

Lernportfolio

Joel Maximilian Mai

Matrikelnummer 11118561

25. April 2019

TH KOELN

WEB DEVELOPMENT



Inhalt

[Workshop 04.04.19 2](#_Toc7068014)

[Wissensstand: 2](#_Toc7068015)

[Requirements: 2](#_Toc7068016)

[Usecases: 3](#_Toc7068017)

[Vereinfachtes Domainenmodell: 3](#_Toc7068018)

[Recap: 4](#_Toc7068019)

[Proof of Concepts: 4](#_Toc7068020)

[Anforderungen: 4](#_Toc7068021)

[Versprechungen vom Dienst: 4](#_Toc7068022)

[Selbststudium mit dem Buch „Reactive Design Patterns“ 5](#_Toc7068023)

[Notizen zu dem Buch 5](#_Toc7068024)

[Sharding-Patterns: 5](#_Toc7068025)

[Responsives System entwickeln: 5](#_Toc7068026)

[Integrating nonreactive Components: 6](#_Toc7068027)

[Reacting to Users: 6](#_Toc7068028)

[Understanding the traditional approach: 6](#_Toc7068029)

[Analyzing latency with shared resource: 6](#_Toc7068030)

[Limiting maximum latency with a queue: 6](#_Toc7068031)

[Exploiting parallelism: 7](#_Toc7068032)

[The limits of parallel execution: 7](#_Toc7068033)

[Reacting to failure 7](#_Toc7068034)

[Compartmentalization and bulkheading 7](#_Toc7068035)

[Using circuit breaker 7](#_Toc7068036)

[Losing strong consistency 7](#_Toc7068037)

[BASE 8](#_Toc7068038)

[Accepting Updates 8](#_Toc7068039)

[The need for Reactive Design Patterns 8](#_Toc7068040)

[Workshop 11.04.19 9](#_Toc7068041)

[Domaindiagramm: 9](#_Toc7068042)

[Erster Aufbau des Services: 9](#_Toc7068043)

[Lernziele bis zum 25.4: 10](#_Toc7068044)

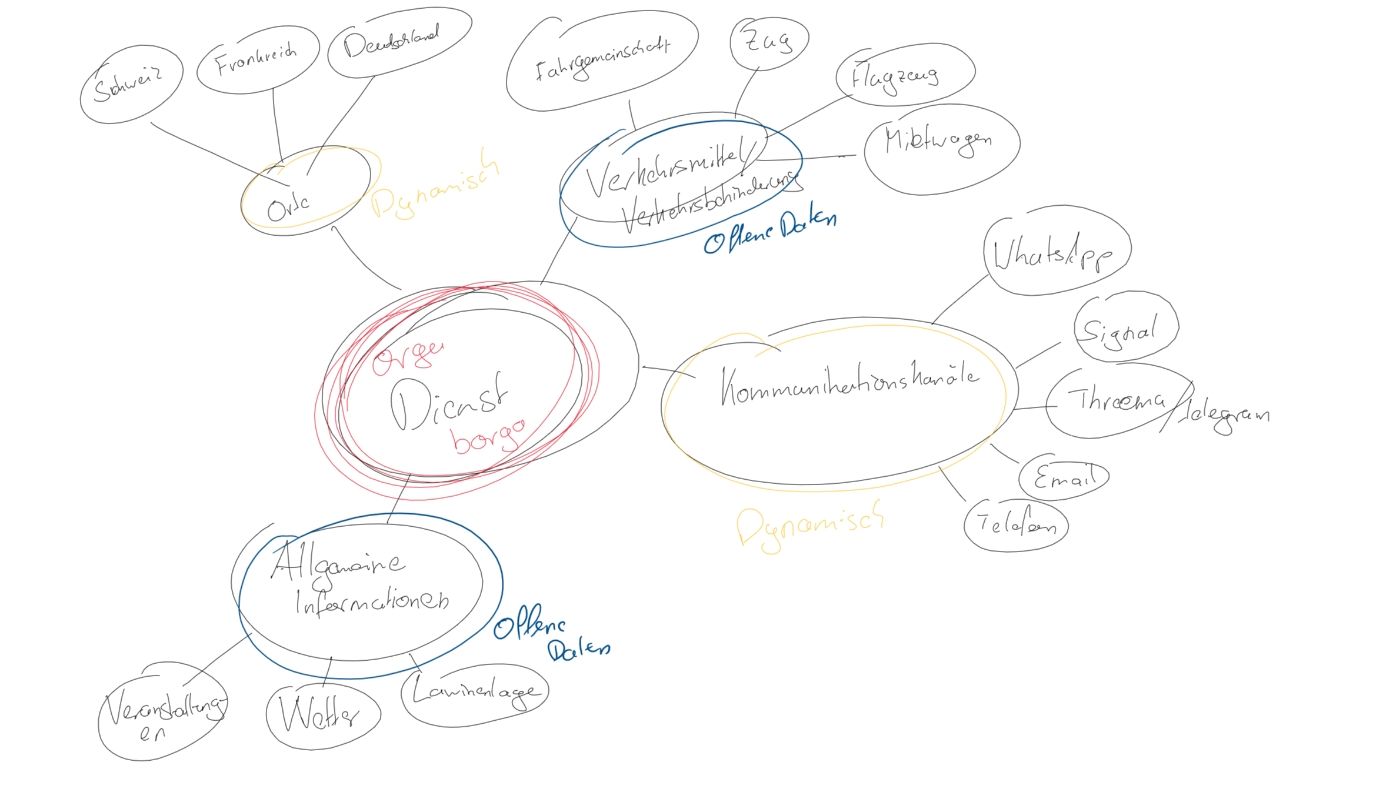
[Lernziele am 25.4: 10](#_Toc7068045)

