Campus Sontheim

HEILBRONN UNIVERSITY
OF APPLIED SCIENCES

› ARCHITECTURE STYLES

Software Architecture | MSEM | Winter Semester 2019/20

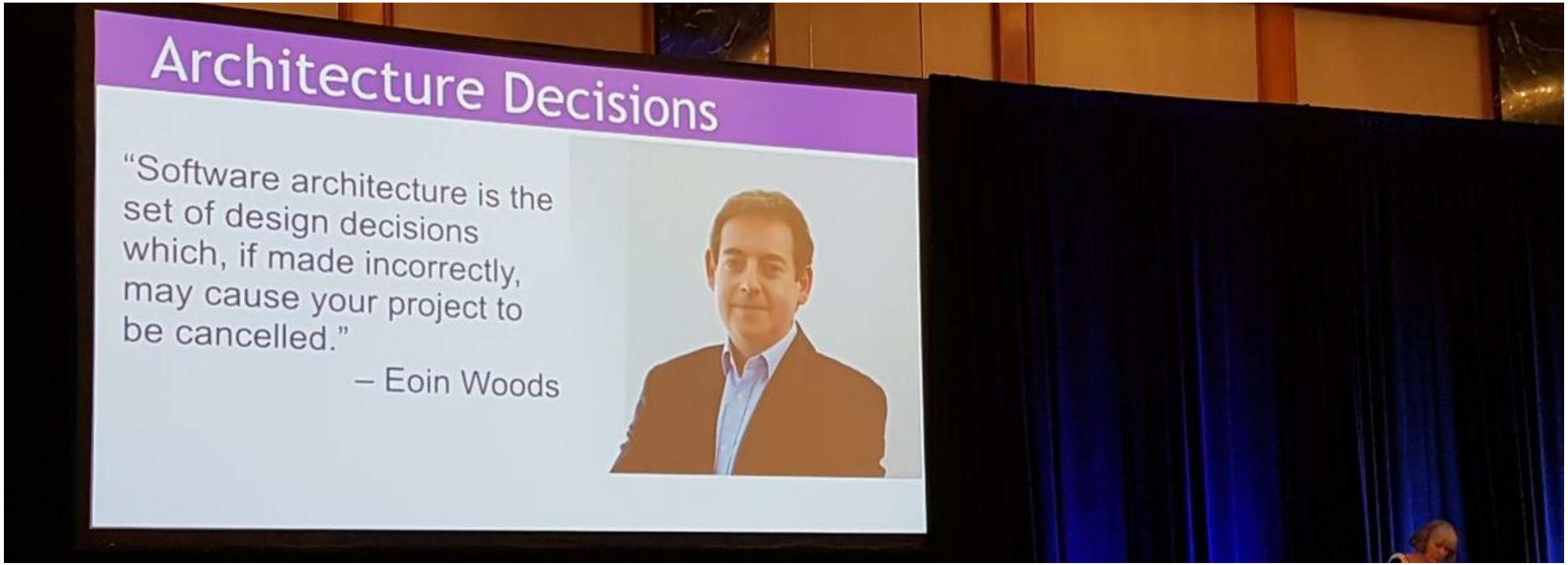
AGENDA

Major architectural styles being covered in this lecture

- Motivation
- Monolith
- Layered Architecture
- Client-Server Architecture
- Component-based Architecture
- Service-oriented Architecture
- REST-style Architecture
- Microservice Architectures
- Event-based Microservice Architectures

Which ones have you already heard about?

MOTIVATION


Architecture Decisions
"Software architecture is the set of design decisions which, if made incorrectly, may cause your project to be cancelled."
– Eoin Woods

› ACHITECTURE STYLES

MONOLITH

- Independent software without dependencies
- Code for user interface, data access and business logic maintained in a common code base
  - No separation of code
  - No relationships
  - Deployed usually as a whole
- A very fundamental way of programming (e.g. one large BASIC program)
- Easy and fast in the very beginning
- Hard to extend and maintain once the code base grows

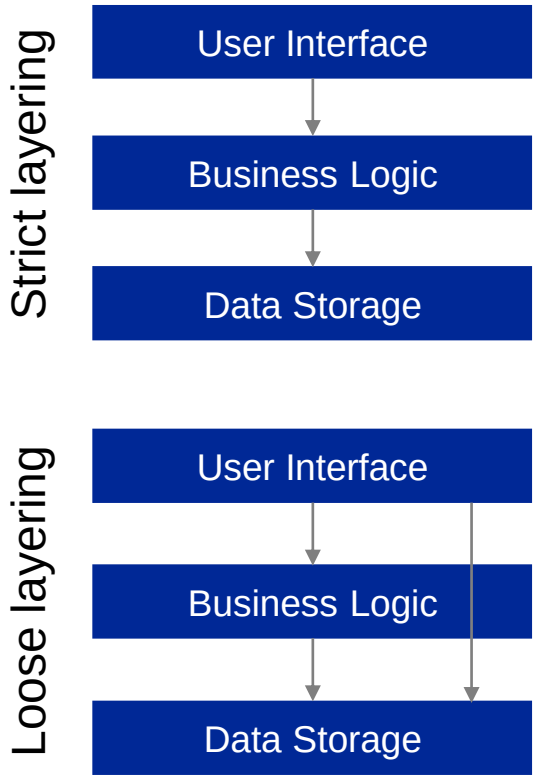
Discussion: Are there still monoliths used? Should you use?

LAYERED ARCHITECTURE (1)

Different aspects of the software are separated in layers (aka tiers)

Two common alternatives:

- Strict layering
  Functionality of one layer is only accessible from the superior layer
- Loose layering
  Functionality of lower layers can be accessed by any higher layers



LAYERED ARCHITECTURE (2)

> Evaluation
- Reduces complexity
- Strong cohesiveness, high logical coherence within the code
- Downside: Increased runtime due to forwarded calls (through layers)

> Well known example:
OSI model

→ Do you know how this particular architecture is usually implemented?

Layer architecture [edit]

The recommendation X.200 describes seven layers, labeled 1 to 7. Layer 1 is the lowest layer in this model.

OSI model				
Layer			Protocol data unit (PDU)	Function[6]
Host layers	7	Application	Data	High-level APIs, including resource sharing, remote file access
	6	Presentation		Translation of data between a networking service and an application; including character encoding, data compression and encryption/decryption
	5	Session		Managing communication sessions, i.e. continuous exchange of information in the form of multiple back-and-forth transmissions between two nodes
	4	Transport	Segment, Datagram	Reliable transmission of data segments between points on a network, including segmentation, acknowledgement and multiplexing
Media layers	3	Network	Packet	Structuring and managing a multi-node network, including addressing, routing and traffic control
	2	Data link	Frame	Reliable transmission of data frames between two nodes connected by a physical layer
	1	Physical	Symbol	Transmission and reception of raw bit streams over a physical medium

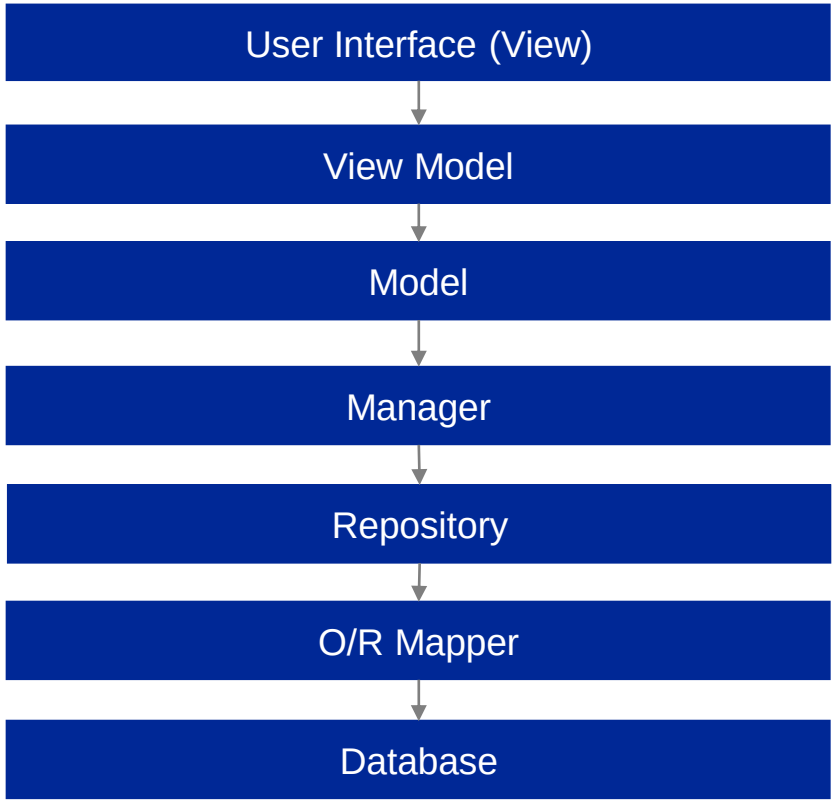
Source: https://en.wikipedia.org/wiki/OSI_model

LAYERED ARCHITECTURE: EXAMPLE

Real World Example

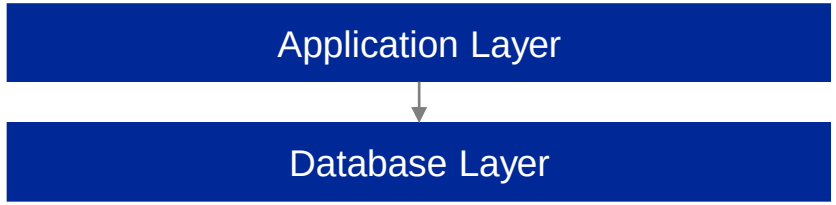
Let's discuss:

- First impression?
- Strict or loose layered?
- Advantages/disadvantages?
- Does it reduce complexity?
- What could be the reason for such a design decision?



TWO TIER ARCHITECTURE

Common layered architecture



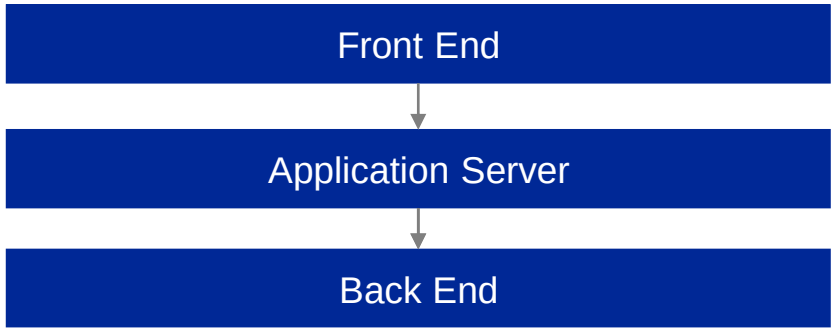
> *Two tier architecture*
> Superior layer accessing the lower one
> Possible advantages?
  Changes in the lower layer are transparent as long as interfaces stay the same
  Layers represented by different software components on the same system
  Layers are distributed over multiple systems

Also known as *client server architecture* (proceeding slides)

LAYERED ARCHITECTURE (4)

