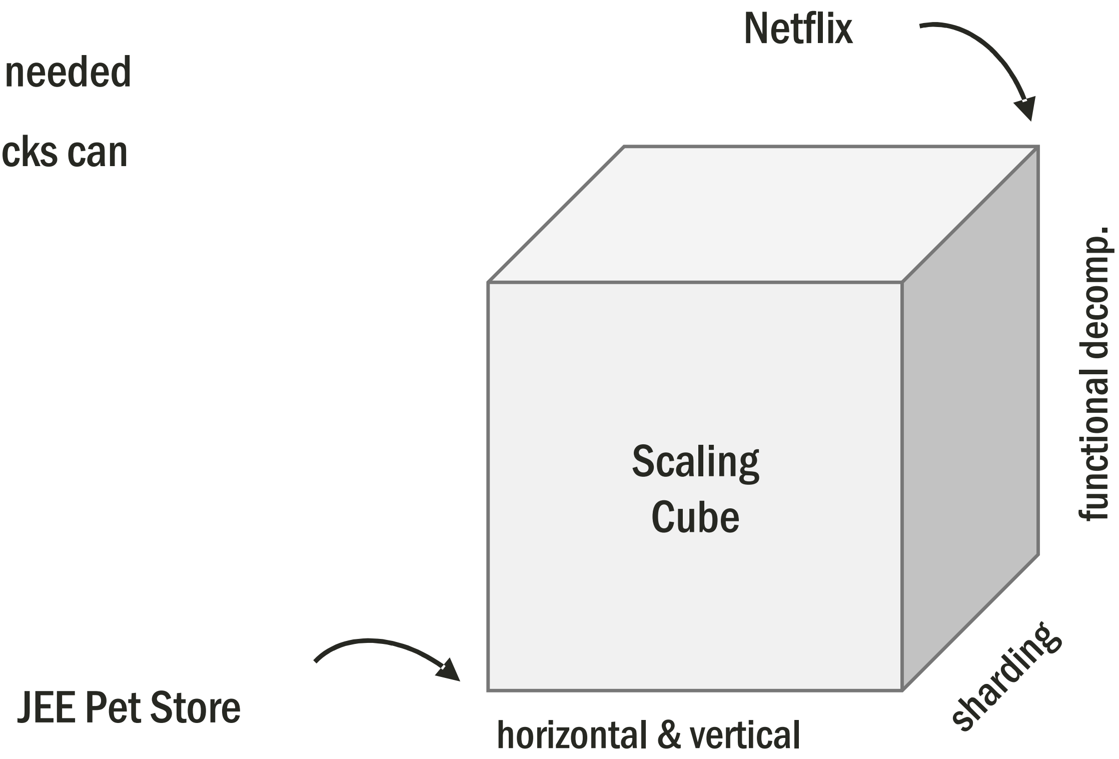
Each service is implemented on its **own technology stacks**

- The technology stack can be selected to fit the task best
- Teams can also experiment with new technologies within a single microservice
- No system-wide standardized technology stack also means
  - No struggle to get your technology introduced to the canon
  - No piggy-pack dependencies to unnecessary technologies or libraries
  - It‘s only your own dependency hell you need to struggle with ☺
- Selected technology stacks are often very lightweight
  - A microservice is often just a single process that is started via command line, and not code and configuration that is deployed to a container.

Independent Scaling

Each microservice **can be scaled independently**

- Identified bottlenecks can be addressed directly
- Data sharding can be applied to microservices as needed
- Parts of the system that do not represent bottlenecks can remain simple and un-scaled



