Cloud Computing

Architectures and applications in Cloud Computing
Slide set 5

Henry-Norbert Cocos
cocos@fb2.fra-uas.de

Computer Science
Department of Computer Science and Engineering
Frankfurt University of Applied Sciences

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Agenda

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Objectives of this slide set

- Learn what **Software Architectures** are!
- Learn what **Distributed Systems** are!
- Learn different architectural styles of distributed systems!
- Learn the relationship between **Software Architectures and Cloud Computing**!

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

What are Software Architectures?

Definition by ChatGPT

Software architecture refers to the high-level structure of a software system, encompassing its components and their interactions. It serves as a blueprint for both the system and the project developing it, providing a set of rules and guidelines for the system's organization and design. The primary purpose of software architecture is to ensure that a system meets its requirements, including functional and non-functional properties like performance, security, and maintainability.

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Characteristics of Software Architectures (not extensive!)

- **Modularity** – The degree to which a system's components can be separated and recombined.
- **Scalability** – The ability of a system to handle increased load by adding resources or by expanding its capacity.
- **Performance** – How well a system performs under specified conditions, often measured in terms of response time, throughput, and resource utilization.
- **Availability** – The degree to which a system is operational and accessible when required for use.
- **Interoperability** – The ability of a system to work with other systems or components, often through common interfaces or standards.
- **Portability** – The ability of a system to run on various platforms or environments without modification.
- **Reusability** – The degree to which components can be used in different systems or contexts without modification.
- **Testability** – The ease with which a system or its components can be tested to ensure they work correctly.
- **Transparency** – The extent to which the complexities of a system are hidden from the user or developer, making it easier to use or manage.
- **Resilience** – The ability of a system to recover quickly from failures and continue to operate under adverse conditions.

Pattern Oriented Software Architectures

Pattern Oriented Software Architectures

More details on the development of reusable and flexible software in the module 6.1 *Pattern Oriented Software Architecture*!

SOLID principles

SOLID is a set of five principles for designing software that is easy to maintain, extend, and refactor. These principles help in creating robust and flexible object-oriented designs. The acronym SOLID stands for:

- Single Responsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

What is a distributed system?

Definition by ChatGPT

*A distributed system is a network of **independent computers** that work together to appear as a **single coherent system** to the end user. These systems are designed to handle tasks that are too large or complex for a single machine to manage effectively. They offer numerous benefits, such as **scalability, fault tolerance, and resource sharing**, which are essential for modern computing environments like **cloud services**, big data processing, and global web applications.*

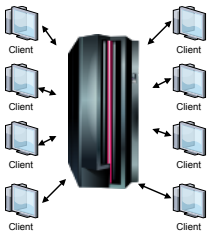
Characteristics of Distributed Systems

- **Multiple Independent Components:**
  - A distributed system consists of multiple autonomous components (nodes), such as computers, servers, or virtual machines, that communicate over a network.
  - Each node in the system operates independently, but they work together to achieve a common goal.
- **Concurrent Processing:**
  - Tasks are distributed across multiple nodes, allowing for concurrent execution and improved performance.
- **Scalability:**
  - Distributed systems can scale horizontally by adding more nodes to handle increased load and storage requirements.
- **Fault Tolerance and Reliability:**
  - The system can continue to function even if some nodes fail, thanks to redundancy and failover mechanisms.
- **Transparency:**
  - The system's complexity is hidden from users, making the system appear as a single entity.

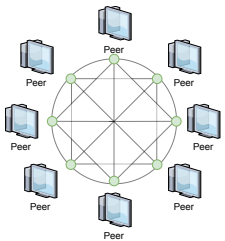
Components of Distributed Systems

- **Nodes**
  - Individual computers or servers that participate in the distributed system.
- **Network**
  - The communication medium that connects nodes, allowing them to share data and resources.
- **Middleware**
  - Software that provides common services and capabilities to applications beyond what's offered by the operating system. It facilitates communication and management of data in a distributed environment.
  - *Example:* Message brokers like Apache Kafka or RabbitMQ
- **Data Storage:**
  - Distributed systems often use distributed databases or file systems to manage data across multiple nodes.
  - *Example:* Distributed databases like Cassandra or MongoDB.
- **Coordination Services:**
  - Mechanisms that ensure nodes can coordinate tasks, manage resources, and maintain consistency.
  - *Example:* Zookeeper or Consul.

Examples of Distributed Systems



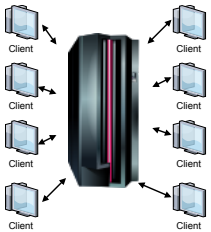
Centralized System



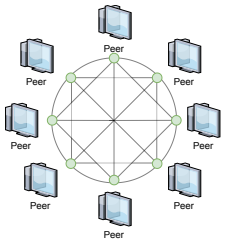
Distributed System

- **Web Applications:**
  - Large-scale web services like Google Search or Facebook, where servers handle millions of user requests concurrently.
- **Distributed Databases:**
  - Systems like Apache Cassandra or Google Bigtable, which store and manage data across multiple servers for high availability and scalability.
- **Cloud Services:**
  - Platforms like AWS or Microsoft Azure provide on-demand computing resources distributed across the globe.
- **Content Delivery Networks (CDNs):**
  - Systems like Akamai or Cloudflare that distribute content to users from servers located worldwide to reduce latency and improve load times.

Challenges in Distributed Systems (1/2)



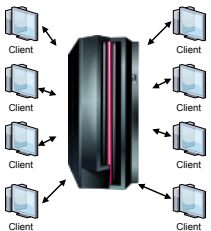
Centralized System



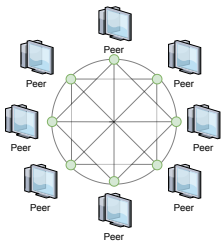
Distributed System

- **Network Latency**
  - Communication between nodes can introduce delays, affecting performance and response times.
- **Fault Tolerance**
  - Ensuring the system continues to operate despite node failures requires redundancy and complex error-handling mechanisms.
- **Consistency**
  - Maintaining a consistent state across distributed nodes, especially during updates, is challenging and often involves trade-offs (as highlighted by the CAP theorem).