Common layered architecture

> Three tier architecture

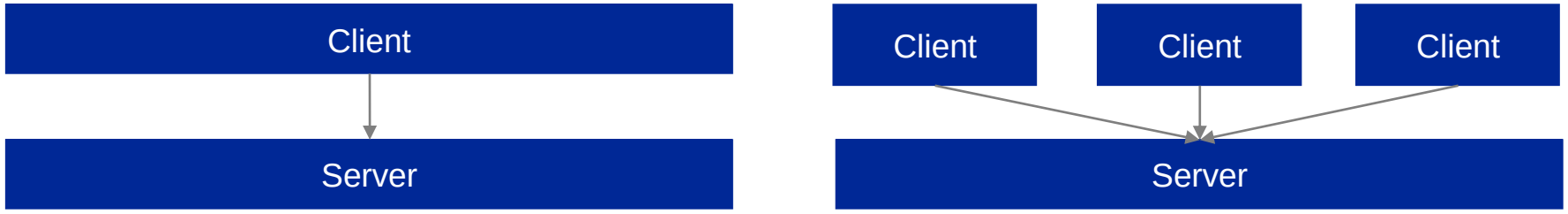


Typical layers

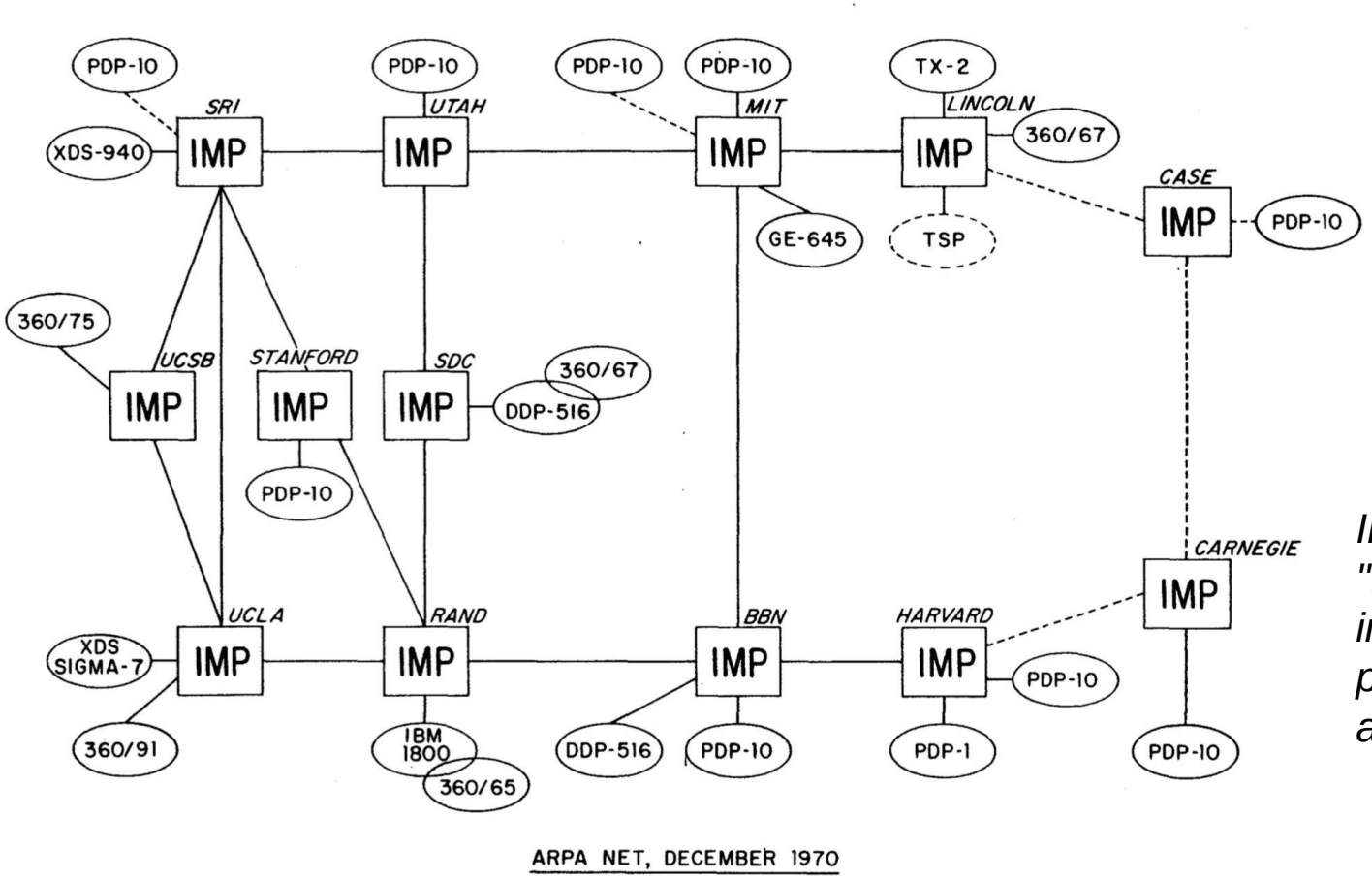
> Front end or client tier
> Application server tier, enterprise tier, or middle tier
> Back end or data server tier
> Stereotype for Web development

CLIENT SERVER ARCHITECTURE

- „*The*“ architecture for distributed systems
- Distribution of computing resources
- Server provides services used by clients
- Very first application ARPANET*) during the 1960s and 1970s
- Basic architecture of the World Wide Web (WWW)



EXAMPLE: ARPA NET 1970



In fiscal year 1969 an ARPA program entitled "Resource Sharing Computer Networks" was initiated. The research carried out under this program has since become internationally famous as the ARPANET.

Source: ARPANET Completion Report 4799, 1978

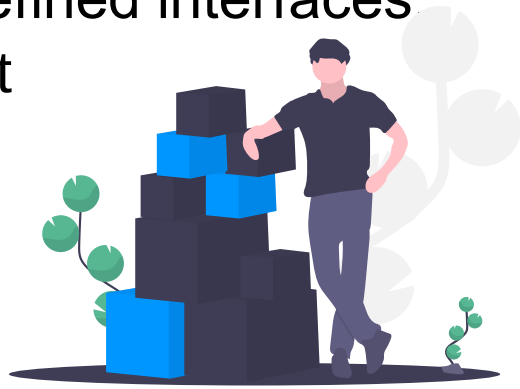
COMPONENT-BASED ARCHITECTURE

- Separation of the design into
  logical and
  functional components

- Overall goal: reusability of components
- Components
  encapsulate functionality and behavior as deployable artifacts
  Examples: COM, JavaBeans, EJB, .NET assemblies or Web Services

CHARACTERISTICS OF COMPONENTS

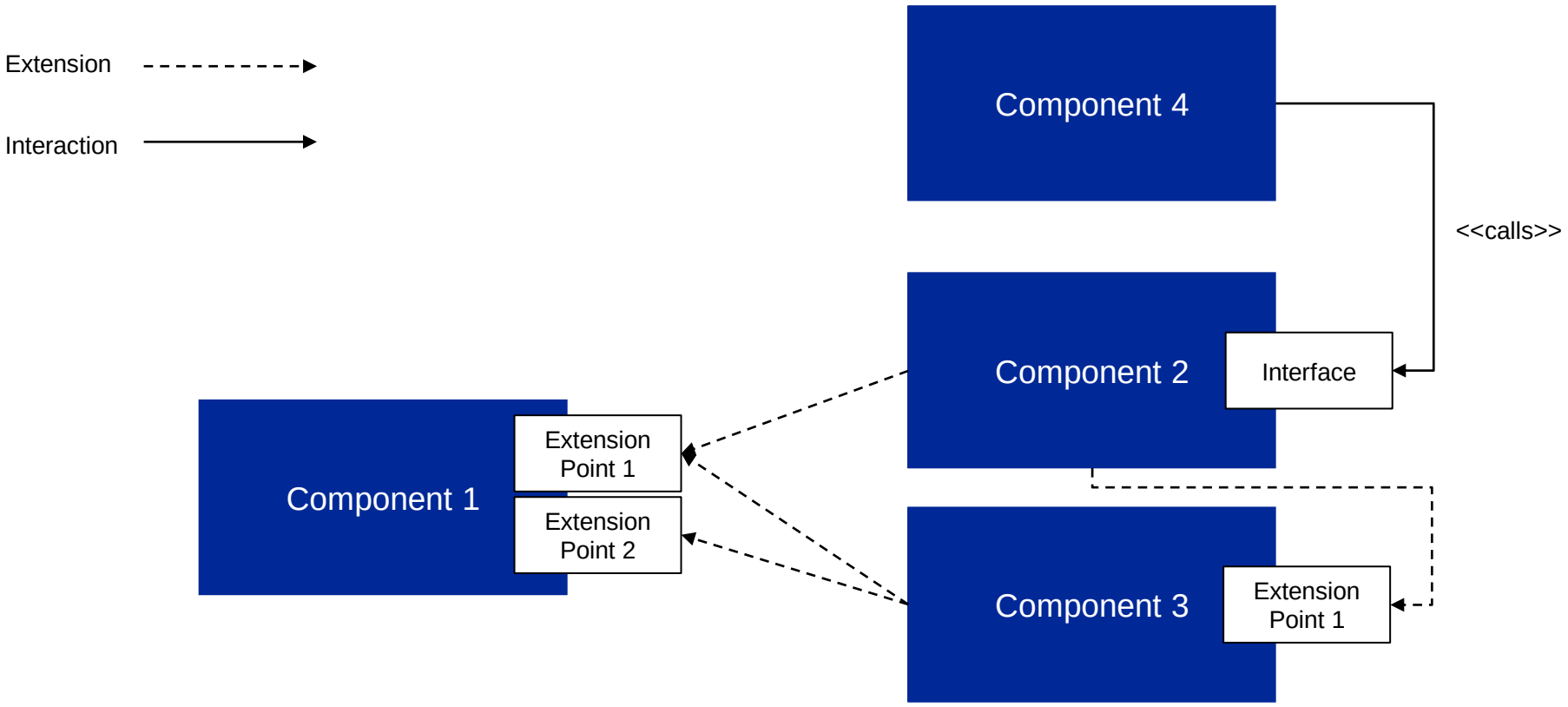
- **Reusability** − reusing components in different applications
- **Replaceable** − replace component by other components
- **Not context specific** − a component can be used in different environments and contexts
- **Extensible** − components can be extended based on already existing components to provide new functionality or behavior
- **Encapsulated** − access to a component is strictly encapsulated by defined interfaces internal states and processes are not visible outside of the component
- **Independent** − components do not depend on other components



PRINCIPLES OF COMPONENT BASED ARCHITECTURES

- Obvious: The system is split into (reusable) components
- Every component
  - specifies its dedicated interface
  - as well as the required ports
  - encapsulates implementation details and
  - provides extension ports to be extended (optional)
- Interaction between components takes place via
  - method calls
  - message-based communication
  - data streams or other protocols

