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)

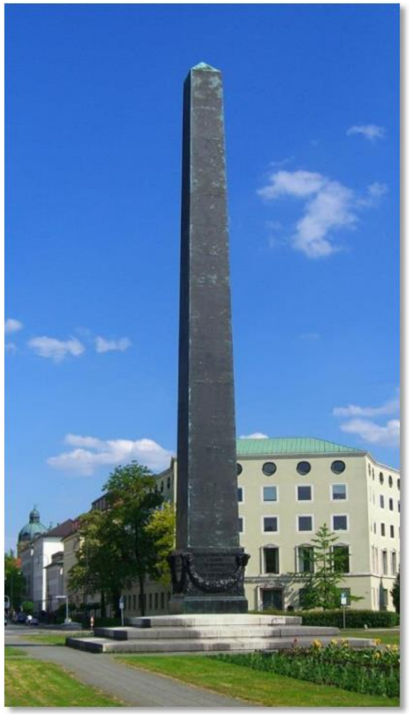
Software Monolith

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

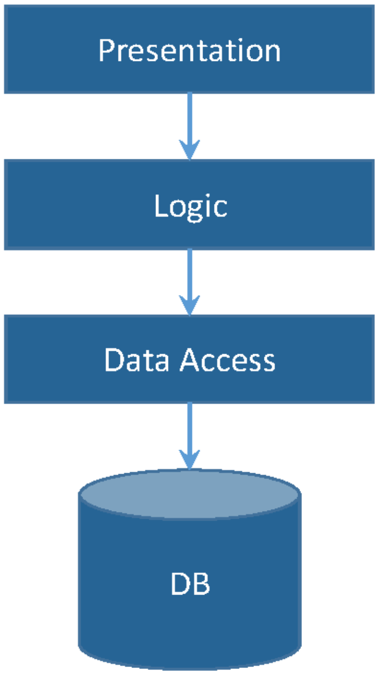
Layered Systems

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 – 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 microservice architecture principles.

Therefore, Microservices

Underlying principle

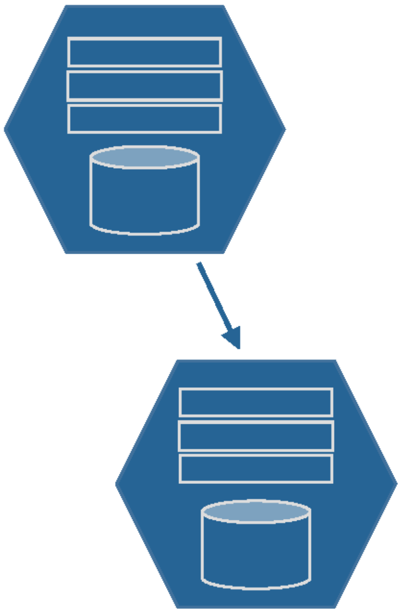
On the logical level, microservice architectures are defined by a

> *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

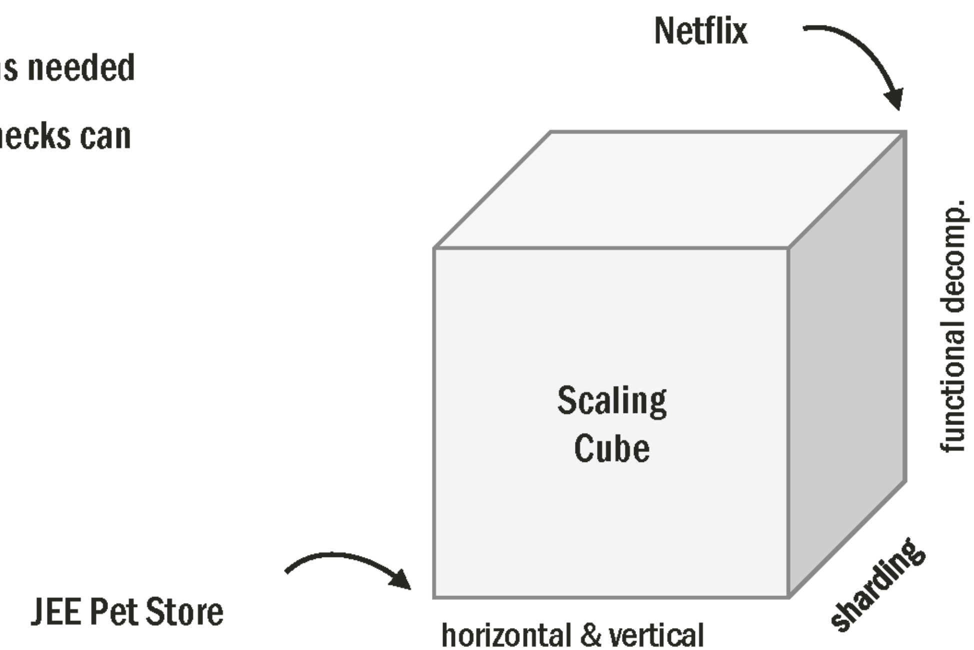
Each service is implemented on its own technology stacks