Challenges in Distributed Systems (2/2)



Centralized System



Distributed System

- **Synchronization**
  - Coordinating actions and data among nodes to ensure they work in harmony is complex, especially in the presence of network partitions.
- **Scalability**
  - Managing the growth of the system without performance degradation requires careful design and resource management.

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Fallacies of Distributed Systems

1. **The network is reliable!**
2. **Latency is zero!**
3. **Bandwidth is infinite!**
4. **The network is secure!**
5. **Topology doesn't change!**
6. **There is one administrator!**
7. **Transport cost is zero!**
8. **The network is homogeneous!**

- The fallacies describe usual problems in the implementation of distributed systems.

History

The *Fallacies of Distributed Systems* were originally formulated by L. Peter Deutsch from Sun Microsystems in 1994. Bill Joy and Tom Lyon had already identified the first four as **The Fallacies of Networked Computing**.

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Fallacy 1: The network is reliable

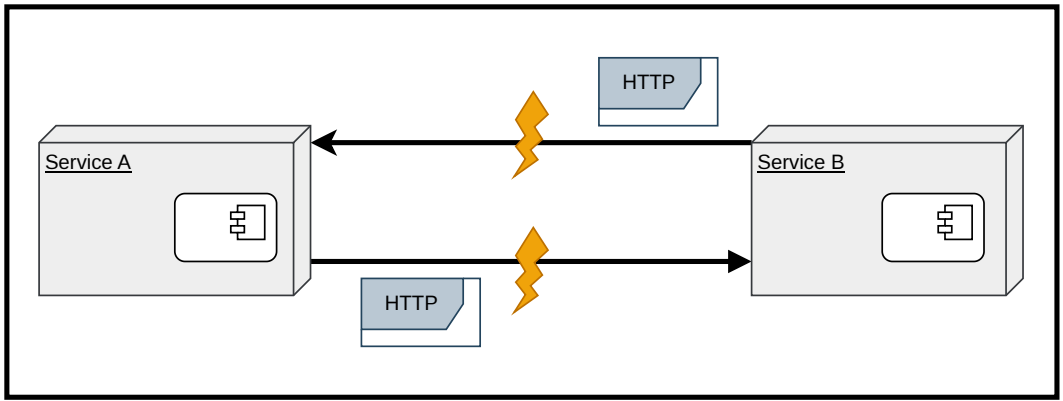


Figure: Fallacy 1: The network is reliable

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Fallacy 2: Latency is zero & Fallacy 4: The network is secure

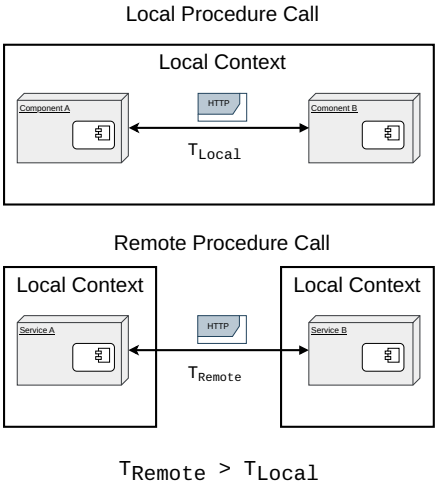


Figure: Fallacy 2: Latency is zero

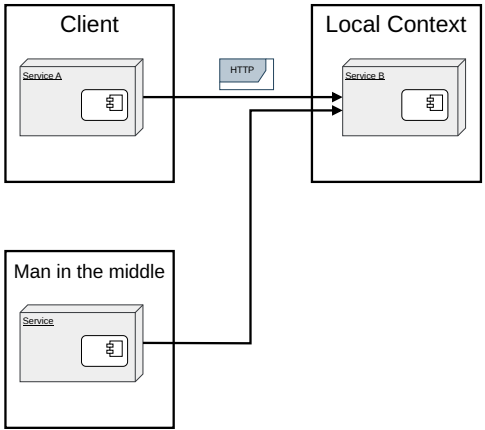


Figure: Fallacy 4: The network is secure

Fallacy 8: The network is homogeneous

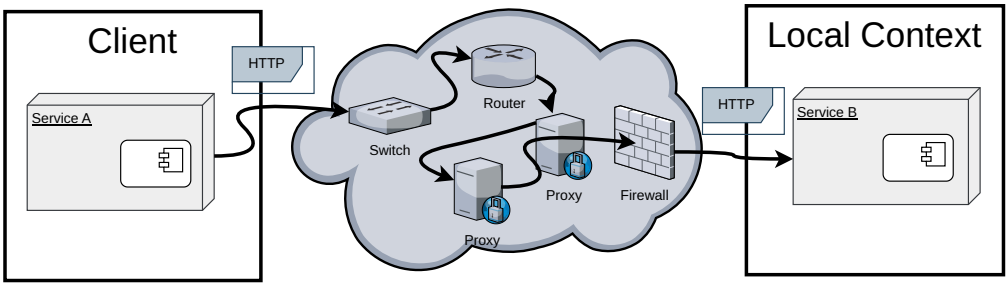


Figure: Fallacy 8: The network is homogeneous.

Communication in distributed systems

Source: Distributed Systems, M. van Steen and A.S. Tanenbaum

- Communication between processes is foundation of distributed systems.
- Important for the communication between independent systems is **Inter-Process Communication**.
- There are many different styles (e.g. Remote Procedure Calls, Message-Oriented Middlewares, etc) and architectures for communication (Client-Server, 3-tier Architecture, etc.).

Scope of this lecture

We will only discuss the most prominent architectures in this lecture! The rest is **not scope** of this lecture! If you want to know more details look into the book *Distributed Systems, M. van Steen and A.S. Tanenbaum*.

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Application Program Interface

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

API – Definition by Wikipedia

An application programming interface (API) is a way for **two or more computer programs or components to communicate with each other**. It is a type of software interface, **offering a service** to other pieces of software. A document or standard that describes how to build or use such a connection or interface is called an API specification. A computer system that meets this standard is said to implement or expose an API. The term API may refer either to the specification or to the implementation. Whereas a system's user interface dictates how its end-users interact with the system in question, its API dictates how to write code that takes advantage of that system's capabilities.

APIs are a central component in the development of software especially in the realm of distributed systems.