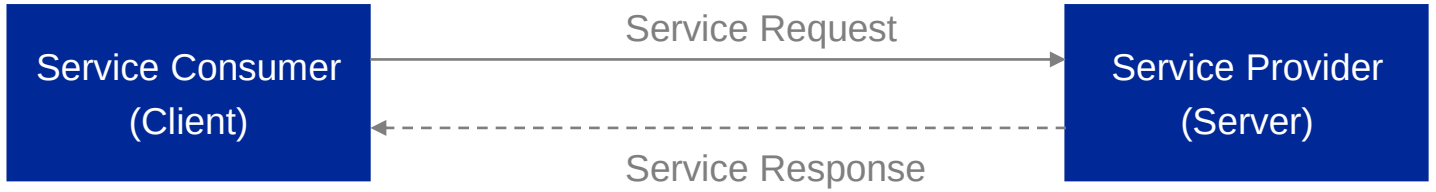
EXTENDING COMPONENTS



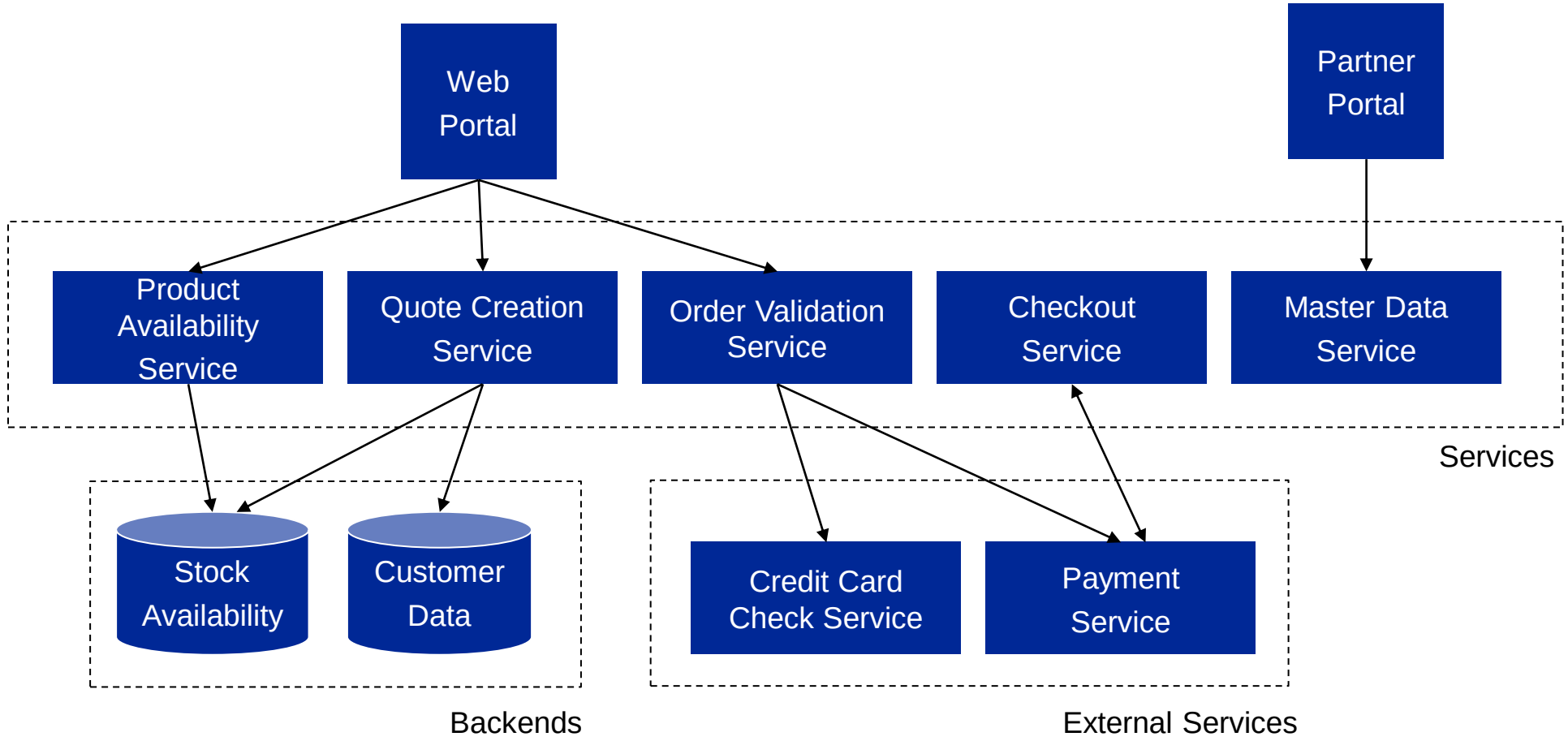
SERVICE-ORIENTED ARCHITECTURES

- aka SOA
- New software components are developed as „services“
- Existing components can be encapsulated as services
- Services provide self-explaining interfaces
- Services provide quality of service (QoS) standards
  - Security
  - Availability
  - Response Times
  - Policies (e.g. 5k calls within 30 sec)

BASIC PRINCIPLE OF SERVICE COMMUNICATION



SOA EXAMPLE


Web Portal
Partner Portal
Product Availability Service
Quote Creation Service
Order Validation Service
Checkout Service
Master Data Service
Services
Stock Availability
Customer Data
Backends
Credit Card Check Service
Payment Service
External Services

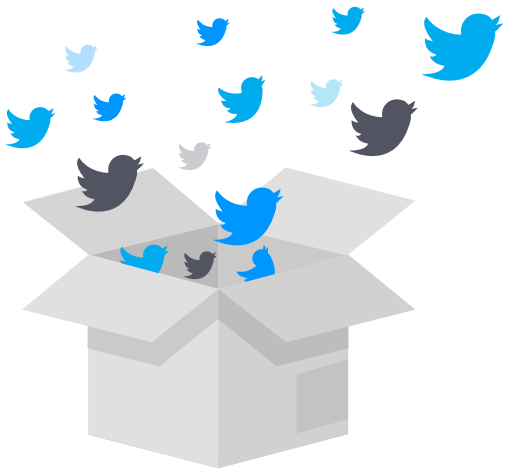
CHARACTERISTICS OF SERVICE

- Are accessible via a network
- Complete / autonomous
- Provide a well-defined interface
- Hide internal details of their implementation (technology, data structures, algorithms)
- Platform independent (usage perspective)
- Dynamical bound (don’t have to be available at implementation time)
- Listedin a registry

REST ARCHITECTURES

Famous because of?

- RESTful APIs (Application Programming Interfaces)
- Programming interfaces bing called over HTTP/S
- Example Twitter
  - https://twitter.com/aheil
  - https://twitter.com/notifications
  - https://twitter.com/compose/tweet
  - https://twitter.com/search?q=softwarearchitecture
  - https://twitter.com/hashtag/softwarearchitecture



Discussions: Is the Twitter application a REST Architecture because it provides a RESTful API?

REPRESENTATIONAL STATE TRANSFER (REST)

- Based on the dissertation of "*Architectural Styles and the Design of Network-based Software Architectures*" by Roy Fielding
- Describes *one* possible architecture style for systems distributed over networks
- Also describes the state-of-the-art architecture of the Web (≠ Internet)
  Coincidence: Fielding was co-author of the HTTP protocol
- Goal: Good design for distributed Web applications

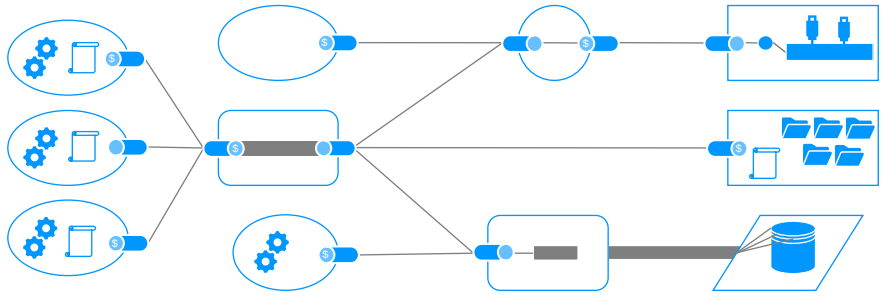
> *"Representational State Transfer is intended to evoke an image of how a **well-designed Web application** behaves: a network of web pages (a virtual state-machine), where the user progresses through an application by **selecting links** (state transitions), **resulting in the next page** (representing the next state of the application) being **transferred to the user** and rendered for their use."*
>
> *Roy Fielding*

REST ARCHITECTURE

- *An architectural approach* – not an API nor a product or similar
- Focus on existing standards
  HTTP(S), XML, URIs (Uniform Resource Identifier)
- Allows to build Web-based application
- Based on URIs
- Concept of a state machine
- Statelessness as a performance indicator (allows caching)
- Focusing on restrictions
  Client-Server, statelessness, unified interfaces, cache, gateways and proxies
- It turns out that REST is an optimal architecture for the Web (coincidence?)

REST RESTRICTIONS

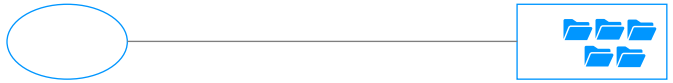
- REST is using a set of restrictions the architectures are based on
- We will investigate the most important restrictions in detail below
  - Client-Server model
  - Statelessness
  - Cache
  - Unified Interfaces
  - Multi-tier system
  - Code-on-Demand
  - Data



REST DETAILS – CLIENT-SERVER MODEL

Restriction 1: Client-server mode

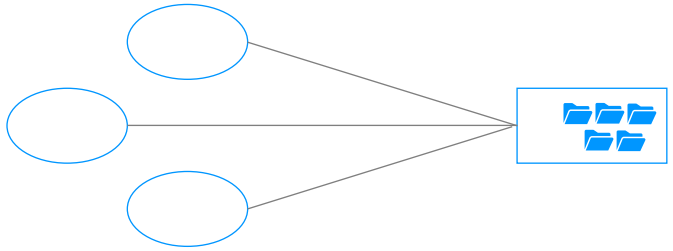
- Basic principle: Separation of Concerns – User Interface and data are strictly separated
- Increases scalability due to reduction of complexity on the server
- Separation allows independent development of components



REST DETAILS – STATELESSNESS (1)

Restriction 2: Stateless

- Discussion: What means stateless? What is the difference between stateless and stateful?
- In REST: Communication is stateless
- A request always does include all information necessary to understand the request
- Session information rest on the client side, the server does not know anything about the state of its clients



REST DETAILS – STATELESSNESS (2)

Advantages

> In case of an (server side) error, the system can quickly recover as you do not have to consider any
> Increases scalability as the server does not store any states
> Reduces complexity on the server side

Discussion

> Does this always work?
> Do you know any Web application where this does not work?

REST DETAILS – STATELESSNESS (2)

Disadvantage

- Overhead due to repeatedly sent data
- Example: Browser F5
- Server has no control over the application
- I.e. the server has no control what the application does with the data

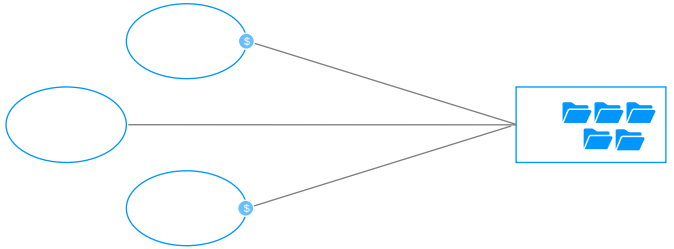
REST DETAILS – CACHE

Restriction 3: Cache

> Data in responses must be marked explicitly as cacheable
> If not, clients are not allowed to cache the data (does this work?
> Cacheable data can be used for similar answers without requesting them again from the server

Advantage

> Reduces amount of data
> Increases scalability



Disadvantage

> Decreasing reliability when the data differs on client and server side

REST DETAILS – UNIFIED INTERFACES

Restriction 4: Unified Interfaces

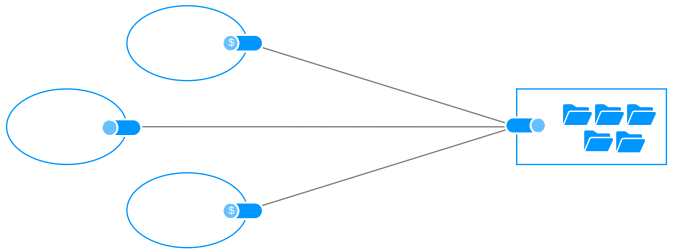
> Main characteristic of the REST architecture style
> Equivalent of a universal component interface in software engineering

Advantages

> Allows loose coupling between components / services

Disadvantages

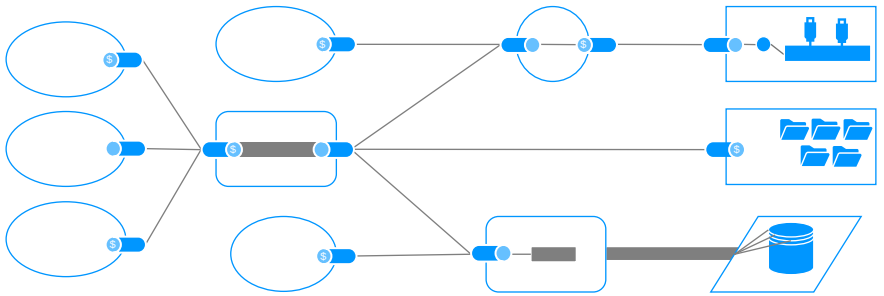
> Inefficiency: data is not transmitted on the best format for the application
> Data must be transformed before and after the transmission