- 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

Stable Interfaces – standardized communication

Communication between microservices is often standardized using

- HTTP(S) – battle-tested and broadly available transport protocol
- REST – uniform interfaces on data as resources with known manipulation means
- JSON – simple data representation format

REST and JSON are convenient because they simplify interface evolution
(more on this later)

Stable Interfaces: HTTP, JSON, REST

HTTP Example

```
GET / HTTP/1.1
Host: www.codecentric.de
Connection: keep-alive
Cache-Control: max-age=0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
            Chrome/38.0.2125.104 Safari/537.36
Accept-Encoding: gzip,deflate
Accept-Language: de-DE,de;q=0.8,en-US;q=0.6,en;q=0.4
Cookie: …
```

```
HTTP/1.1 200 OK
Date: Tue, 21 Oct 2014 06:34:29 GMT
Server: Apache/2.2.29 (Amazon)
Cache-Control: no-cache, must-revalidate, max-age=0
Content-Encoding: gzip
Content-Length: 8083
Connection: close
Content-Type: text/html; charset=UTF-8
```

HTTP

- Available verbs GET, POST, PUT, DELETE (and more)
  - Safe verbs: GET (and others, but none of the above)
  - Non-idempotent: POST (no other verb has this issue)
- Mechanisms for
  - caching and cache control
  - content negotiation
  - session management
  - user agent and server identification
- Status codes in response (200, 404, etc) for
  information, success, redirection, client error, server error
- Rich standardized interface for interacting over the net

JSON

- Minimal and popular data representation format
- Schemaless in principle, but can be validated if need be

Example of two bank accounts:

```
[{
  "number" : 12345,
  "balance" : -20.00,
  "currency" : "EUR"
},
{
  "number" : 12346,
  "balance" : 120.00,
  "currency" : "USD"
}]
```

object
  {}
  { *members* }
members
  pair
  pair , *members*
pair
  string : *value*
array
  []
  [*elements*]
elements
  value
  value , *elements*
value
  string
  number
  object
  array
  `true`
  `false`
  `null`

json.org

REST

- REST is an architectural style for systems built on the web. It consists of a set of coordinated architectural constraints for distributed hypermedia systems.

- REST describes how to build systems on battle-tested protocols and standards that are already out there (like HTTP)

- REST describes the architectural ideas behind HTTP, and how HTTP can be used to do more than serving static web content

REST Architectural Constraints

- Client-Server: Separation of logic from user interface
- Stateless: no client context on the server
- Cacheable: reduce redundant interaction between client and server
- Layered System: intermediaries may relay communication between client and server (e.g. for load balancing)
- Code on demand: serve code to be executed on the client (e.g. JavaScript)
- Uniform interface
  - Use of known HTTP verbs for manipulating resources
  - Resource manipulation through representations which separated from internal representations
  - Hypermedia as the engine of application state (HATEOAS):
    the response contains all allowed operations and the resource identifiers needed to trigger them

HATEOAS example in JSON

Resource representation

```
{ "number" : 12345,
  "balance" : -20.00,
  "currency" : "EUR",
  "links" : [ {
    "rel" : "self",
    "href" : "https://bank.com/account/12345"
  }, {
    "rel" : "deposit",
    "href" : "https://bank.com/account/12345/deposit"
  } ]
}
```

relation name (known by clients)

URI for operation

Stable Interfaces

- HTTP offers a rich set of standardized interaction mechanisms that still allow for scaling
- JSON offers a simple data format that can be (partially) validated
- REST provides principles and ideas for leveraging HTTP and JSON to build evolvable microservice interfaces

