Contents

1 Architecture Significance

Software Architecture Software Architecture is the organization of the system components with their relationships. Design principles should help to make decisions how to create components and the relationships.

> The fundamental organization of a system is embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution. – [ISO/IEC/IEEE 42010, November 2011]

User Story User Stories describe a feature which should be implemented. User Stories can often translate from Functional Requirements.

- As a (**who** wants to accomplish something)
- I want to (**what** they want to accomplish)
- So that (**why** they want to accomplish that thing)

An example:

- As a bank customer
- I want to withdraw money from an ATM
- So that I'm not constrained by opening hours or lines at the teller's

Architectural Significance of Requirements ASR are requirements (Functional Requirements or Non-Functional Requirements) which have a measurable impact on the systems' architecture.

ASR Test With the ASR Test you can test / determine the architectural significance of requirements. **Important**: They are not MECE.

1. The requirement is directly associated with high **business value** or **business risk**.
2. The requirement is a **concern** of a particularly important **stakeholder** (for instance, the project sponsor or an external compliance auditor).
3. The requirement has runtime **Quality-of-Service (QoS)** characteristics (e.g., performance needs) that deviate from those already satisfied by the evolving architecture substantially.
4. The requirement causes new or deals with one or more existing **external dependencies** that have unpredictable, unreliable and/or uncontrollable behavior.
5. The requirement has a **cross-cutting nature** and therefore affects multiple parts of the system and their interactions; it may even have system-wide impact.
6. The requirement has a **first-of-a-kind** character: e.g., the team has never built a component that satisfies this particular requirement.
7. The requirement has been **troublesome** and caused critical situations, budget overruns or client dissatisfaction **in a previous project** in a similar context.

Figure 1: Example for an ASR template

SMART **SMART** are the letters

Letter	Project Mgmt	Requirements Engineering
S	Specific	Specific
M	Measurable	Measurable
A	Achievable	Agreed Upon
R	Relevant	Realistic
T	Time-Bound	Time-Bound

Table 1: SMART Abbrevation Table

S Which feature of part of the system should satisfy the requirement?
M How can you find out whether the requirement is met (or not), is it quantified?

2 Requirements

FURPS **FURPS** is an acronym representing a model for classifying software quality attributes (Functional Requirements and Non-Functional Requirements).
FURPS:

F Functionality
U Usability

R Reliability
P Performance
S Supportability

Plus (+)

- Design constraints
- Implementation constraints
- Physical constraints
- Interface constraints

Quality Attribute Scenario Quality Attribute Scenario (QAS) are Non-Functional Requirements which are very important (high impact). For such requirements you could fill out the template as show in Figure 2.

- Stimulus:
  - What starts the whole thing?
- Stimulus Source:
  - Who starts the whole thing?
- Environment:
  - In which situation (normal operation, stress operation, …)
- Artifact:
  - Everything what specified is
- Response:
  - An observable behavior
- Response Measure
  - Set measurable number as values

Scenario for Order Management SOA at "T" (Business-to-Business), NFR #5		
Scenario(s)		Response Time in Web Channel: Virtual Service Provider (VSP) API
Business Goals		Drive down cost of operations by interacting with VSPs efficiently
Relevant Quality Attributes		Performance (response time), scalability
Scenario Components	Stimulus	Order placed by VSP (system or end user)
	Stimulus Source	External to system (see functional requirements and architecture overview diagram: "Create phone service", "Move phone service")
	Environment	At runtime, normal operational conditions: system up and running, not stressed (by unexpected workload, by an external attack, …)
	Artifact	All components and connectors on different logical layers and physical tiers required to process order (down to customer database in backend)
	Response	Orders are accepted immediately, order number (identifier) is returned
	Response Measure	Order acknowledgment is sent in 3 seconds (or less). This performance is desired but not guaranteed to VSPs in any Service Level Agreement (SLA),
Questions		Can errors be identified and handled properly in this timeframe?
Issues		Backend systems might not have sufficient processing power

Figure 2: QAS Template

Agile Landing Zones Agile Landing Zones can be used to quantify NFR. Instead of estimating a single value you estimate three values (M1, M2, M3). Instead of aiming for a single value you have three values.