Independent evolution of Features

Microservices can be **extended without affecting other services**

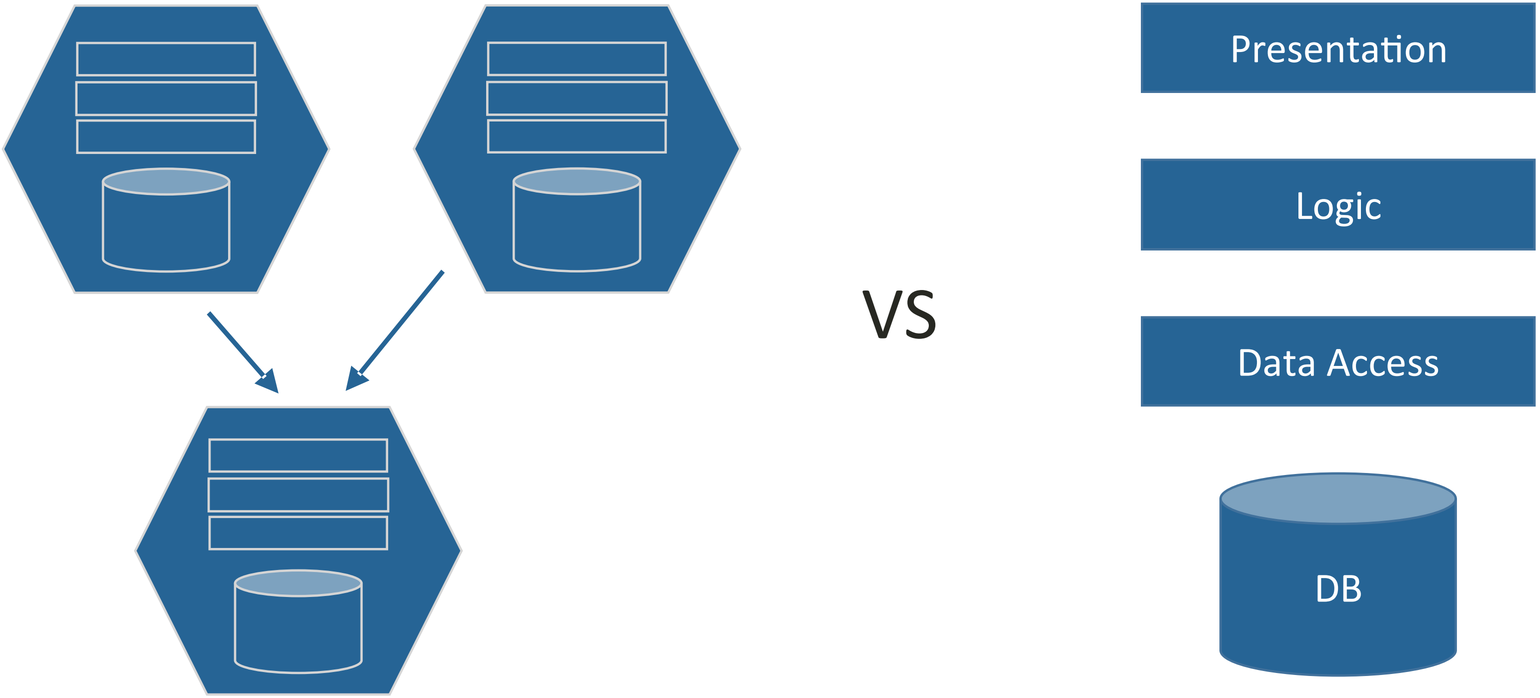
- For example, you can deploy a new version of (a part of) the UI without re-deploying the whole system
- You can also go so far as to replace the service by a complete rewrite

But you have to ensure that the service interface remains stable

Characteristics

Favors Cross-Functional Teams

- Line of separation is along functional boundaries, not along tiers



VS

Presentation

Logic

Data Access

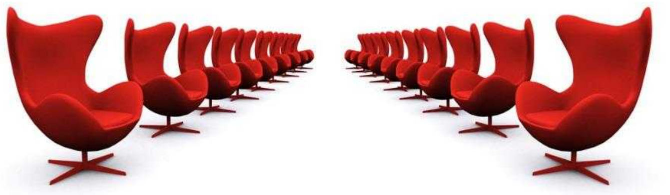
DB

Decentralized Governance

Principle: focus on standardizing the relevant parts, and leverage battle-tested standards and infrastructure

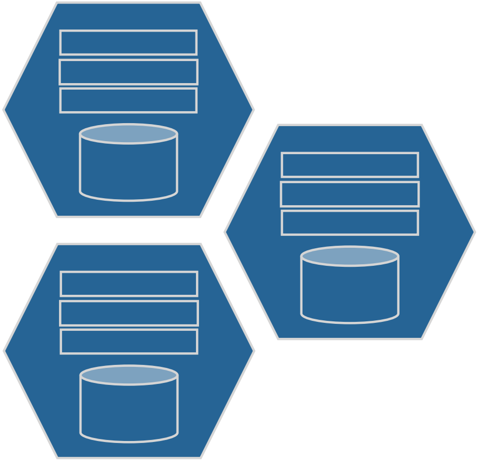
Treats differently

- What **needs** to be standardized
  - Communication protocol (HTTP)
  - Message format (JSON)
- What **should** be standardized
  - Communication patterns (REST)
- What **doesn‘t need** to be standardized
  - Application technology stack