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

APIs in the context of Cloud Computing

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

For the implementation of web services there are two styles:

- **SOAP (Simple Object Access Protocol)**
- **REST (REpresentational State Transfer)**

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Web Services

Web services are services which are offered by a **server to clients** which access the service via a **standardized method** and use a **standardized format (HTML, XML, JSON)**. Web services use protocols like HyperText Transfer Protocol (HTTP) for the exchange of data. The standardized way of integrating Web-based applications using the **XML, SOAP, WSDL and UDDI** open standards over an Internet Protocol backbone.

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

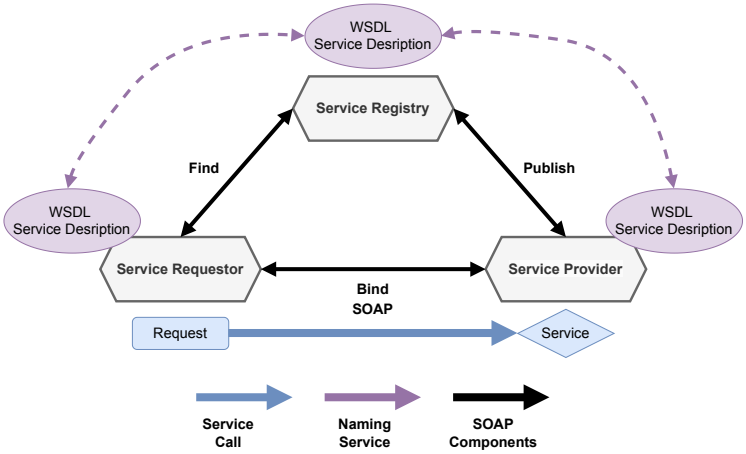
SOAP Web Services – Architecture

Actors in SOAP services:

- **Service Requestor**
- **Service Provider**
- **Service Registry**

Components of SOAP services:

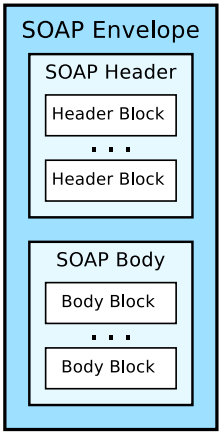
- **Web Service Description Language (WSDL)**
- **SOAP Messages**



OF APPLIED SCIENCES

SOAP – Message Format and WSDL

- **SOAP Envelope** – The element of an XML message identifying it as a SOAP message (example next slide).
- **SOAP Header** – A collection of one or more header blocks targeted at each SOAP receiver.
- **SOAP Header Block** – One ore more discrete computational blocks within the SOAP header.
- **SOAP Body** – The body of the message. The interpretation and processing of SOAP body is defined by header blocks.



FRANKFURT UNIVERSITY OF APPLIED SCIENCES

SOAP – Message

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

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 299
SOAPAction: "http://www.w3.org/2003/05/soap-envelope"