M1 Minimal Goal (you **HAVE** to meet this requirement)
M2 Target (you plan for this goal)
M3 Outstanding (you design for this goal)

C4 Model The C4 Model consists of 4 different diagrams:

- Context
- Container
- Component
- Classes

Each diagram zooms more and more into the class diagram.

System Context Diagram A System Context Diagram (SCD) is used to define the boundary between systems, as part of the whole system and its relationship. The Context Diagram from the C4 Model (What is the C4 Model) is such a SCD.

3 Solution Strategy

Solution Strategy

> Summary of the fundamental decisions and solution strategies that shape the architecture. Can include technology, top-level decomposition, approaches to achieve top quality goals and relevant organizational decisions.

In Solution Strategy you will make big decisions as:

- client / server architecture
- our application communicates over JSON
  - JSON schema is not required to be defined at this stage

Solution Strategy will be performed during Inception & Elaboration from RUP.

Architectural Decisions Architectural Decisions are used to capture **key design** issues and the rationale behind chosen solutions. These decisions can be documented with e.g. Y Statements

Y Statements The Y-Statement can be used to document Architectural Decisions. **Facing** and **to achieve** look very similar. **Facing** talks about what is required / needed (by the client). **To achieve** talks about what I want to solve with the solution.

- In the context of the order management scenario at T,
- facing the need to process customer orders synchronously, without loosing any messages,
- we decided to apply the Messaging pattern and the RPC pattern
  - and neglected File Transfer, Shared Database, no physical distribution (local calls)
- to achieve guaranteed delivery and request buffering when dealing with unreliable data sources
- accepting that follow-on detailed design work has be performed and that we need to select, install, and configure a message-oriented middleware provider.

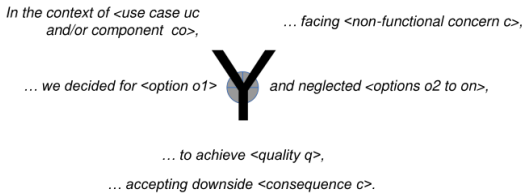

Figure 3: Y-Statement Template

4 Container Architecture

Beans Beans are self written components (see C4 Model).

Inversion of Control (IoC) Inversion of Control is a design pattern which is often used by frameworks. I write a component, and then I tell the framework how and when it should be called.
The Framework controls the workflow and calls your components when appropriate. This is also called /Hollywood-Principle:

> Don't call us, we call you!

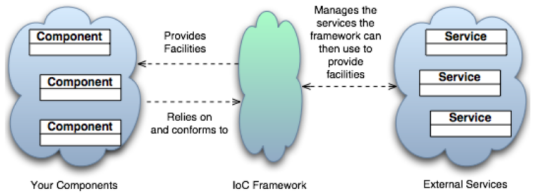

Figure 4: Example for IoC

5 Story Splitting

INVEST If a User Story is too large for a sprint you have to brake it down. The INVEST properties should help you:

I Independent
N Negotiable
V Valuable
E Estimable
S Small
T Testable

Story Splitting Sometimes a User Story does not satisfy the INVEST criteria. In these cases you should perform Story Splitting. Story Splitting is the process of splitting a bad written User Story into many smaller and better User Stories.

Story Splitting: Example ATTACH

```
As a Virtual Service Provider (VSP) and client o
on behalf of my end customers rapidly and reliab
me rather than switch to T or another VSP.
```

Splitting Criterion	Relevance in T	New Story Snippets (Subject to Component Identification)
Workflow Steps	High, order has 9 steps	Place order, Confirm order
Business Rule Variations	Number porting desired? Number porting possible?	Simulate change in customer database and billing system, reserve resources in telephony network
Major Effort	Relocation vs. new service	Create PSTN, move PSTN (to another address)
Simple/Complex	Visit of technician needed?	Schedule technician visit
Variations in Data	Mobile vs. fixed line telephony	Create new mobile service
Data Entry Methods	Browser UI vs. Rich Client App	Create new PSTN via Web API, Create new PSTN in browser
Defer Performance	Time to complete order	Respond to VSP (confirm order) within 4 hours (relaxed QA)
Operations (e.g. CRUD)	Create, Read, Update, Delete, Search (CRUDS) on orders	Request order status, Cancel order; Search order by id, by customer segment, by VSP, by VSP type
Break Out a Spike	20 backend systems	(Run a proof of Technology, mock each backend connection)

Figure 5: Example output after Story Splitting

6 Tactic DDD

Domain-Driven Design Domain-Driven Design is a software design approach, focusing on modeling software to match a domain according to input from a domain's experts.