Be of the web, not behind the web

Ian Robinson

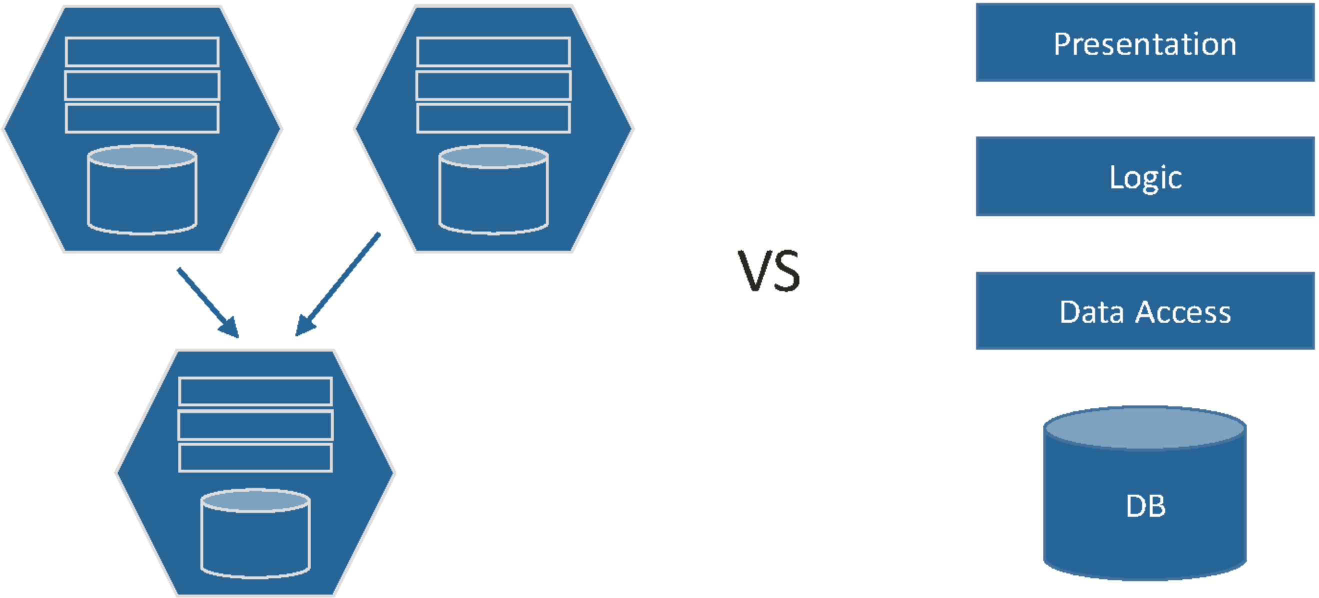
Characteristics

Componentization via Services

- Interaction mode: share-nothing, cross-process communication
- Independently deployable (with all the benefits)
- Explicit, REST-based public interface
- Sized and designed for replaceability
  - Upgrading technologies should not happen big-bang, all-or-nothing-style
- Downsides
  - Communication is more expensive than in-process
  - Interfaces need to be coarser-grained
  - Re-allocation of responsibilities between services is harder

Favors Cross-Functional Teams

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



Decentralized Governance

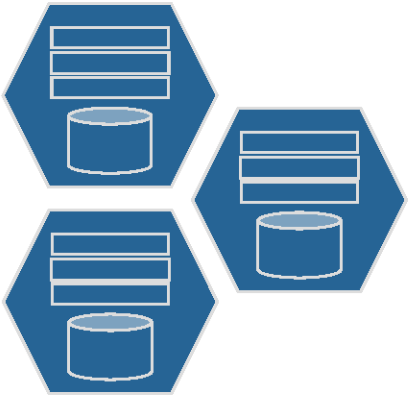
Principle: focus on standardizing the relevant parts, and leverage battle-tested standards and infrastructure