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope" xmlns
    :m="http://www.example.org">
  <soap:Header>
  </soap:Header>
  <soap:Body>
    <m:GetStockPrice>
      <m:StockName>T</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
```

REpresentational State Transfer

- Invented by Roy Fielding in the dissertation *Architectural Styles and the Design of Network-based Software Architectures* at UC Irvine.
- **Aim:** construction of a universally usable API over the WWW!
- **Result:** request-response-based architectural style making use of **HTTP-verbs**

Principles of REST:

- **Client-Server Architecture** – The server (provider) provides a service that can be requested by the client (requestor) if required.
- **Caching** – The requests should be cacheable.
- **Statelessness** – Each REST message contains all the information to understand the message. Neither the server nor the application should store status information between two messages.
- **Multi-Layered Systems** – As a result it is sufficient to offer the user just one interface. The underlying layers can remain hidden, thus simplifying the architecture as a whole.
- **Unified API** – REST-based services should offer a standardized interface based on self-describing messages, addressable resources, representations of resources and the principle of **Hypermedia as the Engine of Application State (HATEOAS)** principle.

CRUD Operations

Source: https://en.wikipedia.org/wiki/Create,_read,_update_and_delete

"In computer programming, create, read, update, and delete (CRUD) are the four basic operations of persistent storage.CRUD is also sometimes used to describe user interface conventions that facilitate viewing, searching, and changing information using computer-based forms and reports."

`https://somedomain/user/123?format=json`

- a `GET` request to `/user/` returns a list of registered users on a system
- a `POST` request to `/user/` creates a user with the **ID 123** using the body data.
- a `DELETE` request to `/user/123` deletes user **123**

Operation	SQL	REST
Create	INSERT	POST, PUT
Read	SELECT	GET
Update	UPDATE	PUT
Delete	DELETE	DELETE

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Consistency models in data-centric applications

Consistency models

In the context of databases, **ACID** and **BASE** are principles that describe different approaches to managing transactions and consistency. They represent two different philosophies for ensuring data integrity and availability in database systems, particularly in distributed environments.

ACID Principles

ACID (Atomicity, Consistency, Isolation, and Durability). The principles are associated with relational databases and ensure that transactions are processed reliably.

- **Atomicity**
  - Ensures that all parts of a transaction are treated as a single unit, which either completes entirely or not at all. There are no partial transactions.
  - Prevents data corruption in the case of a failure during a transaction.
  - *Example:* In a banking system, a transaction that transfers money from one account to another should debit one account and credit another as a single atomic action. If one part fails, the whole transaction is rolled back.
- **Consistency**
  - Ensures that a transaction brings the database from one valid state to another, maintaining data integrity. All rules (constraints, triggers, etc.) must be satisfied.
  - Guarantees that database constraints are not violated, preserving the correctness of the data.
  - *Example:* In an e-commerce system, an order cannot be placed if the inventory does not have enough items in stock. Consistency ensures that the stock levels are correctly updated.

ACID Principles

- **Isolation**
  - Ensures that transactions are executed in isolation from each other, meaning the intermediate states of a transaction are not visible to other transactions until the transaction is complete.
  - Prevents concurrent transactions from interfering with each other, ensuring predictable results.
  - *Example:* In a ticket booking system, two users booking the last available ticket simultaneously should not both succeed. Isolation ensures that one transaction completes before the other starts.
- **Durability**
  - Ensures that once a transaction has been committed, it remains so, even in the case of a system failure.
  - Guarantees that committed transactions are permanently recorded.
  - *Example:* Once a purchase is confirmed in an online store, it remains confirmed even if there is a subsequent system crash.

BASE Principles

BASE (Basically Available, Soft state, Eventual consistency). The principles are associated with NoSQL databases and are designed for systems that prioritize availability and partition tolerance over strict consistency, which is often necessary in distributed systems.

- **Basically Available** – Rather than enforcing immediate consistency, BASE-modelled NoSQL databases will ensure availability of data by spreading and replicating it across the nodes of the database cluster.