> Under domain-driven design, the structure and language of software code (class names, class methods, class variables) should match the business domain. For example, if software processes loan applications, it might have classes like loan

application, customer, and methods such as accept offer and withdraw.

Try to avoid naming like in Figure 6.

Figure 6: Aligning organization and architecture with strategic DDD

Tactic DDD In tactic / tactical DDD you focus on how to build a Domain Model from a few basic building blocks.

Entities A potentially changeable object with an identity (Person)

Value Objects An immutable object, without identity (int, char, Design Pattern - Value Object)

Aggregates Combines / Groups Entities & Value Objects into cohesive units (Car consists of an Engine, Wheels, …)

- The parent *Entity* of this cluster is called *Aggregate Root*.

Services Stateless objects which performs logic not fitting in *Entities* or *Value Objects*

Repositories Are used to deal with storage. Used to save / load aggregates.

Factories Are used to build complex *Entities*, *Value Objects* or *Aggregate Root*

Events An immutable representation of something that happened in the domain

- Often exchanged between *Aggregates*

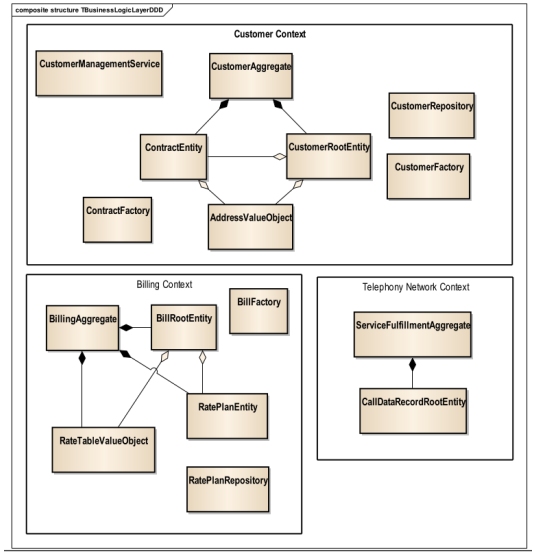

Figure 7: Tactic DDD Example

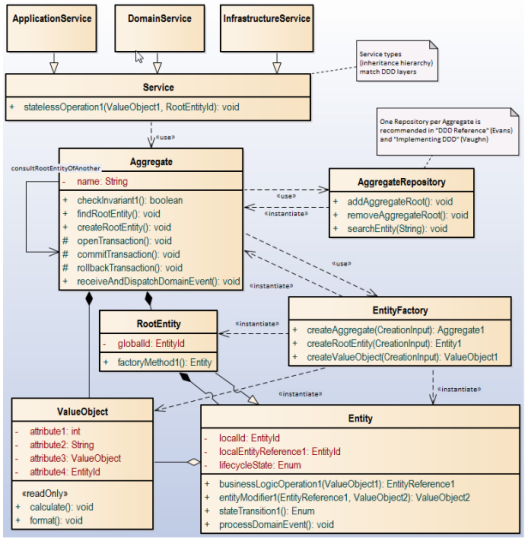

Figure 8: Tactic DDD Example 2

CRC Cards CRC Cards (Class-Responsibility-Collaboration Cards) is a brainstorming method to design and find components. Original from the OO world, but is well suited for software architecture and web.