REST DETAILS – MULTI TIER SYSTEM (1)

Restriction 5: Multi tier system

- Multiple hierarchical layers
- Interaction is strictly limited to the next layer



REST DETAILS – MULTI TIER SYSTEM (2)

Advantages

- Allows isolation/encapsulation legacy systems
- Allows the protection of legacy systems from new systems
- Increases scalability
  (e.g. by preceding load balancers layer)
- Tiers can process information independently
  (remember messages are stateless and contain all information)

Disadvantages

- Additional overhead caused by multiple processing of the data in different layers
- Increased latency

REST DETAILS – CODE ON DEMAND

Restriction 6: Code-On-Demand

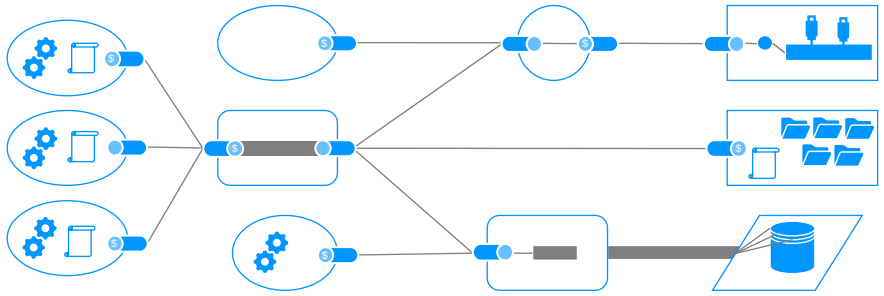
> In terms of the REST architecture style: optional
> Client functionality is extended by code on demand
> Todays browser technology is based on this concept:
> JavaScript, Java Applets, Java Web Start, formerly Microsoft Silverlight

Advantages

> Lightweight clients

Disadvantages

> Reduced transparency
> Attack vectors for servers are
> Directed towards clients



MICROSERVICE ARCHITECTURE (1)

Indicators for Microservice architectures (technical)

- Server-based applications
- Accessed by various clients (Web, Mobile, Desktop)
- Optional: API for 3rd party usage
- Interaction with further applications using Web services or a message broker
- Message exchange with additional applications
- Logical components, each covering another area
- Usage of HTML, JSON and or XML

MICROSERVICE ARCHITECTURE (2)

Further indicators for Microservice architectures (organizational)

- Multiple developer teams working on the application
- New team members should be productive very quickly
- Application should be easy to understand
- Application should easily be maintained and extended
- Continuous deployment of the application
- Scalability by paralell deployment of the application / parts of the application

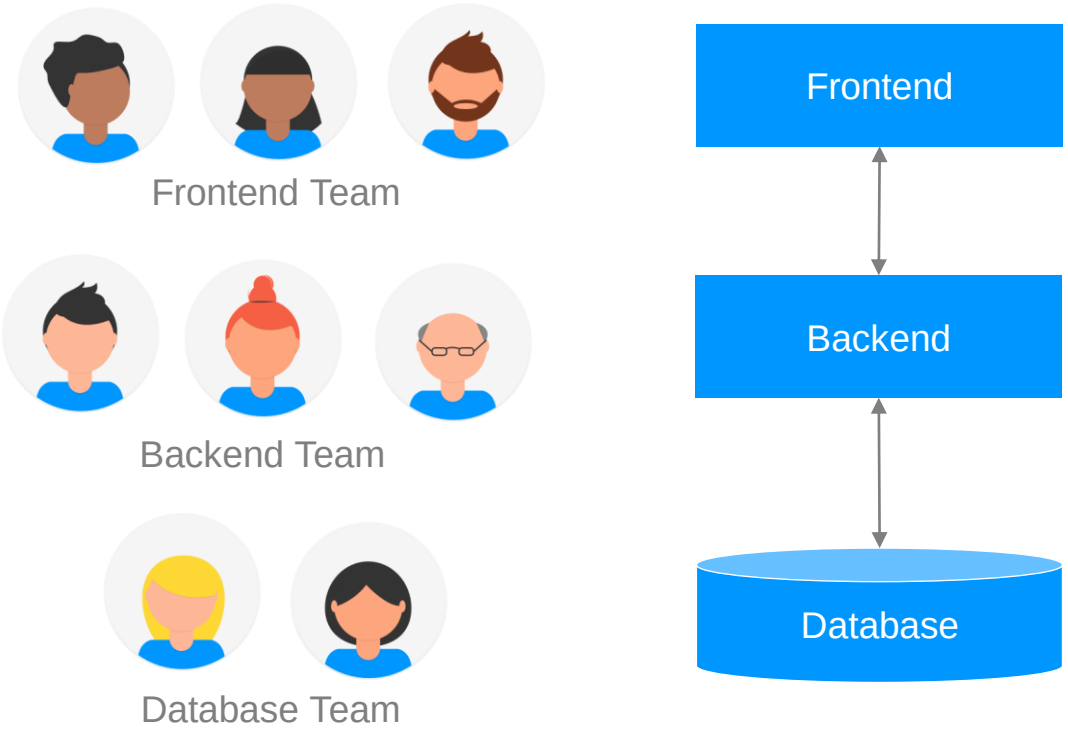
MICROSERVICE ARCHITECTURE (3)

Basic architecture concepts

- Application structured as loosely coupled, cooperating services
- Every micro service implements a **small**, **independent** set of **coherent** functionality
- Usage of synchronous protocols (e.g.. HTTPS) and/or asynchronous messaging or streaming (e.g. AMPQ, JMS, Kafka)
- The term *micro* indicates a **small set of business** functionality!
- Eventually, Microservices architecture is a lot about how to structure your teams and how to deploy your software

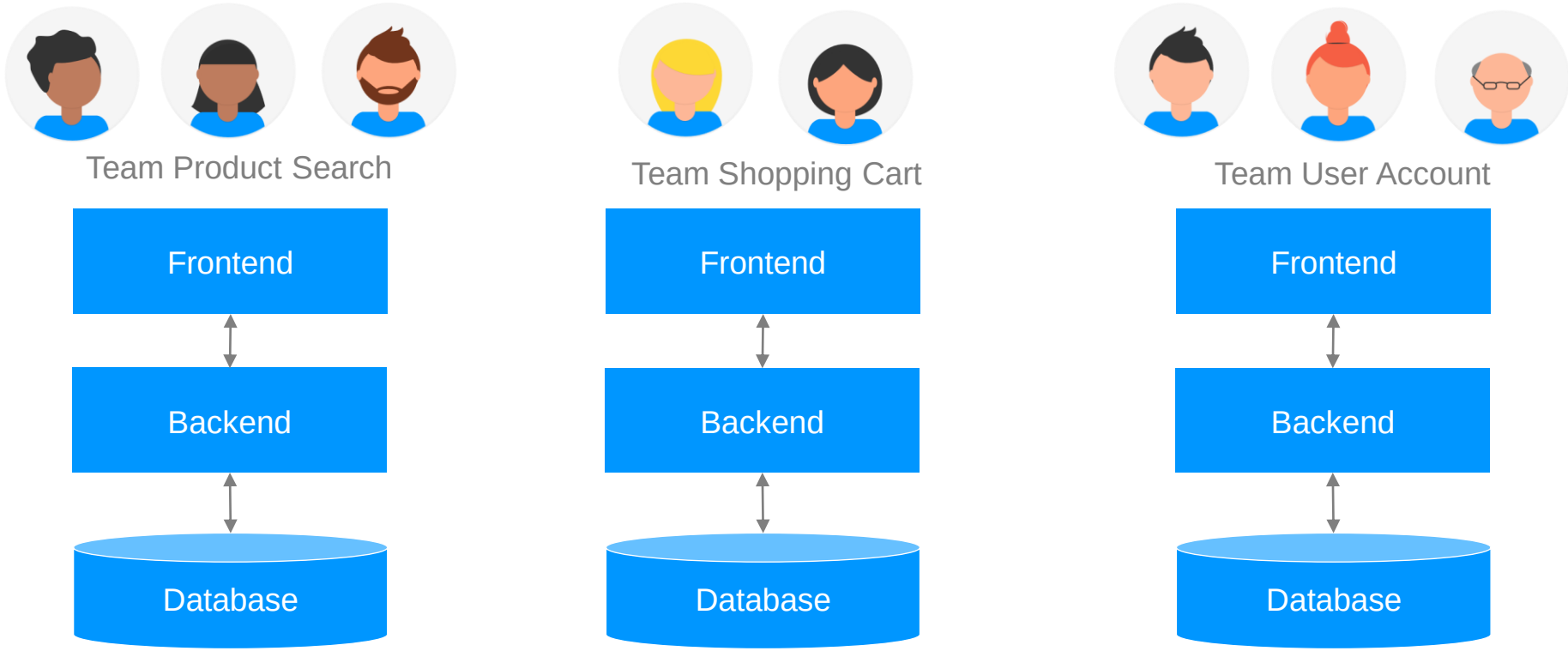
MICROSERVICE ARCHITECTURE (4)

"Classical" structure of teams

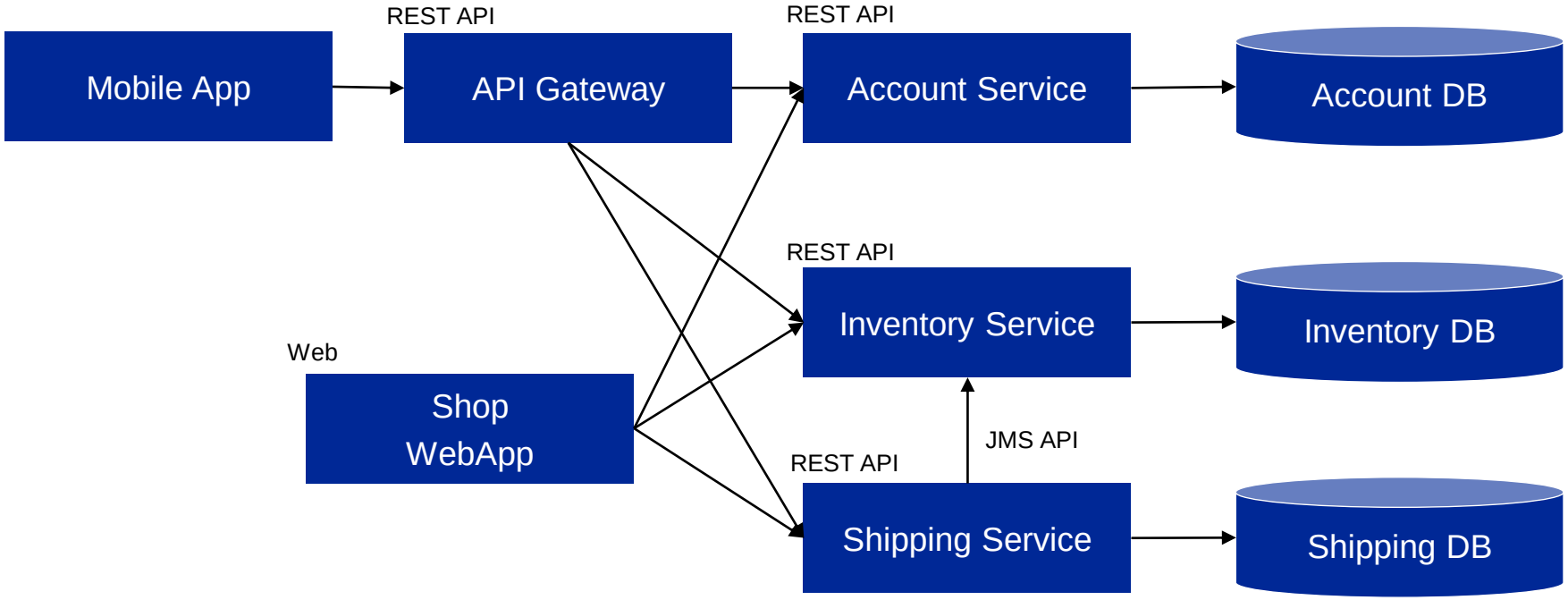


MICROSERVICE ARCHITECTURE (5)