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

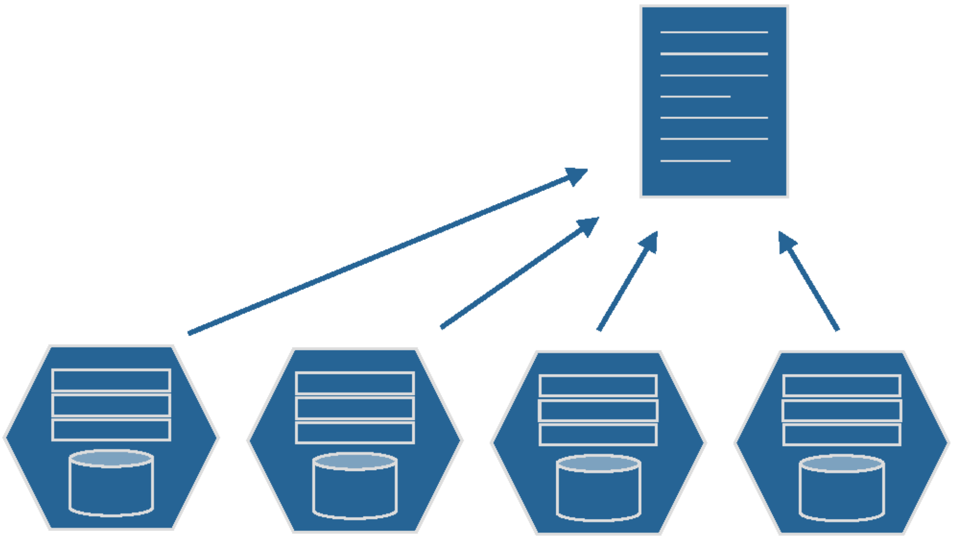
Decentralized Data Management

- 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 enterprisey persistence



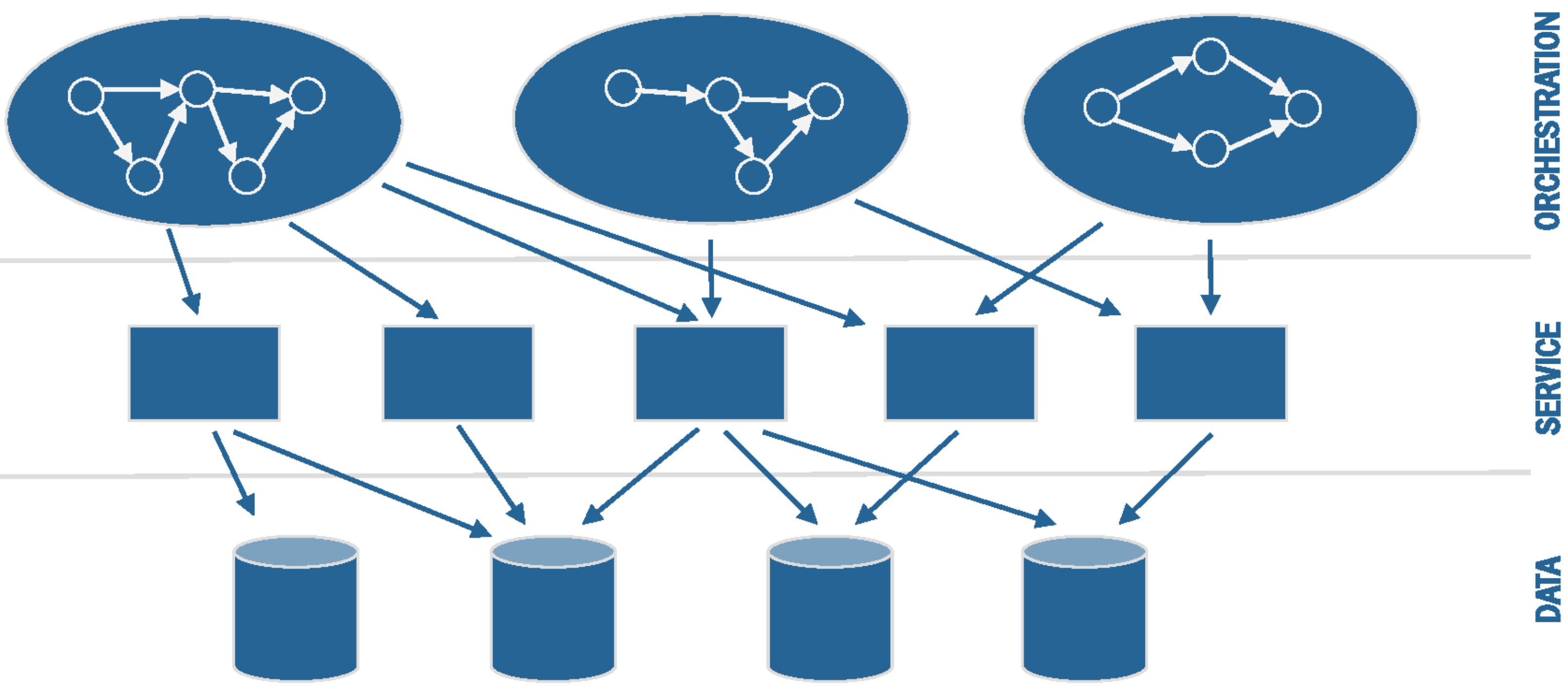
Infrastructure Automation

- 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



Comparisons with Precursors

Service-Oriented Architecture



Service-Oriented Architecture

SOA systems also focus on functional decomposition, but

- services are not required to be self-contained with data and UI, most of the time the contrary is pictured.
- It is often thought as decomposition within tiers, and introducing another tier – the service orchestration tier

In comparison to microservices

- SOA is focused on enabling business-level programming through business processing engines and languages such as BPEL and BPMN
- SOA does not focus on independent deployment units and its consequences
- Microservices can be seen as “SOA – the good parts”

Component-Based Software Engineering

Underlying functional decomposition principle of microservices is basically the same.

Additionally, the following similarities and differences exist:

- State model
  - Many theoretical component models follow the share-nothing model
- Communication model
  - Component technologies often focus on simulating in-process communication across processes (e.g. Java RPC, OSGi, EJB)
  - Microservice communication is intra-process, serialization-based
- Code separation model