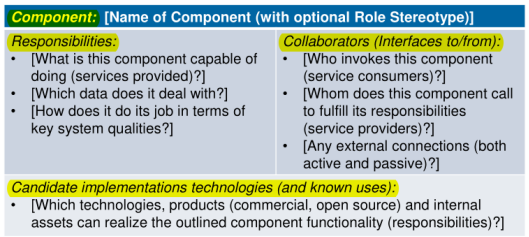

Figure 9: CRC Card Template

7 Strategic DDD

Business Rule A Business Rule has (at least) two meanings:

1. Executable part of the business logic (algorithm) which is expressed declarative
   - if [cond.] then [do]
   - rules as in logical programming
2. A Statement or condition about the domain model
   - E.g. the sum of all withdrawals is equal to the sum of all payments

Strategic DDD In Strategic DDD you will define

- Bounded Contexts
- Ubiquitous Language
- Context Map

Bounded Context When you try to model a big Domain Model you may encounter the problem that different groups use different terms. A Bounded Context is mainly for linguistic purpose. Each Bounded Context has its own Ubiquitous Language.
As your model evolves you may want to create relationships between Bounded Contexts. For this we have the Context Map.

Ubiquitous Language Inside a Bounded Context you speak a ubiquitous language. This language is determined in collaboration with the whole team and the domain expert. The language should uniquely identify each concept in this context (sub domain).

Context Map In a big Domain Model you will have multiple Bounded Context. At some point you have to bring this Bounded Context in a relationship. The context map should help here.
The Context Map has the following relationships:

Shared Kernel (SK) Two bounded contexts use a common kernel of code (for example a library)

Open Host Service (OHS) A Bounded Context specifies a protocol by which any other bounded context can use its services (e.g. RESTful HTTP)

Published Language (PL) The interacting bounded contexts agree on a common a language (for example a bunch of XML or JSON schemas)

Conformist (CF) BC uses the services of another but is not a stakeholder (e.g. public API)

Anti-Corruption Layer (ACL) BC uses the services of another but is not a stakeholder (e.g. public API). But has build a layer to minimize impact from changes in the other bounded context.

Customer/Supplier (CS) BC uses the services of another but is a stakeholder (customer). It can influence the development in the other BC.

The relation also needs to specify how the data flow is:

> an upstream context will influence the downstream counterpart while the opposite might not be true. This might apply to code (libraries depending on one another) but also on less technical factors such as schedule or responsiveness to external requests.

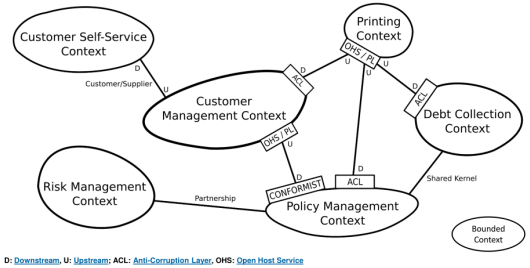

Figure 10: Context Map Example

8 Messaging

Lose Coupling Loose coupling means that two or more components does not depend on each strictly. Every component could be replaced if desired.
Loose coupling has (at least) four dimensions:

- Reference autonomy (aka. location transparency)
  - producer & consumer don't know each other
- Platform autonomy
  - Producer & consumer may be located in different technical environments, different languages
- Time autonomy
  - producer & consumer access channel at their own pace

- Format autonomy
  - Producers & consumer may use different formats of data exchanged

Integration Styles Integration Styles are used to communicate between different components. A few examples:

File Transfer e.g. `java.io` or FTP
Shared Database e.g. JDBC/SQL
Remote Procedure Call/Invocation (RPC, CPI) e.g. Java RMI, gRPC
Messaging e.g. JMS, RabbitMQ

Integration Style Messaging A message is transmitted in five steps:

1. Create: the sender creates the message
2. Send: the sender adds the message to a channel
3. Deliver: the messaging system moves the message from sender to receiver
4. Receive: the receiver reads the message from the channel
5. Process: the receiver extracts the data from the message

The Messaging Pattern has many variations.

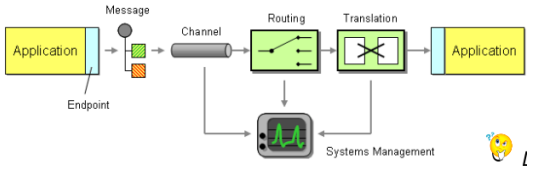

Figure 11: Enterprise Integration/Messaging

Document Message Pattern Send Document (JSON, XML)
Command Message Pattern Command Message to invoke a procedure in another application
Point-to-Point Channel Only one receiver will receive a particular message
Publish-Subscribe Channel A sender sends the message to many receivers

AMQP AMQP is a Peer-to-Peer Transport Protocol operating over TCP for message-oriented middleware. AMQP is a great protocol to use the Messaging Integration Style.