Cross-functional teams based on business case/area



MICROSERVICE ARCHITECTURE EXAMPLE



MICROSERVICES – MANY ADVANTAGES

Allows Continuous Delivery und Deployment of larger scale applications

- Increased testability: small services are much easier to test
- Easy deployment: independent deployment and roll out of services (WARNING: only if interfaces are stable)
- Development can be distributed over multiple teams, „two pizza teams“ can cover all aspects of the service lifecycles (development, deployment, operations, monitoring)
- Each Microservice is relatively small
- Easy to understand (for developer)
- Side effect: IDE will perform better due to smaller codebase and less dependencies
- Application starts rapidly, increasing development and test cycles
- Increased error isolation
- Example: memory leak in one service affects only this service instead of the overall application
- No long-term technology commitment
- Every service could used different technologies or versions for development, deployment

MICROSERVICES – SOME DISADVANTAGES

Developers must deal with increased complexity of a distributed application

Testing of the overall application is more complex and expensive

Additional „inter-service“ mechanisms have to be developed

Problem of „distributed transaction“

- Use cases without distributed transactions are rarely possible
- Use cases including multiple services (=multiple teams) are more complex, development, test and deployment need to be synchronized across teams

MICROSERVICES DESCISSION MAKING

In the beginning Microservices usually do not solve your existing problems

Later, once the Microservice architecture comes in handy, the problem of decomposition of services arises

Apply common principles to solve this issue

- SRP – Single Responsibility Pattern
- CQRS – Command Query Responsibility Pattern
  Also you will need queries spanning multiple services

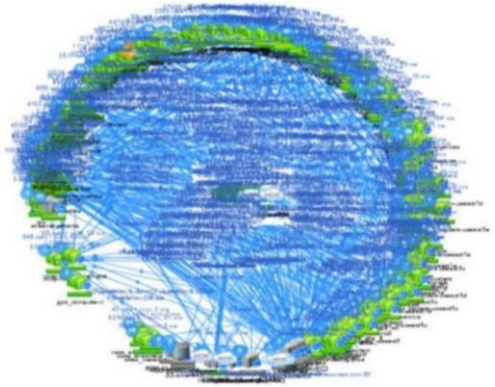
Again, use common principles

- API Composition
- CQRS Pattern

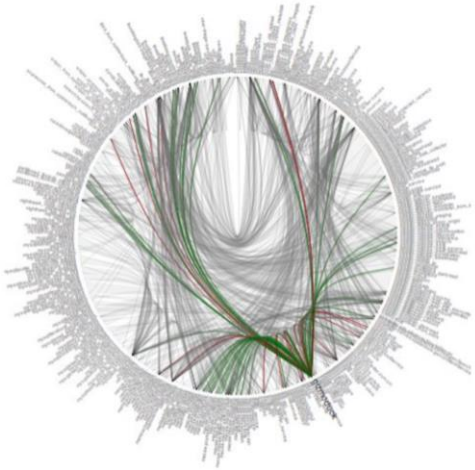
MICROSSERVICES – REFLECTION

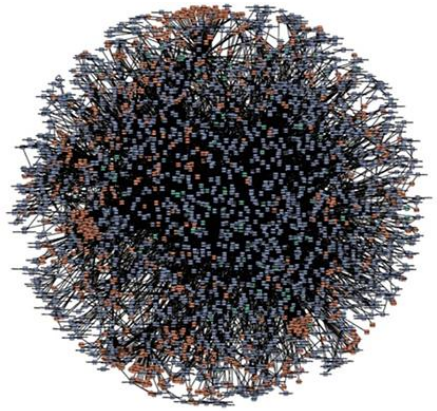
- Microservice architectures are deployment structures
- Not all applications (only few applications) are auditable for a microservice architectures
- Refactoring an old application is sometimes more beneficial than redesigning it as microservice architecture
- Transactions are the key challenge
- Overhead and complexity of the overall application must not be neglected

DEATH STAR DIAGRAMS



NETFLIX





amazon.com

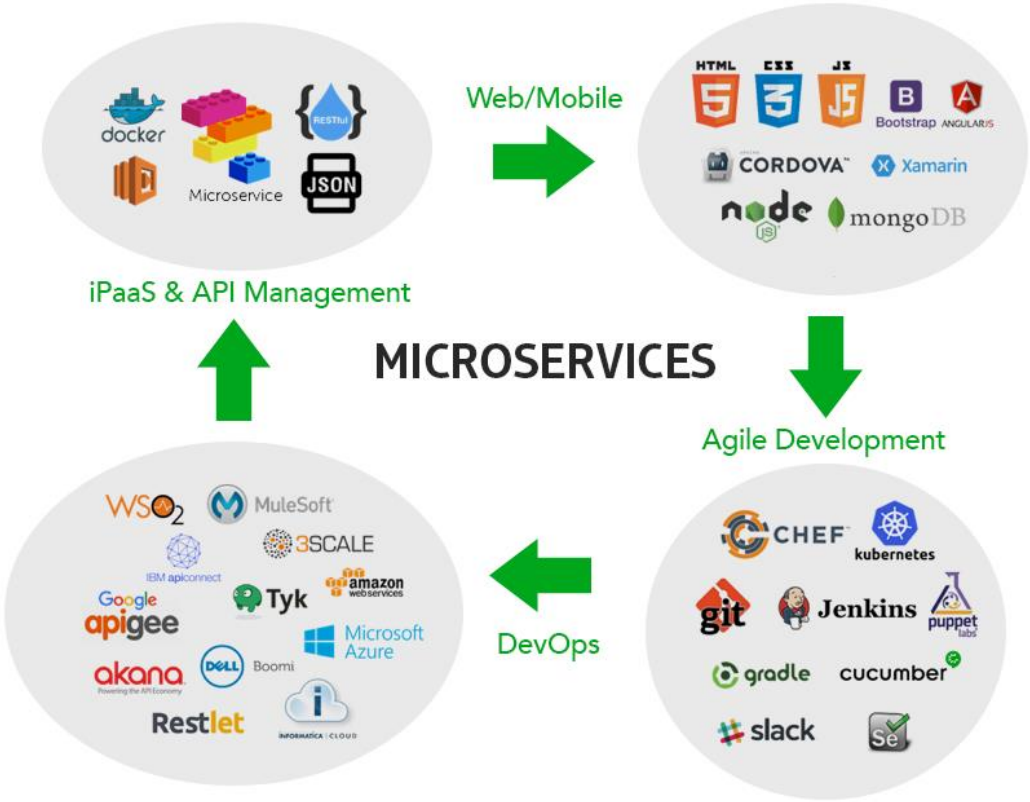
THE TOOLCHAIN IS THE KEY

Why does it work at?

The tools/toolchains are the key

Tools allow

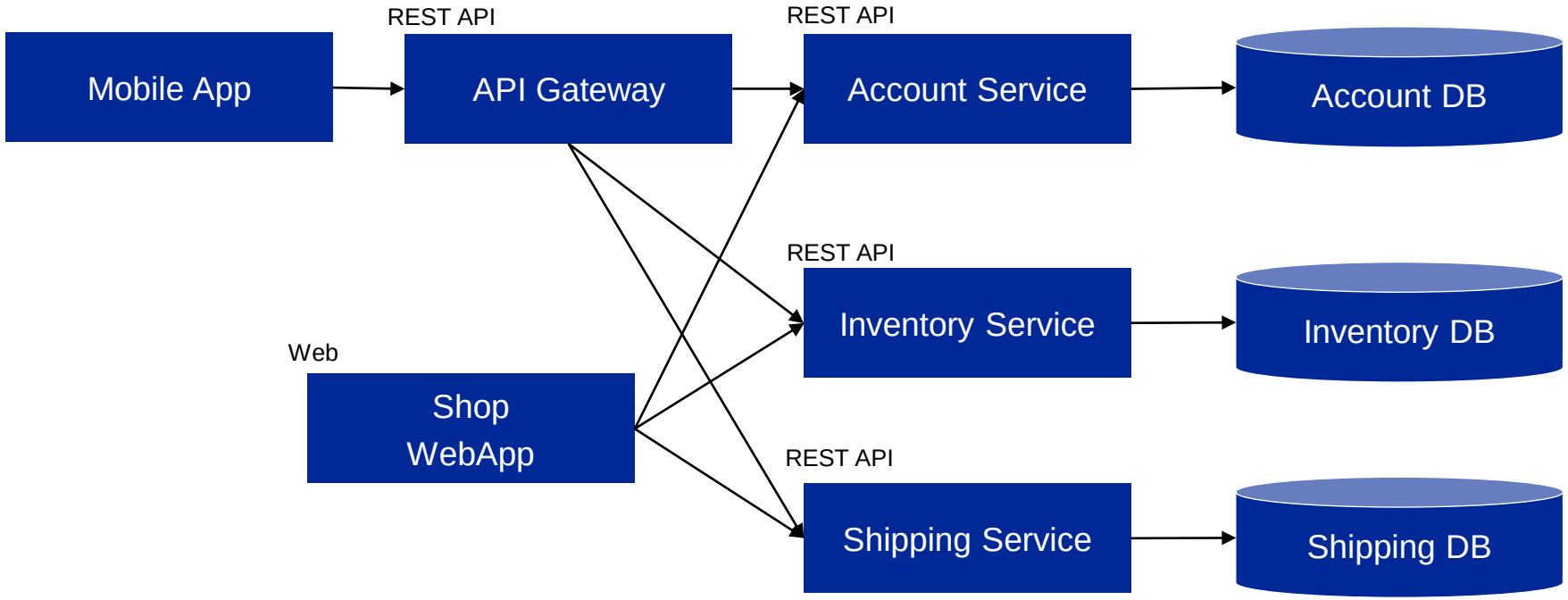
- rapid development
- easy testing
- to deploy fast and often
- fast correcting of error
- efficient maintenance of code
- fast delivery of functionality