- **Soft State** – Due to the lack of immediate consistency, data values may change over time. The BASE model breaks off with the concept of a database which enforces its own consistency, delegating that responsibility to developers.
- **Eventually Consistent** – The fact that BASE does not enforce immediate consistency does not mean that it never achieves it. However, until it does, data reads are still possible (even though they might not reflect the reality).

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

ACID vs. BASE

- **Consistency vs. Availability**
  - ACID prioritizes strict consistency, ensuring that transactions are reliable and isolated.
  - BASE relaxes consistency for higher availability and resilience in distributed environments.
- **Transaction Complexity**
  - ACID transactions are more complex and are suited for applications requiring strong transactional integrity, like financial systems.
  - BASE is simpler and more scalable, fitting applications that can tolerate some level of inconsistency, like social networks or content delivery systems.
- **Use Cases**
  - **ACID:** Banking systems, order processing, inventory management where precise transaction control is critical.
  - **BASE:** Distributed systems like cloud services, big data applications, and real-time analytics where responsiveness and availability are more critical than immediate consistency.

Summary ACID vs. BASE

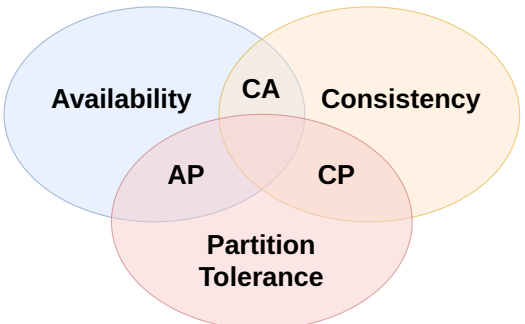
ACID vs. BASE

- ACID focuses on reliable and consistent transactions, making it ideal for systems where data integrity is paramount.
- BASE offers flexibility, availability, and scalability at the cost of relaxed consistency, suitable for distributed systems where high availability and partition tolerance are crucial.

CAP theorem

The ACID and BASE principles stem form the CAP theorem. The CAP Theorem is a fundamental principle in the design of distributed systems. Formulated by computer scientist Eric Brewer, it states that a distributed system can only guarantee at most two out of three fundamental properties of distributed data-centric applications!

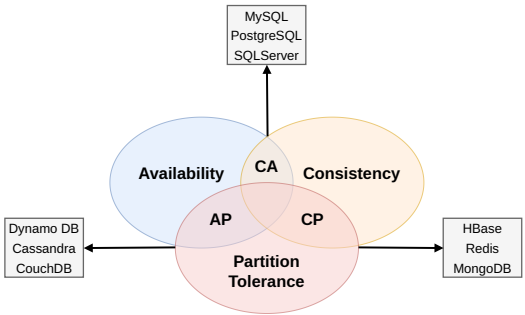
CAP theorem (1/2)



- **Consistency** – Every read receives the most recent write or an error.
- **Availability** – Every request receives a (non-error) response, without the guarantee that it contains the most recent write.
- **Partition tolerance** – The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes.

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

CAP theorem (2/2)

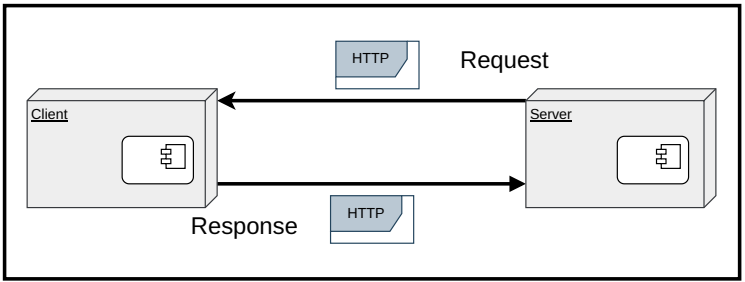


- **CA** – The system is always consistent and available as long as there is no network partition
- **CP** – The system is consistent and can handle partitions, but might sacrifice availability. If a partition occurs, some nodes might not respond until the partition is resolved.
- **AP** – The system is always available and can handle partitions, but it might serve outdated or inconsistent data during partitions.

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Client/Server Architecture

Source: Distributed Systems, M. van Steen and A.S. Tanenbaum



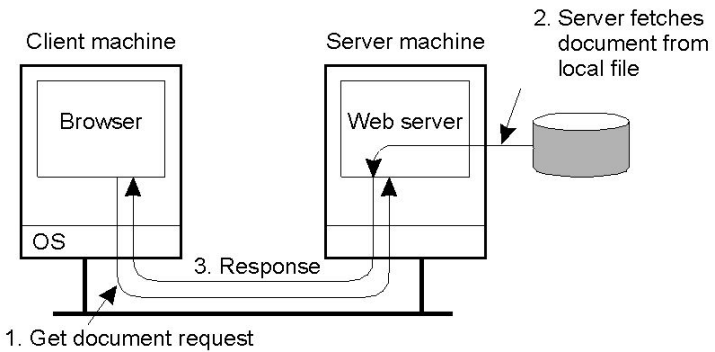
Definition by van Steen

*"In the **basic client-server model**, processes in a distributed system are divided into two (possibly overlapping) groups. A **server** is a process implementing a specific service, for example, a file system service or a database service. A **client** is a process that requests a service from a server by sending it a request and subsequently waiting for the server's reply."*

Client/Server – The World Wide Web

Source: Distributed Systems, M. van Steen and A.S. Tanenbaum

- The core of a Web site is formed by a process that has access to a local file system storing documents!
- A URL specifies the application-level protocol for transferring the document across the network.
- The communication between a browser and Web server is standardized: they both adhere to HTTP!



FRANKFURT UNIVERSITY OF APPLIED SCIENCES

N-Tier Architecture

Source: Distributed Systems, M. van Steen and A.S. Tanenbaum

- The Web started out as the relatively simple two-tiered client-server system...
- ...however more layers can be added!

Definition Source: Distributed Systems, M. van Steen and A.S. Tanenbaum

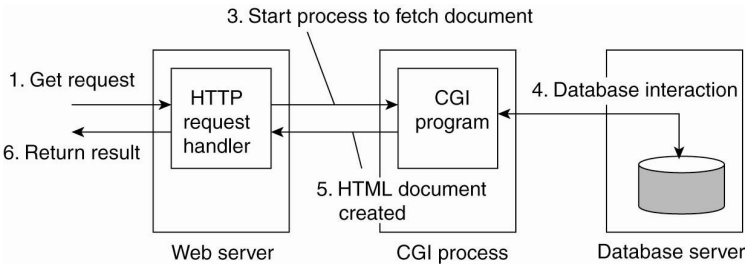
"In software engineering, multitier architecture (often referred to as n-tier architecture) is a client–server architecture in which presentation, application processing and data management functions are physically separated. The most widespread use of multitier architecture is the three-tier architecture[...]."

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Example:Common Gateway Interface

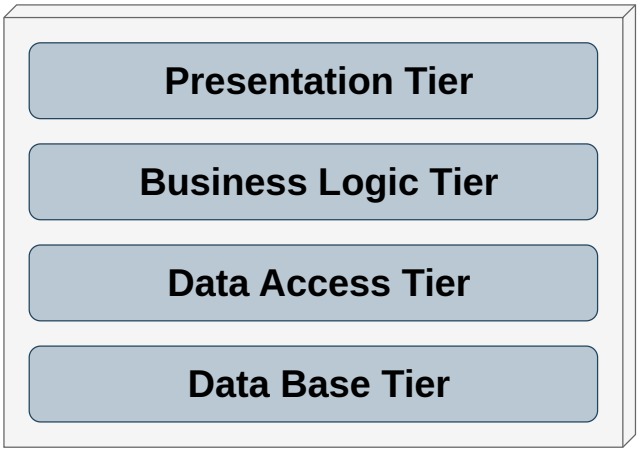
Source: Distributed Systems, M. van Steen and A.S. Tanenbaum