9 Microservices

Service vs Components A Component is a piece of software which can be used by foreign applications without changing its source code (e.g. JAR, DLL, …). A Service is similar to a component, but you will use it remotely through some remote interface (gRPC, Messaging, …).

Clean Architecture In the clean architecture you have rings. The further out the code is, the more volatile it is. The more in the center, the more stable is the code. So your domain code is in the center and at the outer ring are things like UI frameworks.

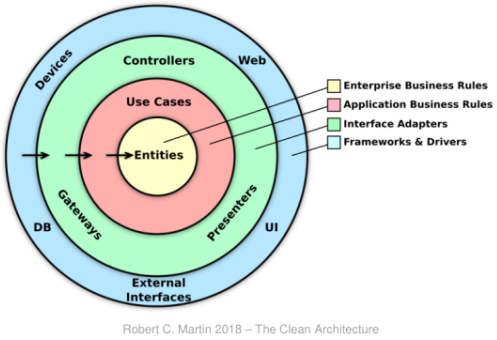

Figure 12: Clean Architecture

Hexagonal Architecture The Hexagonal Architecture / Onion Architecture / Ports and Adapters Pattern is a way how you can structure your application.

On the outside are the *adapters*. Adapters are concrete and are using libraries, services, databases, … In the inside are the *ports*. They are not interested in the concrete service / component (What are the differences between components and services). Therefor, you implement against interfaces (dependency inversion principle).

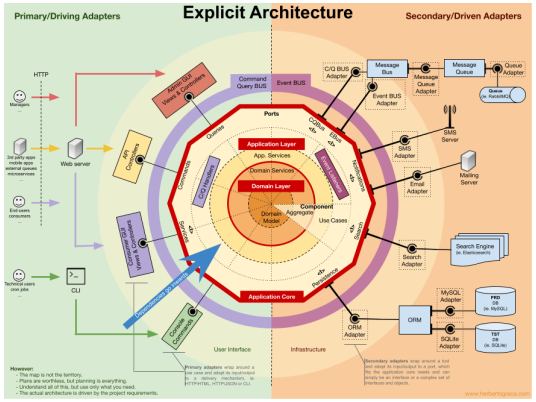

Figure 13: Hexagonal Architecture

Service-Oriented Architecture Service-Oriented Architecture is:

- a set of services and operations (Business Domain Analyst)
- an architectural style (IT Architect)
- a set of architectural patterns (IT Architect)
- a programming and deployment model (Developer, Administrator)

Microservices Microservices is an architecture type consists of many

- independently deployable
- scalable
- changeable

services.
Each service encapsulates and manages its *own* store and communicates via message-based remote APIs (Messaging). Idealy in a loosely coupled fashion (What does loose coupling mean).

Pro / Con Microservices Pros:

- single components can be scaled
- support for agile software development and continues delivery
- well suited for implementing 'IDEAL' cloud-native applications
- allows an incremental migration of monolithic applications

Cons:

- communication overhead within a distributed architecture
- requires a disciplined approach to their life cycle management, monitoring, and debugging
- single points of failure and cascading failure proliferation effects need to be avoided
- data consistency and state management challenges are introduced
- autonomy and consistency for the whole microservice architecture cannot be both guaranteed

REST REST is an architectural style (for integration), defined via constraint. To be a REST conform you have to follow the following constraints:

- client-server
- stateless
- cacheable
- uniform interface (URI, HTTP)
- layered system
- code on demand (optional)

To be RESTFul you have also to satisfy the next four constraints:

- identification of resources
- manipulation of resources through representations
- self-descriptive messages
- hypermedia as the engine of application state (HATEOAS)

REST Maturity Levels An API on the web can be on different maturity levels:

- Level 0: The Swamp of POX (Plain Old XML Objects)
  - Send data in an arbitrary way with XML / JSON objects
- Level 1: Resources / HTTP Resource API (by Olaf Zimmermann)
  - You have meaningful URLs
- Level 2: HTTP Verbs / Web API (by Olaf Zimmermann)
  - You use the HTTP Verbs (GET, POST, …) in the correct way
- Level 3: RESTFul HTTP / Hypermedia API (by Olaf Zimmermann)
  - Navigation using links in response
  - They must contain application (flow) state

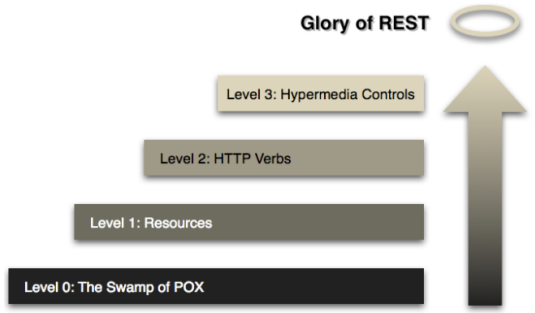

Figure 14: REST Maturity Levels

HATEOAS HATEOAS (Hypermedia As The Engine Of Application State) is constraint of the RESTFul style. In HATEOAS the server provides the possible next actions in the request in form of links (see Listing 1)

```
HTTP/1.1 200 OK
Content-Type: application/vnd.acme.
    account+json
Content-Length: ...

{
    "account": {
        "account_number": 12345,
        "balance": {
            "currency": "usd",
            "value": 100.00
        },
        "links": {
            "deposit": "/accounts/12345/
                deposit",
            "withdraw": "/accounts
                /12345/withdraw",
            "transfer": "/accounts
                /12345/transfer",
            "close": "/accounts/12345/
                close"
        }
    }
}
```

Listing 1: HTTP Response

From Monolith and Components to SOA

- **Monolitich application** aka the big ball of mud
- **Internally componentized application**
  - *Logic* könnte ein Aggregate sein!
  - modular monolith
- **Microservices application**

10 API Description

Describe API To describe / document an API at least 3 information are required:

- API Endpoint Information (e.g. URL / URI / …)
- Operation Names (e.g. HTTP verbs)
- Message Content (structure and meaning, e.g. using MIME, Content Type)

A more complete description / documentation contains a few additional information:

- Parameter Data Types (string, int, object, …)
- Usage Examples
- Error Codes and Reports
- Compliance Test Cases
- Behavior
- Versioning Metadata

Interface Definition Language An Interface Definition Language is used to describe an API in a Programming Language independent way. One of the most common one for REST APIs is OpenAPI Specification. An alternative WDSL (Web Services Description Language).

OpenAPI Specification The OpenAPI Specification (OAS) is an IDL to describe REST APIs. The OpenAPI Specification is often written in JSON or YAML.

11 Service Granularity

Service Granularity Service Granularity is the size of a service.

> Granularity is not defined by the number of classes or lines of code in a service, but rather what the service does. – Dr. Gerald Reif

To find the right size / granularity for a service we have Service Granularity Integrators and Service Granularity Disintegrators.

Service Granularity Disintegrators Provide guidance and justification for when to break a service into smaller pieces.

Service scope and function Is the service doing too many unrelated things?

Code volatility Are changes isolated to only one part of the service?

Scalability and throughput Do parts of the service need to scale differently?

Fault tolerance Are there errors that cause critical functions to fail within the service?

Security Do some parts of the service need higher security levels than others?

Extensibility Is the service always expanding to add new contexts?

Service Granularity Integrators Provide guidance and justification for putting services back together (or not breaking apart a service in the first place).

Database transactions Is an ACID transaction required between separate services?

Workflow and choreography Do services need to talk to one another?

Shared code Do services need to share code among one another?

Database relationships Although a service can be broken apart, can the data it uses be broken apart as well?