Source: https://www.appcentrica.com/the-rise-of-microservices/

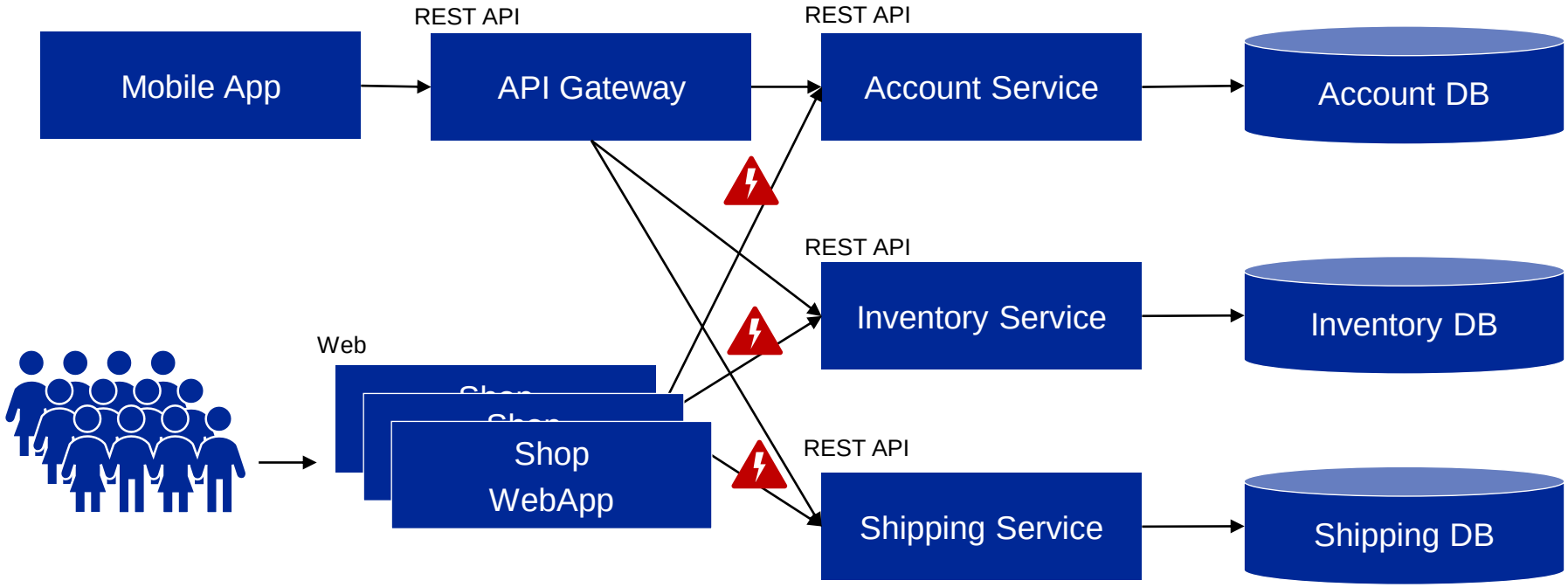
EVENT-BASED MICROSERVICE ARCHITECTURES

Scalability issues of microservice infrastructures



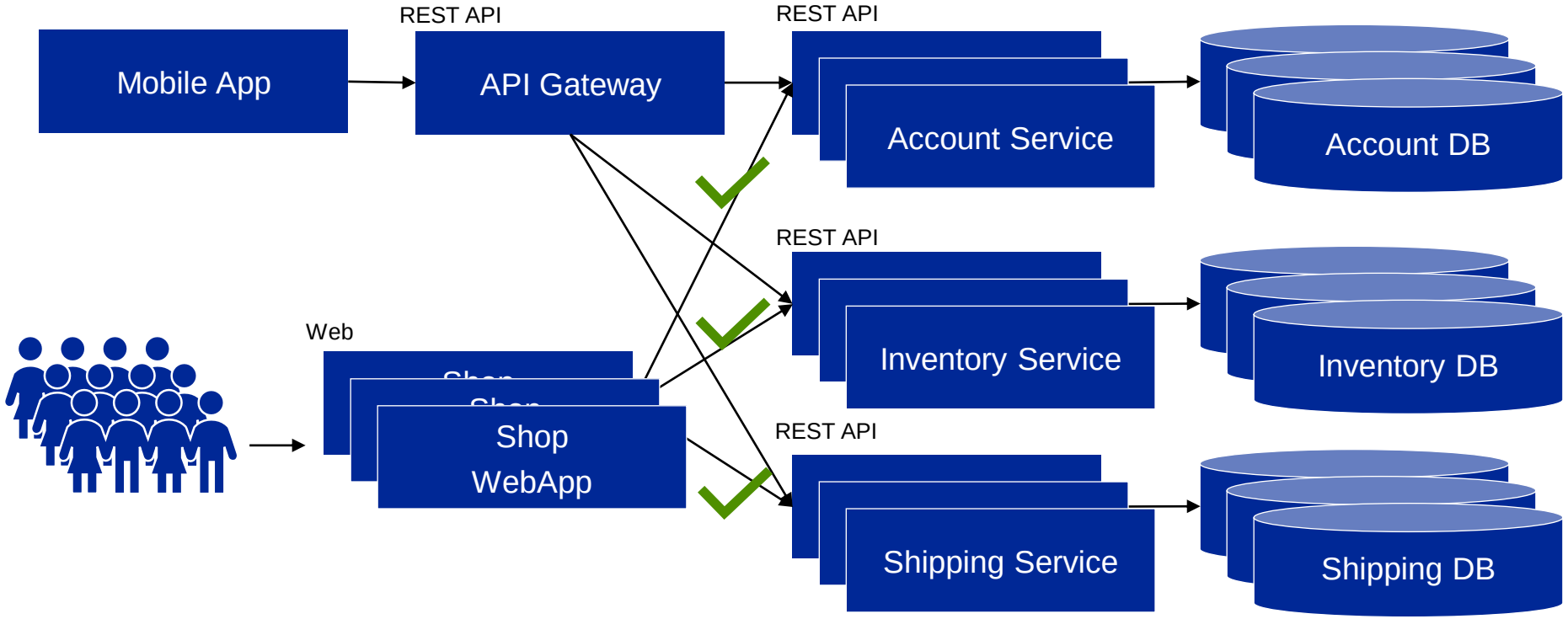
EVENT-BASED MICROSERVICE ARCHITECTURES

Scalability issues of microservice infrastructures



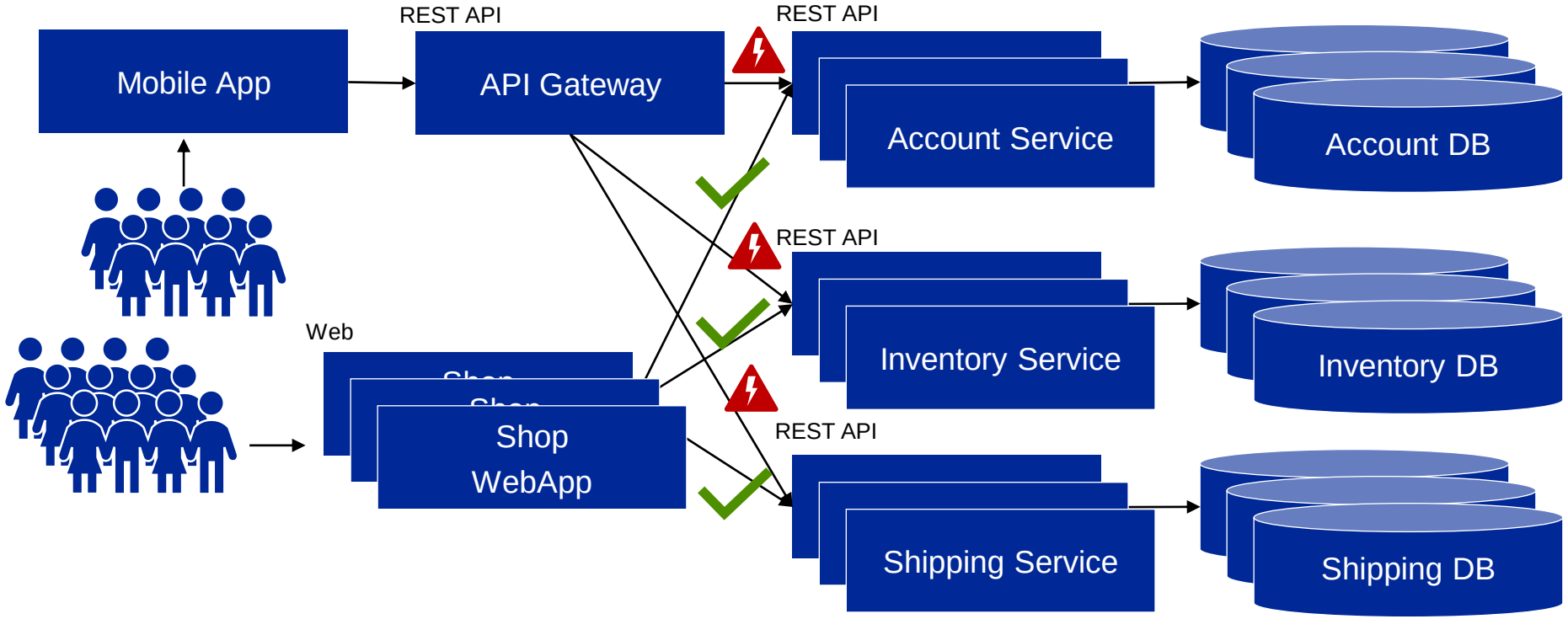
EVENT-BASED MICROSERVICE ARCHITECTURES

Scalability issues of microservice infrastructures



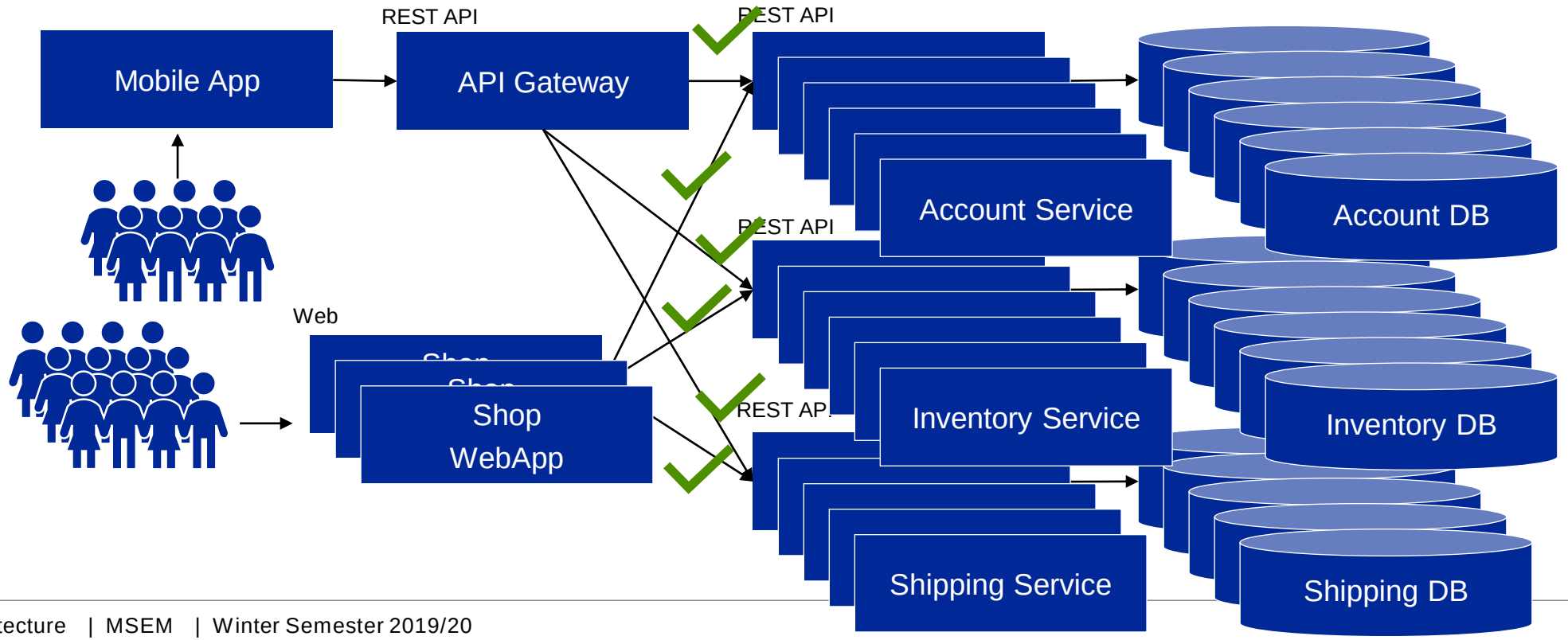
EVENT-BASED MICROSERVICE ARCHITECTURES

Scalability issues of microservice infrastructures



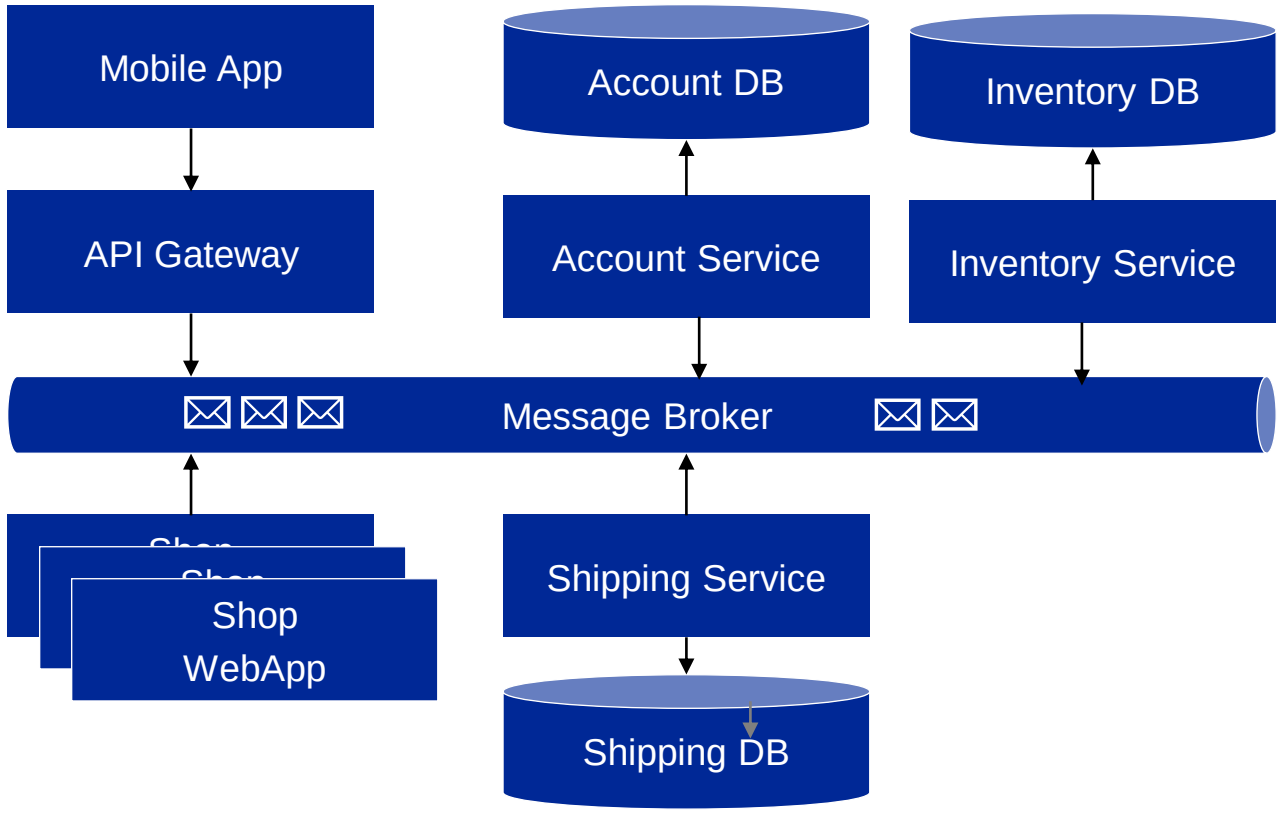
EVENT-BASED MICROSERVICE ARCHITECTURES

Scalability issues of microservice infrastructures



EVENT-BASED MICROSERVICE ARCHITECTURES

Solution



EVENT-BASED MICROSERVICE ARCHITECTURES

Origin:

- Large number of Microservices communicating with each other
- One or many Microservices generating a (very) large number of events to be processed by multiple other services
- Choose a communication pattern which is loosely coupled (less coupled than synchronous HTTP/S calls)