- CGI defines a standard way by which a Web server can execute a program taking user data as input.
- Usually user data come from an HTML form; it specifies the program that is to be executed at the server side.
- The server sees the request, it starts the program named in the request and passes it the parameter values.
- In other words, the server does nothing but delegate the fetching of a document to an external program.



FRANKFURT UNIVERSITY OF APPLIED SCIENCES

N-Tier Architecture



N-Tier Architecture

- **Presentation Tier** – User interface of the application.
- **Business Logic Tier** – Application logic of the system.
- **Data Access Tier** – Logic for the access of the Data Base (optional).
- **Data Base Tier** – Data Base of the system.

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

N-Tier Architecture in Software development

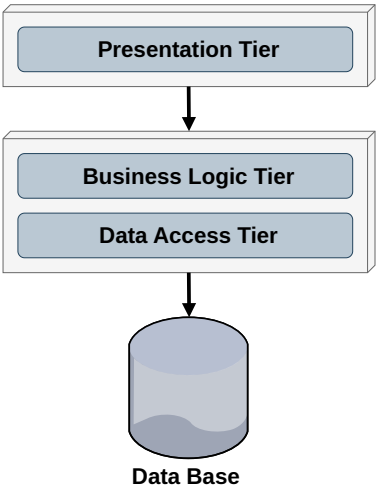
- The classic setup is the **3-Tier-Architecture**.
- It is common that the data access tier is considered a sublayer of the business logic tier.
- Typically it encapsulates the API definition for the data access.

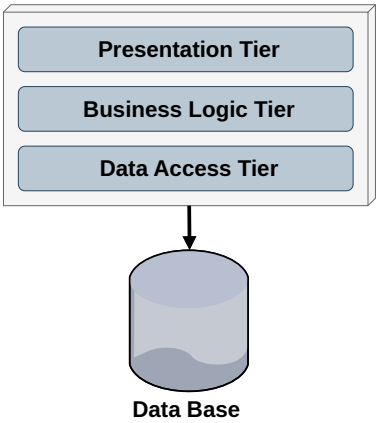
Web development usage

This architecture is used by many frameworks for the development of web applications (e.g, Java Platform, Enterprise Edition (Java EE), now Jakarta EE).

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

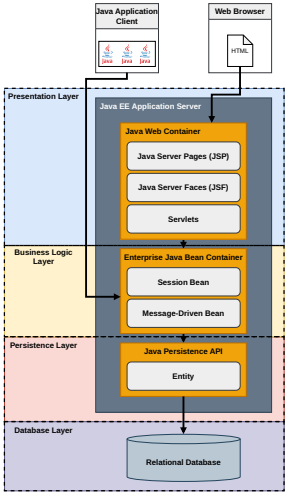
N-Tier Architecture deployment





FRANKFURT UNIVERSITY OF APPLIED SCIENCES

N-Tier-Architecture Example: Jakarta EE



Jakarta EE

- It provides a productive (and complicated) infrastructure of services offering many different types of components.
- Each component is deployable separately, with many ways for the various parts to communicate with each other.
- Those services are delivered through the use of software packages called **application servers**.

Source: M. Sciabarrà, Learning Apache Openwhisk: Developing Open Serverless Solutions

"At the time Java was considered to be a programming language suitable for building large, scalable applications (meant to replace C++). Scripting languages like Python were not yet largely used, and JavaScript was still in its infancy."

N-Tier Architecture – Web development usage

For the development of web services 3-Tier-Architectures were used to implement large scale web applications (e.g. e-commerce systems):

- A front-end web server serving static content, maybe some cached dynamic content.
- A application server running the application code (e.g., Jakarta EE).
- A back-end database or data store.

Drawbacks

- Infrastructure does not scale very good!
- The application server software is complex and proprietary!
- The application code needs to be packaged in application server $\Rightarrow$ Large Binary Package!

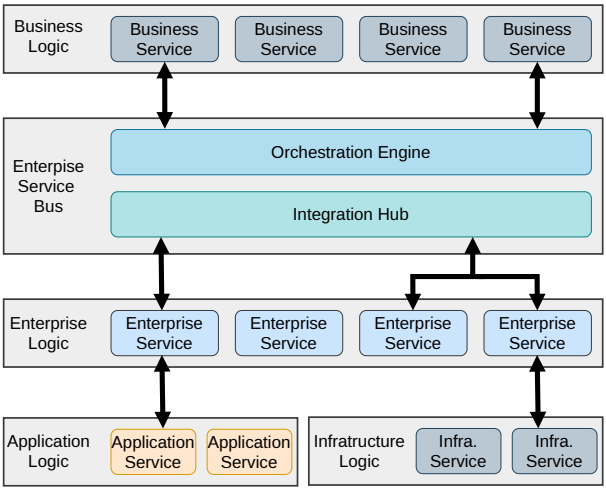
Service-Oriented Architecture

- Service-Oriented Architecture focus on discrete services instead of a monolithic design.
- A service is a discrete unit of functionality that can be accessed remotely and acted upon and updated independently.
- **Aim:** Independence of vendors and technologies, Functional decomposition of (business) service.

According to the Open Group a service has four properties:

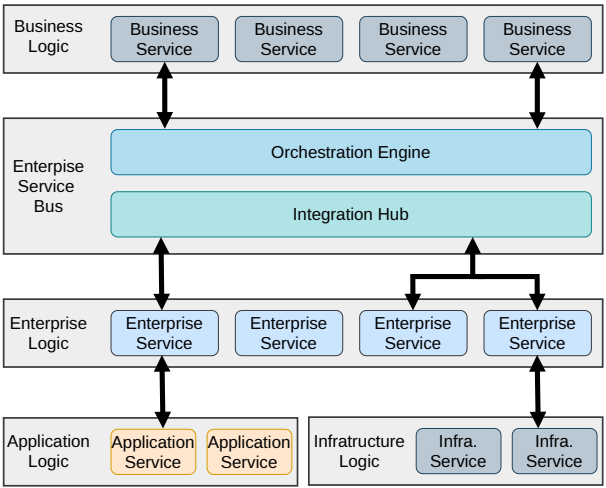
- It logically represents a repeatable business activity with a specified outcome.
- It is self-contained.
- It is a black box for its consumers, meaning the consumer does not have to be aware of the service's inner workings.
- It may be composed of other services.

Service-Oriented Architecture (1/3)



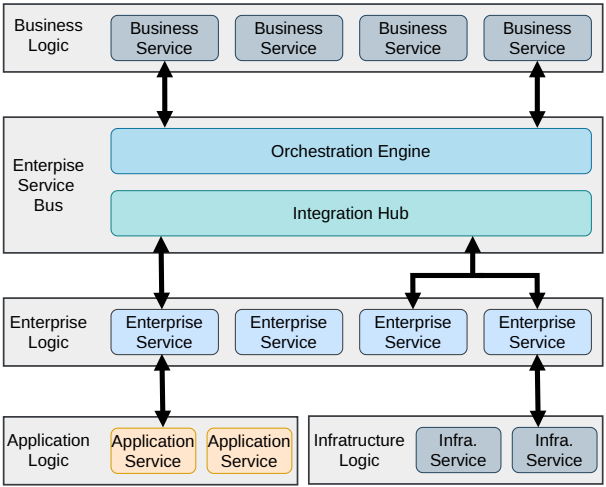
- **Business Logic**
  - Consists of domain specific, atomic **business services** defined by the business experts.
  - No code definition here!
- **Enterprise Logic**
  - Fine granular, shared **enterprise services** defined by the sw developers.
  - Building blocks for the business services connected by an **Orchestration Engine**.

Service-Oriented Architecture (2/3)



- **Application Logic**
  - The implemented **application services** are not as fine grained as business and enterprise services.
  - Once implemented and not intended for reuse by other services.
  - The implemented **infrastructure services** are implemented for technical solutions.
  - Implementations are usually monitoring, logging and authentication.

Service-Oriented Architecture (3/3)



- **Orchestration Engine:**
  - Connector of business services to enterprise logic.
  - Orchestrates the use of services and usage (e.g. transaction management or conversions).
  - Implements a database for the coordination of services.
- **Integration Hub:**
  - Connector between the different applications and services of the enterprise.
  - Integrates the code parts of the architecture.

Service-Oriented Architecture

Drawbacks

- The Orchestration and Integration Hub pose many limitations on the architecture:
  - One single point for coupling of services.
  - Only one (or few) databases for service orchestration ⇒ **Bottleneck!**
  - All service share one single architectural pattern ⇒ **all need to oblige to the ESB!**
- The distributed nature of the services led to the following problems:
  - Requests (and transactions) were distributed over the whole architecture!
  - Performance of the architecture was rather bad!

This unites the drawbacks of monolithic architectures!

Therefore Microservice Architectures were developed!

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Microservice Architecture

Source: Martin Fowler, https://martinfowler.com/articles/microservices.html

Definition by Martin Fowler

"The microservice architectural style is an approach to developing ***a single application*** *as* ***a suite of small services***, ***each running in its own process*** *and* ***communicating with lightweight mechanisms***, *often an* ***HTTP resource API***. *These services are built around business capabilities and* ***independently deployable*** *by* ***fully automated deployment*** *machinery. There is a bare minimum of centralized management of these services, which* ***may be written in different programming languages*** *and* ***use different data storage*** *technologies.*

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Principles for developing Microservices

Source: Cloud-Native Computing, Kratzke

1. Create models around business concepts
2. Create a culture of automation
3. Hide internal implementation details
4. Decentralize
5. Define independently updatable units
6. Isolate errors
7. Build easily monitored services

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Properties of a Microservice architecture

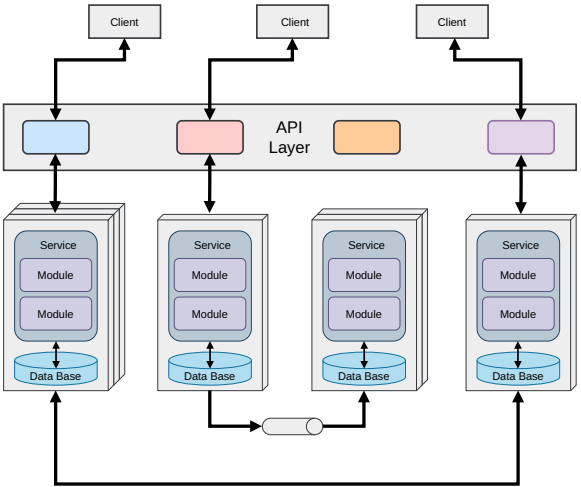
- **Distributed** – Each service runs in its own process.
- **Bounded Context** – Each service is its own domain.
- **Fine Granularity** – Each service is responsible for its own domain.
- **Isolation** – Data is distributed over the architecture! No *Single Point of Truth* (SPOT).

Single Point of Truth (SPOT)

A *Single Point of Truth* in software architecture is a place where different entities (classes, services, etc.) have a single point of data retrieval and therefore the integration of the architecture is done in the data storage (usually a relational database).

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Microservice Architecture



Microservice Architecture

- Each service can be replicated.
- Services communicate with each other via defined APIs (e.g. REST/HTTP).
- A high degree of decoupling but less reuse!
- Each service ships its own context (classes, entities) and databases. No sharing of code or data storage!

API-Layer

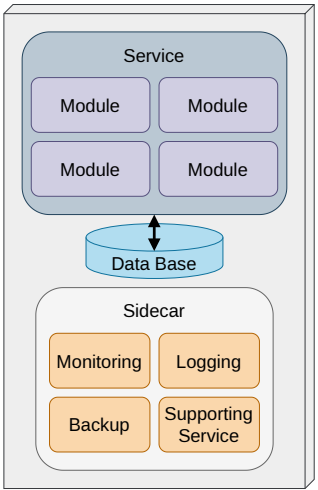
An optional layer for the connection of consumer (e.g. users) and the system. Here functionality like service discovery is implemented.

Operational Reuse

- Microservice architecture favor duplication over coupling.
- However sometimes coupling is useful if elements could be reused (especially technical infrastructure!).
- The common use cases here is functionality like monitoring and logging.
- This type of schema is called **operational reuse** and refers to the **sidecar** pattern.

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Microservice Sidecar



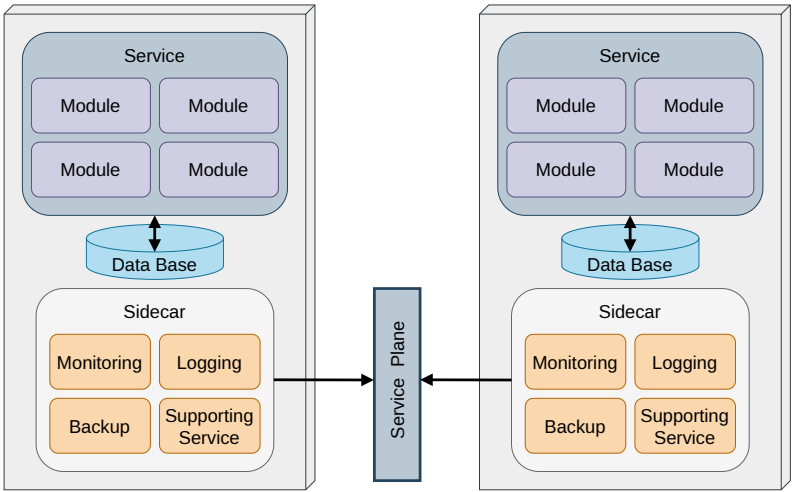
Sidecar

- The sidecar implements the common services reused though out the architecture.
- The sidecar is implemented as a separate component in each microservice.
- The sidecar can be exchanged and updated separately form the other components of the service.
- The sidecar components can exchange messages with each other over a service plane.

Service plane

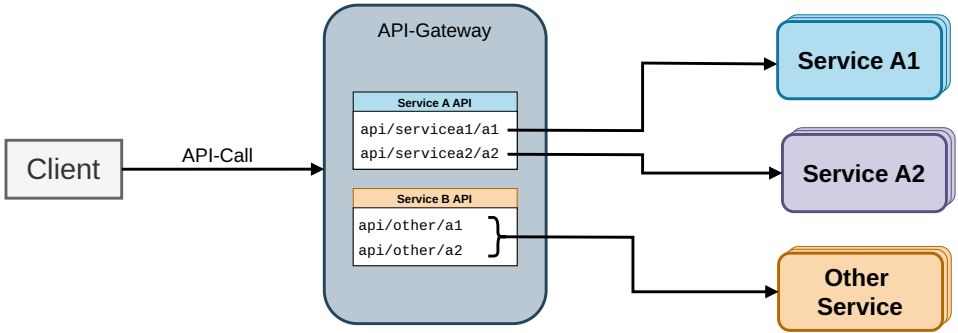
The service plane can be integrated by a service mesh, where all supporting services for the microservices are managed in central manner (see slide 58).

Microservice Sidecar communication



FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Microservice – API-Gateway



API-Gateway

The API-Gateway forms a central request-response-based access point to the backend services. The more such backend services are created, the more likely it is that frontend services will not have to interact with dozens or hundreds of individual services, but will instead have a central access point.

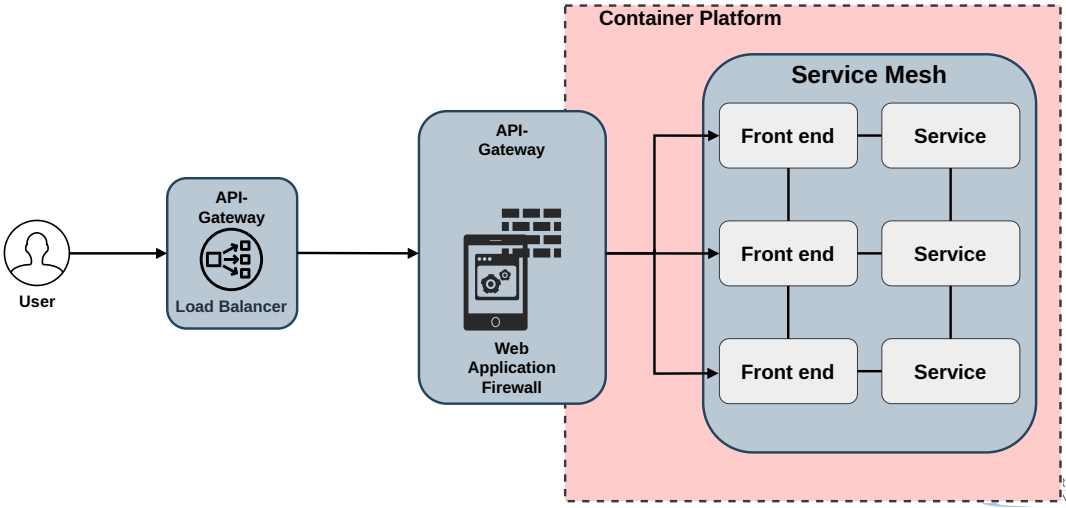
Microservice – API-Gateway

By introducing an API gateway that forms a boundary to the backend systems (and is therefore sometimes called an "edge service") and serves as a central access point, various orchestration disadvantages in microservice architectures (but also serverless architectures) can be compensated for. The disadvantages of service architectures include many communication connections for clients. Overall, the complexity of a complete system increases with the number of APIs provided.

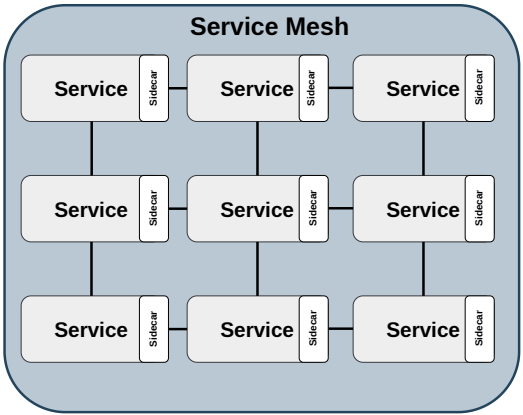