Async vs sync Two services can communicate in a synchronous or asynchronous (async vs parallel) manner. Both have their pros and cons.
sync contra:

- performance impact on highly interactive systems
- creates dynamic entanglements
- creates limitations in distributed architectures

sync pro:

- easy to model transactional behavior
- mimics non-distributed method calls
- easier to implement

async contra:

- complex to build and debug
- presents difficulties for transactional behaviors
- error handling

async pro:

- allows highly decoupled systems
- common performance tuning technique
- high performance and scale

Orchestrator An Orchestrator implements the whole workflow logic. The different services are not communicating directly, but over the orchestrator. This is an implementation of the Design Pattern - Mediator.

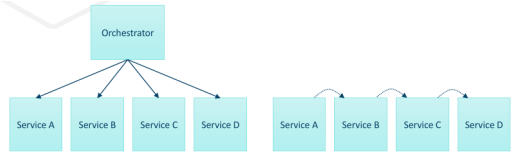

Figure 15: orchestration vs choregraphy

orchestration vs choreography orchestration contra:

- slow responsiveness
- fault tolerance (Orchestrator down, everything down)
- scalability (Orchestrator not easy scalable)
- service coupling (Orchestrator has to know every service)

orchestration pro:

- centralized workflow
- error handling (single point for error handling)
- recoverability (knows everything, greater change to recover)
- state management (management in a single point)

choreography contra:

- distributed workflow (Workflow is distributed over all services)
- state management (difficult because no single place for state)
- error handling (every service has to it by itself)
- recoverability (difficult because service knows only its own information)

choreography pro:

- responsiveness (first service can react immediately)
- fault tolerance (single service down, other services works)
- scalability (you can scale each service individually)
- service coupling (each service only has to know the service which communicates with)

atomicity vs eventual consistency *Atomicity* guaranties that each transaction is treated as a single ”unit”. This unit succeeds completely or fails completely.
Eventual Consistency guaranties if no new updates are made to a given data item, eventually all access to that item will return the last updated value. Language: [en] eventual -> [de] schlussendlich, irgendwann, letzten Endes – **Nicht eventuell!**

12 Session State

Session State Management In general there exists 3 types how / where you can manage / store state:

- Client Session State (REST Level 3)
  - cookies, hidden fields, URL (for REST no cookies!), JWT
- Server Session State
  - uses main memory of the application server
  - not recommended because you very difficult to scale
- Database Session State
  - database and service can be scaled separately

13 Patterns

Architecture Pattern - Layering

Context A system that needs to be segmented due to its size and specialization.
Problem System design that supports a mixture of tasks on different levels.
Solution Organize the system into cohesive layers that are stacked on top of each other.

This pattern is often applied twice (logical and physical). Logical layers (layers):

- presentation layer
- business logical layer
- data access layer

Physical layers (tiers)

- Browser
- Application Server
- Database Server

Architecture Pattern - Distribution Pattern The Distribution Pattern describes how you can partition your application in client and server components.

- Distribution Pattern - Remote User Interface
- Distribution Pattern - Distributed Presentation Layer
- Distribution Pattern - Distributed Application Kernel
- Distribution Pattern - Remote Database
- Distribution Pattern - Distributed Database

Distribution Pattern - Remote User Interface In *Remote User Interface* the tier 1 runs dialog control sub-layer (the C and the M in MVC) in addition to display. An example for this pattern are today's Single-Page Applications (JavaScript performs calls to API).

Distribution Pattern - Distributed Presentation Layer In the *Distributed Presentation Layer* the tier 1 client only renders display elements (the V in MVC). Tier 2 owns/-controls screen/page flow.

Example for this pattern are traditional (MVC) PHP application.

Distribution Pattern - Distributed Application Kernel In the *Distributed Application Kernel* pattern the business logic is **physically** distributed. An example for this pattern are Microservice.

Distribution Pattern - Remote Database In the *Remote Database* pattern you use a Remote Database Access Protocol to connection your application to the database. Example for this is JDBC/ODBC (Java) or sqlx (Rust, Go). The transaction management is performed by the database itself.

Distribution Pattern - Distributed Database In the *Distribution Database* your database is partitioned and/or partially replicated. This can be done using a Two-Phase Commit with an external transaction manager.

Design Pattern - Template View You insert markers in the markup (for example HTML). These markers are then replaced at runtime with real values. Known uses are:

- Handelbars (JS)
- tera (Rust)

  Renders information into HTML by embedding markers in an HTML page.