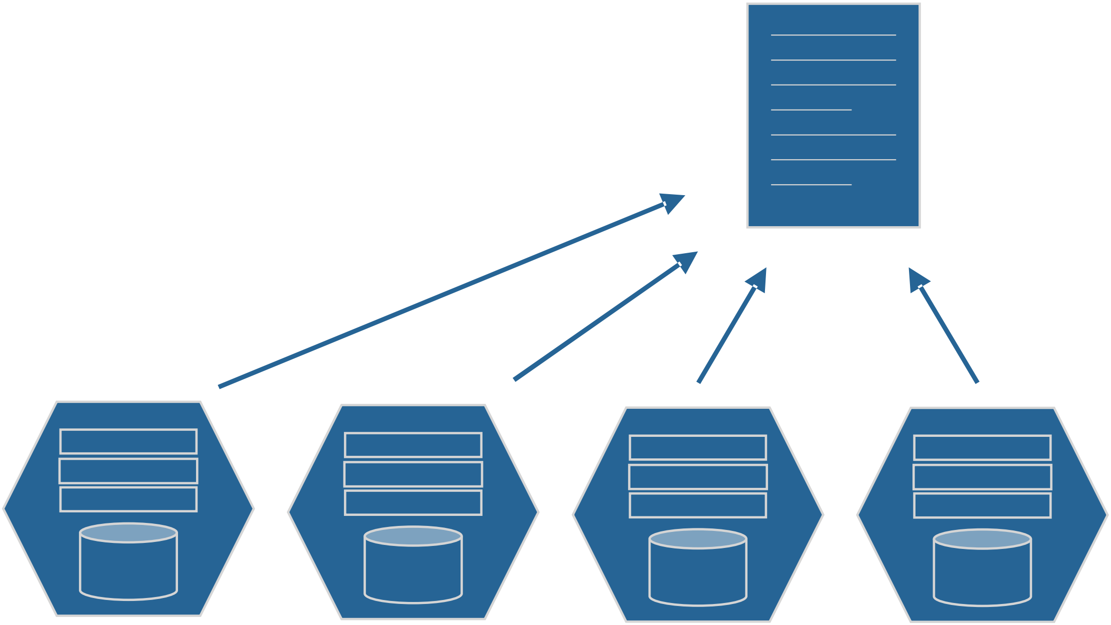
Decentralized Data Management

- OO Encapsulation applies to services as well
- Each service can choose the **persistence solution that fits best** its
  - Data access patterns
  - Scaling and data sharding requirements
- Only few services really need enterprise persistence



Infrastructure Automation

- Having to deploy significant number of services forces operations to **automate the infrastructure** for
  - **Deployment** (Continuous Delivery)
  - **Monitoring** (Automated failure detection)
  - **Managing** (Automated failure recovery)
- Consider that:
  - Amazon AWS is primarily an internal service
  - Netflix uses Chaos Monkey to further enforce infrastructure resilience

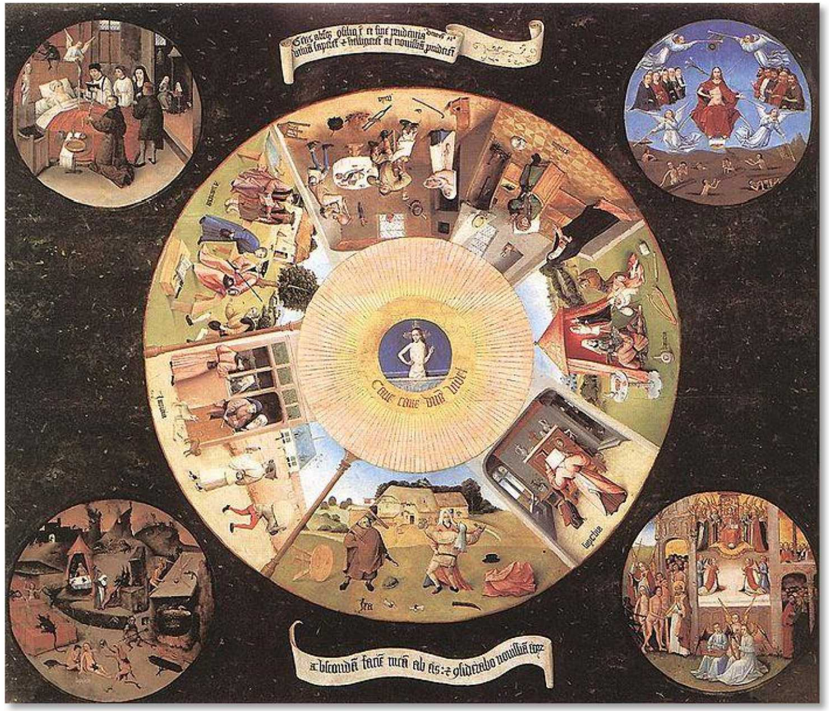


Challenges

Fallacies of Distributed Computing

Essentially everyone, when they first build a distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause big trouble and painful learning experiences.