- Choose an asynchronous communicating pattern

EVENT-BASED MICROSERVICE ARCHITECTURES

Advantages

- Services are loosely coupled and can be maintained independently
- Messages are buffered and processed as soon as a service is available
- Services are scalable independently to process a high number of messages
- Messaging platforms/broker can set up highly available and redundant
- Message producers and consumers can (still) be implemented in different technologies

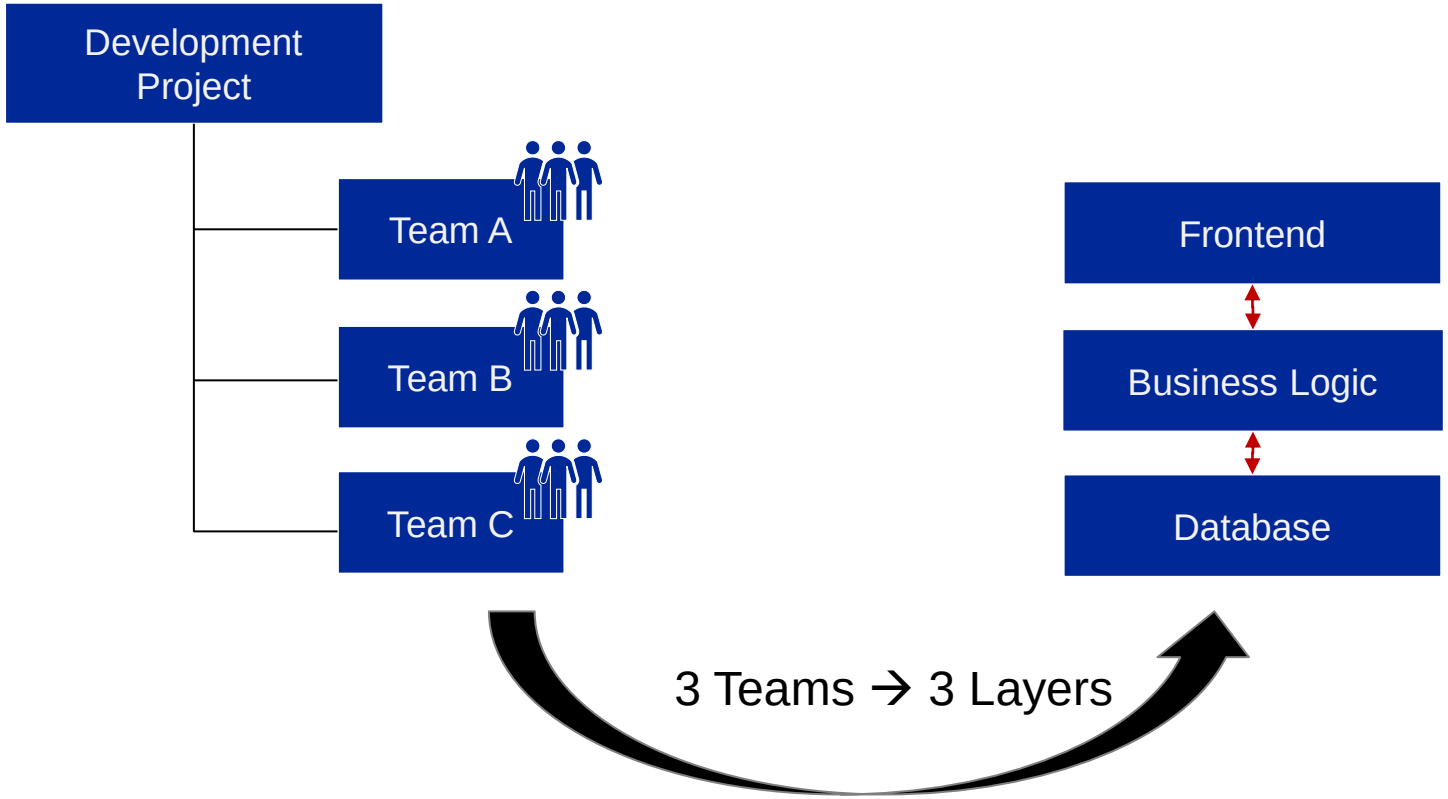
CONWAY‘S LAW

How do communication structures affect your architecture?

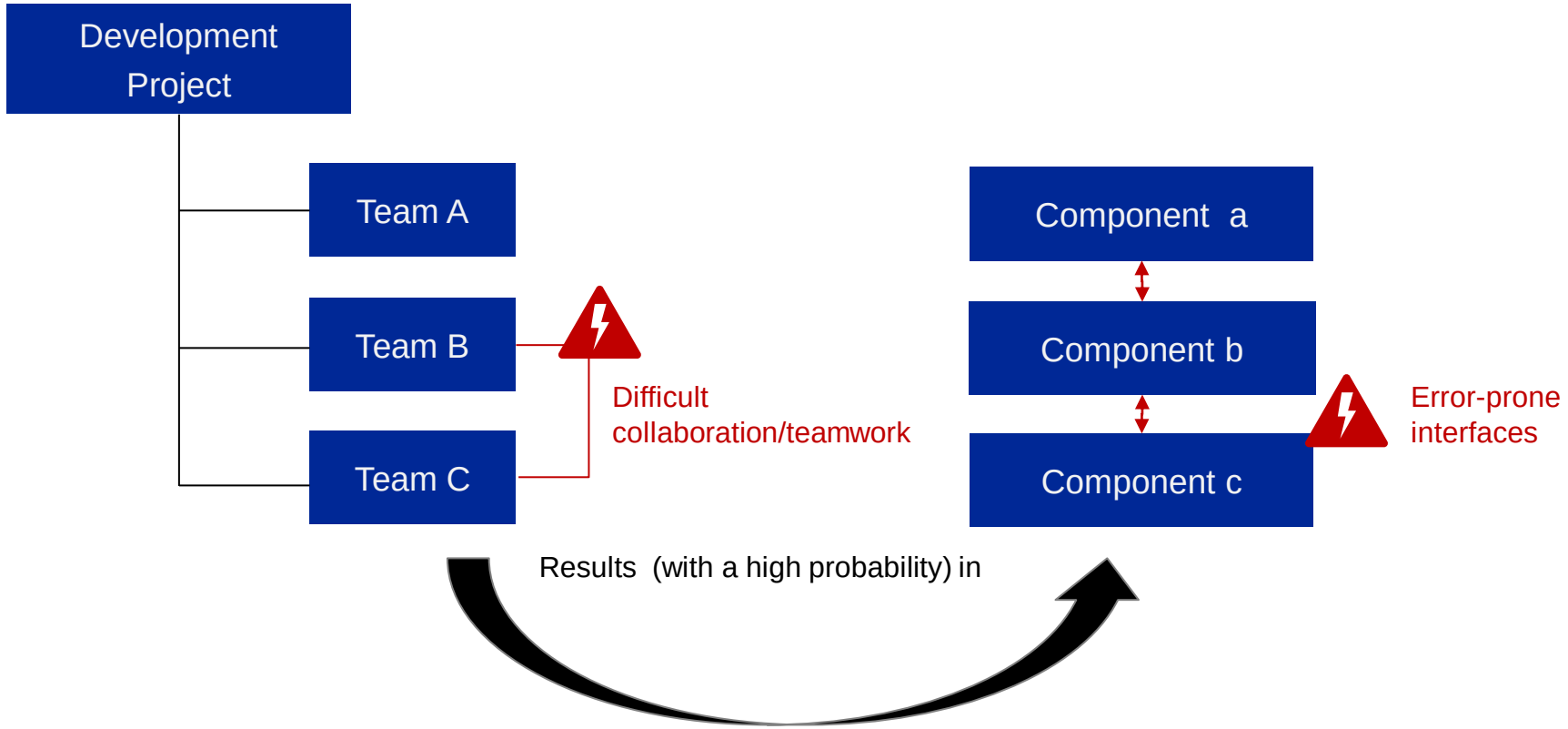
> *“Any organization that designs a system (defined more broadly here than just information systems) will inevitably produce a design whose structure is a copy of the organization's communication structure.”*
>
> Mel Conway, http://www.melconway.com/Home/Committees_Paper.html

In other words: The communication within your teams and/or organization will strongly influence your architecture.