The API-Gateway has the following requirements:

- **Cross-functional solutions** should not have to be implemented individually for each backend service, but should be implemented uniformly across all services.
- The aim is to monitor the use of APIs using **analysis and monitoring tools** in order to determine the purposes for which the APIs provided are used.
- If APIs are to be monetized, a connection to a billing system must be established.
- In **service-of-services architectures**, dozens of upstream services may need to be queried to respond to individual requests. New API services will be added or removed over time. Nevertheless, all these changes should not filter through to the client, which should always find all services in their usual place.

Microservice – API-Gateway Location



Microservice – Service Mesh



Service Mesh

- The service mesh is a console for developers to gain access to the services.
- Each service is a node in the service mesh.
- The operational coupling (logging and monitoring) can be controlled globally.
- **Service discovery** is an important part of microservice architectures.

Service Discovery

All requests are sent to the service discovery tool (e.g. Consul). The tool monitors the requests and starts new instances and services on request making this architecture scalable and each service elastic.

Summary on Microservice Architecture

Benefits

- **Protocol-awareness** – Each service needs to know how to call the other services ⇒ **No Interation Hub!**
  - The communication between services is standardized REST/HTTP, gRPC, Message-Queues. . . ⇒ **Service Discovery!**
- **Heterogeneity** – Each service can implement a different technology stack (polyglot).
  - The used programming languages, frameworks, storage technologies can differ in every service.
- **Interoperability** – Each service is communicating with other services over the network ⇒ **No proprietary communication!**

Drawback

- Complexity!!!

Serverless Architecture

In addition to microservice architectures, so-called serverless architectures are increasingly so-called **serverless architectures**. This is an architectural approach that ultimately takes up the **microservice architecture** approach and takes into account special features that resulting from the **FaaS programming model**. FaaS platforms are merely **event processing systems** (see slide set 3). **Events** can be sent **via HTTP**, for example, or received from other event event sources (from cloud infrastructures). The platform then determines which **functions** are registered for an event, **sends the event to the function instance** and **waits for a response**.

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Serverless Architecture

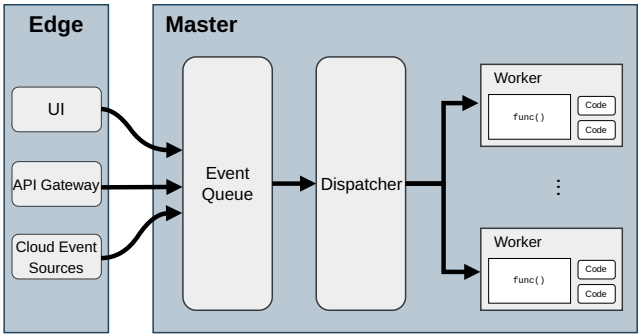


Figure: Serverless Architecture

Serverless Architecture

- **Edge** – API-Gateway or events triggering a function call.
- **Event Queue** – Ordering and managing the requests from the sources.
- **Dispatcher** – Managing the spawn of workers in the platform.
- **Workers** – Functions executing the code and returning the results.

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Properties of Serverless Architectures

Serverless architectures have the following properties:

- Serverless computing is designed to be stateless ⇒ **No states are stored in functions between calls!**
- If functions do not have to process event , no resources (processor, main memory) are allocated to them. ⇒ **Therefore the term serverless!**
- Pricing is based on the pay-as-you-go principle based on the amount of resources actually consumed (processor, main memory), required to process events.

Summary

Serverless computing therefore refers to a cloud computing model in which cloud providers allocate resources on an event-driven basis. Developers of serverless applications therefore do not have to worry about capacity planning, configuration, management, maintenance, operation or scaling of containers, VMs or physical servers (hence the name serverless!).

Summary

Brief summary over the lecture and focal takeaway points!!!

Outlook

~~1st part: Introduction~~

~~2nd part: Technological foundations~~

~~3rd part: Service models, deployment models~~

~~4th part: Adoption and strategy~~

5th part: Architectures and applications $\Longleftarrow$ *This slide set*

6th part: Cloud-Native applications

7th part: Current and future trends

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

6th part: Cloud-Native applications

Topics:

- Cloud-Native Applications
- Components of Cloud-Native Computing
- Architectures and Patterns in Cloud-Native Computing
- Benefits and Challenges in Cloud-Native Computing
- Summary

FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Thank You For Your Attention!

Henry-Norbert Cocos, M.Sc
Frankfurt University of Applied Sciences
Room 1-230
☎ +49 69 1533-2699
✉ cocos@fb2.fra-uas.de
www.henrycocos.de