  - Component technologies do require code separation
  - Components are often developed in a common code repository
- Deployment model
  - Components are often thought as being deployed into a uniform container

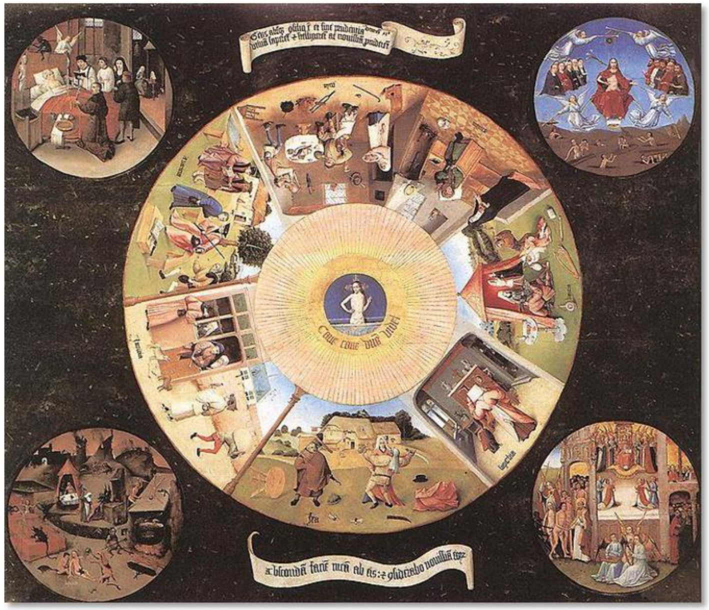
Challenges

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!

Forward compatibility through REST and JSON

REST and JSON have a set of inherent agreements that benefit forward compatibility

- JSON: only validate for what you really need, and ignore unknown object fields (i.e. newly introduced ones)
- REST: HATEOAS links introduce server-controlled indirection between operations and their URIs

```
{ "number" : 12345,
  ...
  "links" : [ {
    "rel" : "deposit",
    "href" : "https://bank.com/account/12345/deposit"
  } ]
}
```

```
"https://accounts.bank.com/12345/deposit"
```

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/

Further Challenges

- Testing the whole system
  - A single microservice isn‘t the whole system.
  - A clear picture of upstream and downstream services is needed for integration testing
- Transactions
  - Instead of distributed transactions, compensations are used (as in SOA)
- Authentication
  - Is often offloaded to reverse proxies making use auf authentication (micro)services
- Request logging
  - Pass along request tokens
  - Add them to the log
  - Perform log aggregation

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.

- 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
- 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