- The network is reliable
- Latency is zero
- Bandwidth is infinite
- The network is secure
- Topology doesn‘t change
- There is one administrator
- Transport cost is zero
- The network is homogeneous



Peter Deutsch

Microservices Prerequisites

Before applying microservices, you should have in place

- **Rapid provisioning**
  - Dev teams should be able to automatically provision new infrastructure
- **Basic monitoring**
  - Essential to detect problems in the complex system landscape
- **Rapid application deployment**
  - Service deployments must be controlled and traceable
  - Rollbacks of deployments must be easy

Source
http://martinfowler.com/bliki/MicroservicePrerequisites.html

Evolving interfaces correctly

- Microservice architectures enable independent evolution of services – but how is this done **without breaking existing clients**?
- There are two answers
  - **Version service APIs** on incompatible API changes
  - **Using JSON and REST** limits versioning needs of service APIs
- Versioning is key
  - Service interfaces are like programmer APIs – you need to know which version you program against
  - As service provider, you need to keep old versions of your interface operational while delivering new versions
- But first, let’s recap compatibility

API Compatibility

There are two types of compatibility

- Forward Compatibility
  - Upgrading the service in the future will not break existing clients
  - Requires **some agreements on future design features**, and the design of new versions to respect old interfaces
- Backward Compatibility
  - Newly created service is compatible with old clients
  - Requires the **design of new versions to respect old interfaces**

The hard type of compatibility is forward compatibility!

Compatibility and Versioning

Compatibility can’t be always guaranteed, therefore versioning schemes (major.minor.point) are introduced

- *Major version change*: **breaking** API change
- *Minor version change*: **compatible** API change

Note that versioning a service imposes work on the service provider

- Services need to exist in their old versions as long as they are used by clients
- The service provider has to deal with the mapping from old API to new API as long as old clients exist

REST API Versioning

Three options exist for versioning a REST service API

1. Version URIs

   ```
   http://bank.com/v2/accounts
   ```

2. Custom HTTP header

   ```
   api-version: 2
   ```

3. Accept HTTP header

   ```
   Accept: application/vnd.accounts.v2+json
   ```

Which option to choose?

- While developing use option 1, it is easy to pass around
- For production use option 3, it is the cleanest one

REST API Versioning

- It is important to
  - version your API directly from the start
  - install a clear policy on handling unversioned calls
    - Service version 1?
    - Service most version?
    - Reject?

Sources
http://www.troyhunt.com/2014/02/your-api-versioning-is-wrong-which-is.html
http://codebetter.com/howarddierking/2012/11/09/versioning-restful-services/

Conclusion

Microservices: just …?

- Just adopt?
  - No. Microservices are a **possible design alternative for new web systems** and **an evolution path for existing web systems.**
  - There are **considerable amounts of warnings** about **challenges**, **complexities** and **prerequisites** of microservices architectures from the community.
    So don’t underestimate the **implementation effort** (D. Taibi 2018: + 20% in respect to multi-tier)!

- Just the new fad?
  - **Yes and no**. Microservices is a new term, and an evolution of long-known architectural principles applied in a specific way to a specific type of systems.
  - The term is dev and ops-heavy, not so much managerial.
  - The tech landscape is open source and vendor-free at the moment.

Summary

- There is an alternative to software monoliths and multi-tier
- Microservices: **functional decomposition** of systems into **manageable and independently deployable services**
- Microservice architectures means
  - **Independence in code, technology, scaling, evolution**
  - Using battle-tested infrastructure (HTTP, JSON, REST)
- Microservice architectures are challenging
  - Compatibility and versioning while changing service interfaces
  - … transactions, testing, deploying, monitoring, tracing is/are harder

Microservices are no silver bullet, but may be the best way forward for

- **large web systems**
- **built by professional software engineers**

Sources and Further Reading

- http://martinfowler.com/articles/microservices.html
- http://www.infoq.com/articles/microservices-intro
- http://brandur.org/microservices
- http://davidmorgantini.blogspot.de/2013/08/micro-services-what-are-micro-services.html
- http://12factor.net/
- http://microservices.io/
- https://rclayton.silvrback.com/failing-at-microservices
- http://www.activestate.com/blog/2014/09/microservices-and-paas-part-iii
- http://highscalability.com/blog/2014/7/28/the-great-microservices-vs-monolithic-apps-twitter-melee.html
- http://capgemini.github.io/architecture/microservices-reality-check/
- http://www.devopsconference.it/