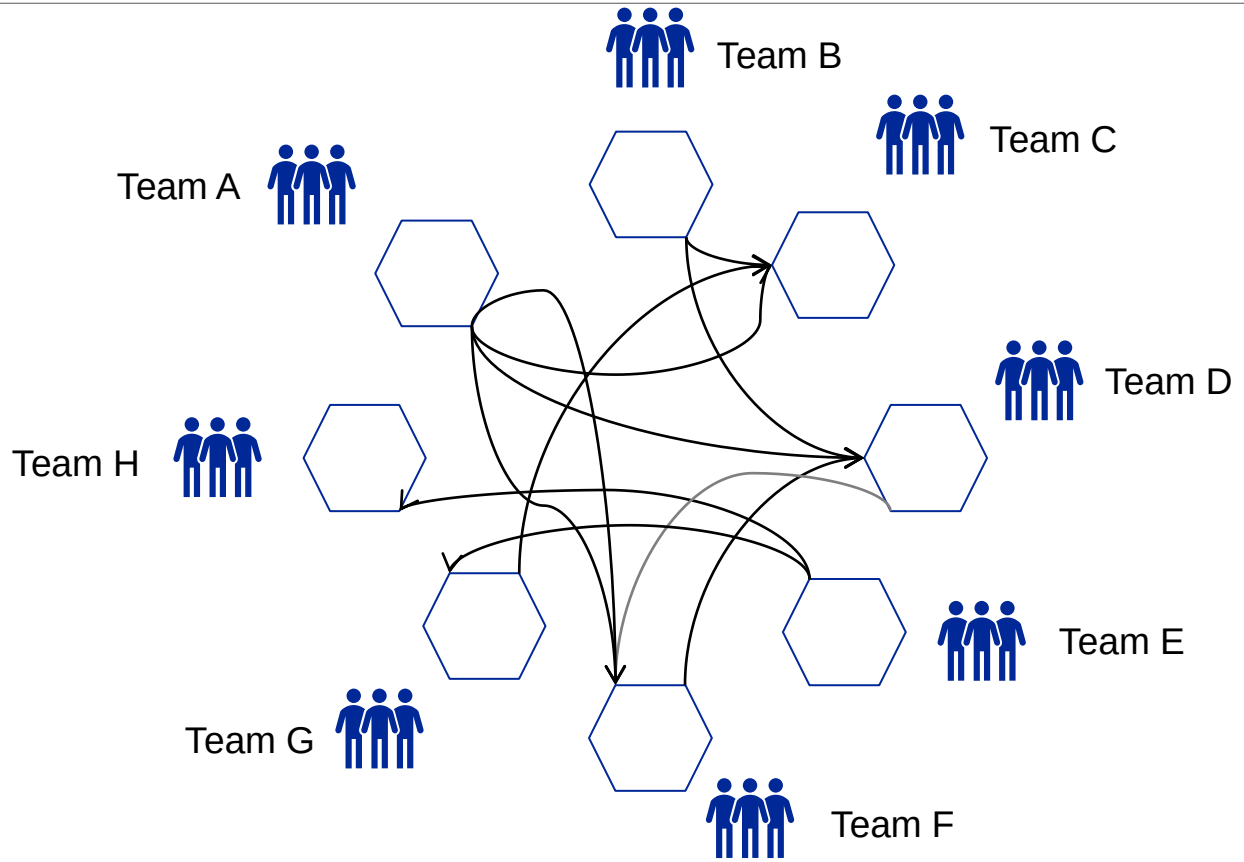
CONWAY'S LAW EXAMPLE: 3-TIER



CONWAY'S LAW: EXAMPLE

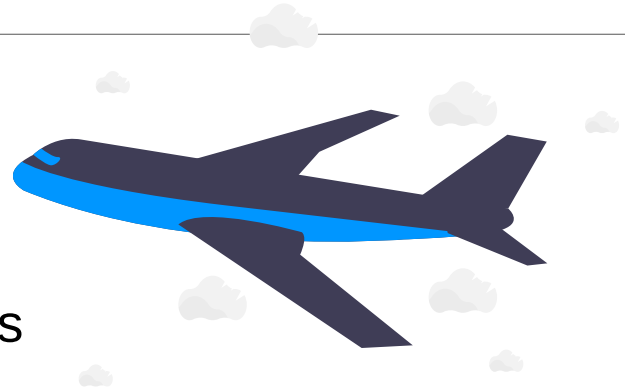


CONWAY‘S LAW EXAMPLE: MICROSERVICES



CARGO-CULT

Some words about architecture hypes



What is a Cargo-Cult?

> Applying practices without fully understanding the underlying principles

Origin

> During US occupation of New Guinea occupants brought a lot of „great“ cargo to the natives

Consequences

> Once the military abandoned the airbases, the natives did built airbases, control towers and planes out of litter, set up fires on the runways hoping to attract planes

CARGO CULT: MICROSERVICES

Example: Microservices as general solution! Why?

- Some (indeed only very few) large companies succeeded with introducing these concepts
- In contrast the monolith was propagated as “bad“ solution
- But: Microservices do not suite all problems

- The **business model of Netflix** (delivering movies) is **quite different** from the **business model of healthcare insurance agencies** (managing patient information, accounting and billing)
- A Large amount of all applications are probably well suited based on their current architecture
- Ever architecture comes with its very own advantages and disadvantages

- **Thus, understand WHY a certain architecture is suited for your problem!**

MICROSERVICES VS MONLITH – REALITY CHECK

23. Microservice envy

Microservices has emerged as a leading architectural technique in modern cloud-based systems, but we still think teams should proceed carefully when making this choice. **Microservice envy** tempts teams to complicate their architecture by having lots of services simply because it's a fashionable architecture choice. Platforms such as Kubernetes make it much easier to deploy complex sets of microservices, and vendors are pushing their solutions to managing microservices, potentially leading teams further down this path. It's important to remember that microservices trade development complexity for operational complexity and require a solid foundation of automated testing, continuous delivery and DevOps culture.

ThoughtWorks

Unknown

Why a Monolith works

Once the code for all destinations lived in a single repo, they could be merged into a single service. With every destination living in one service, our developer productivity substantially improved. We no longer had to deploy 140+ services for a change to one of the shared libraries. One engineer can deploy the service in a matter of minutes.

The proof was in the improved velocity. In 2016, when our microservice architecture was still in place, we made 32 improvements to our shared libraries. Just this year we've made 46 improvements. We've made more improvements to our libraries in the past 6 months than in all of 2016.

The change also benefited our operational story. With every destination living in one service, we had a good mix of CPU and memory-intense destinations, which made scaling the service to meet demand significantly easier. The large worker pool can absorb spikes in load, so we no longer get paged for destinations that process small amounts of load.

Segment

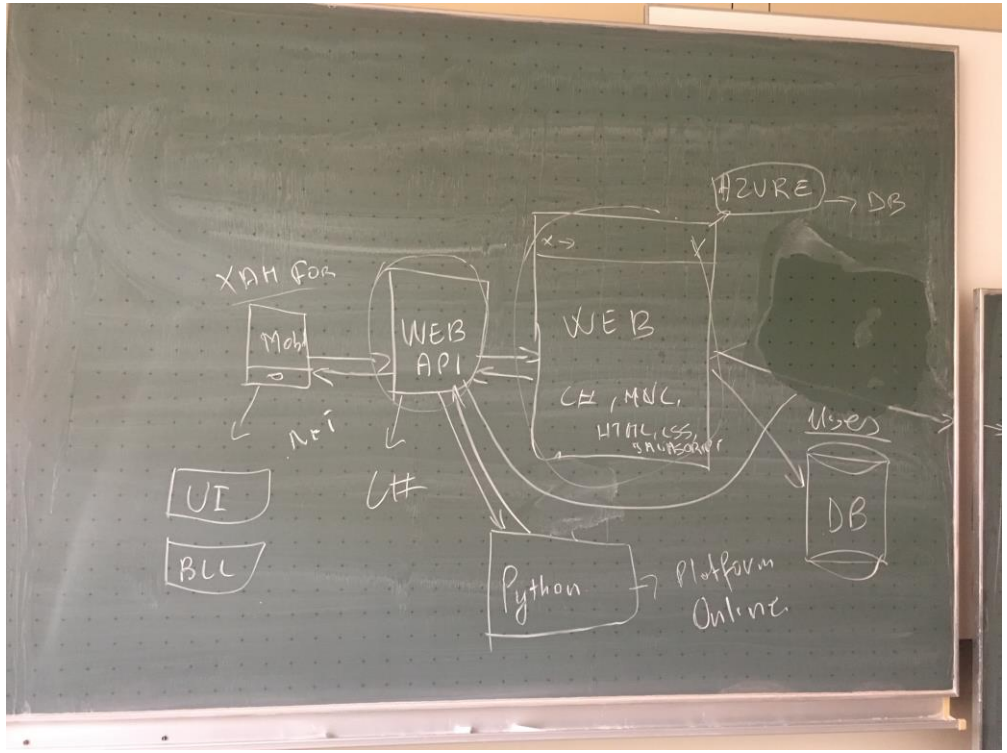
› EXERCISE

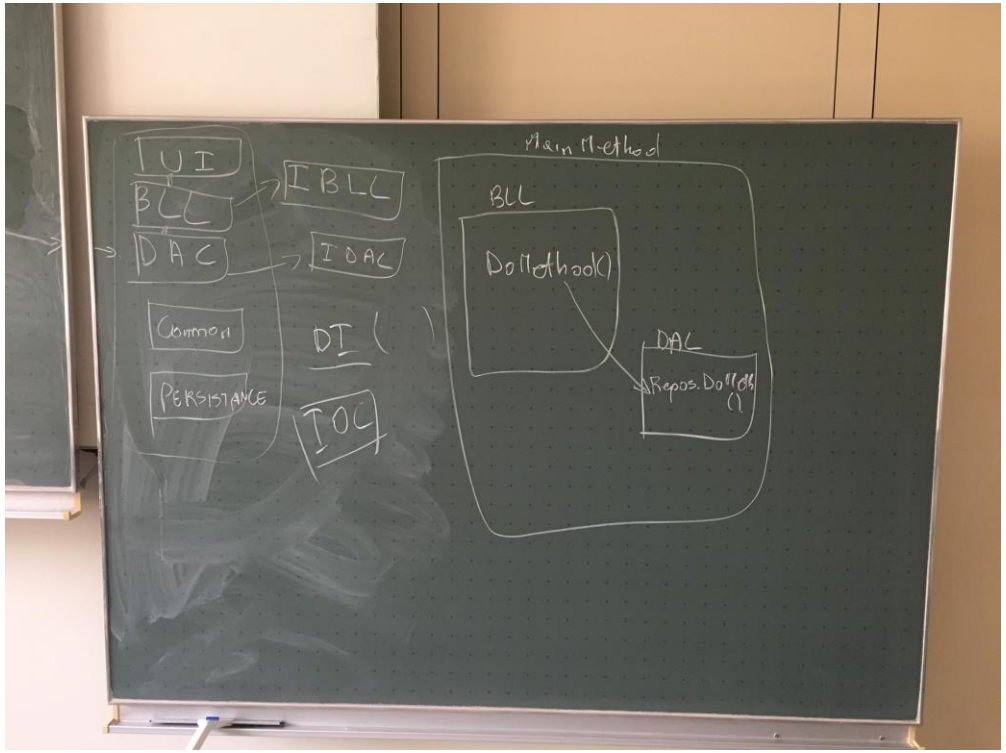
APLYING ARCHITECTURAL STYLES

HEILBRONN UNIVERSITY
OF APPLIED SCIENCES

REFERENCE ARCHITECTURE

Discussion: What architectural styles can be found here?





QUESTIONS SO FAR?