# Workshop 04.04.19

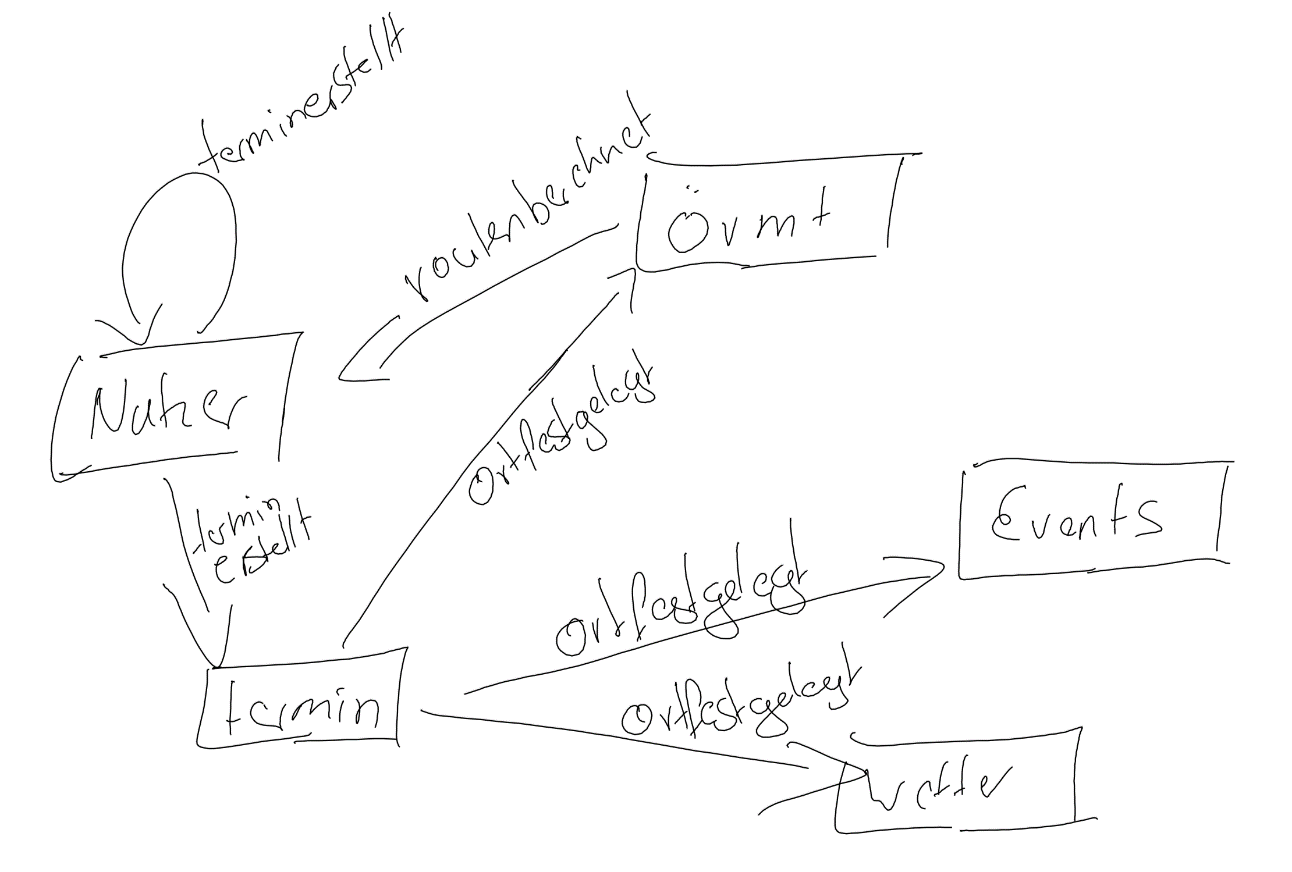
## Wissensstand:

Nach kurzer Recherche der notwenigen Kenntnisse für einen online Dienst, wurde mir schnell klar, dass ich mein Javascript-Wissen auffrischen und mir XML anlernen muss.

## Requirements:

Wir erarbeiteten zu aller erst die sich aus dem Szenario ergebenden Requirements:

Als Lösungsvorschlag wählten wir ein asynchrones Peer-to-Peer Netzwerk. Bestehend aus Nutzern und Services – Wir entwickelten eine erste Skizze eines Domain-Diagramms:



Wir diskutierten darüber ob es ein Zentrales oder Verteiltes System besser wäre für das Zusammenspiel der Services.

## Usecases:

Wir entwickelten die Usecases für unser System:

*Nutzer füllt Termin mit Daten: (offline)*

* *Ort*
* *Datum / Uhrzeit*
* *Teilnehmer (einladen)*
* *Name der Veranstaltung*
* *Beschreibung*

*Nutzer erstellt Termin (synchron)*

*Termin forcedJoined die Nutzer (asynchron)*

*Termin prüft Wetter am angefragten Ort (s)*

*Termin prüft Events in der Nähe (s)*

*Termin prüft Verkehrsbehinderung (s)*

*Termin sendet ggf. Informationen (a)*

*Optional:*

*Teilnehmer verlässt Termin (a)*

*Nutzer löscht Termin (s)*

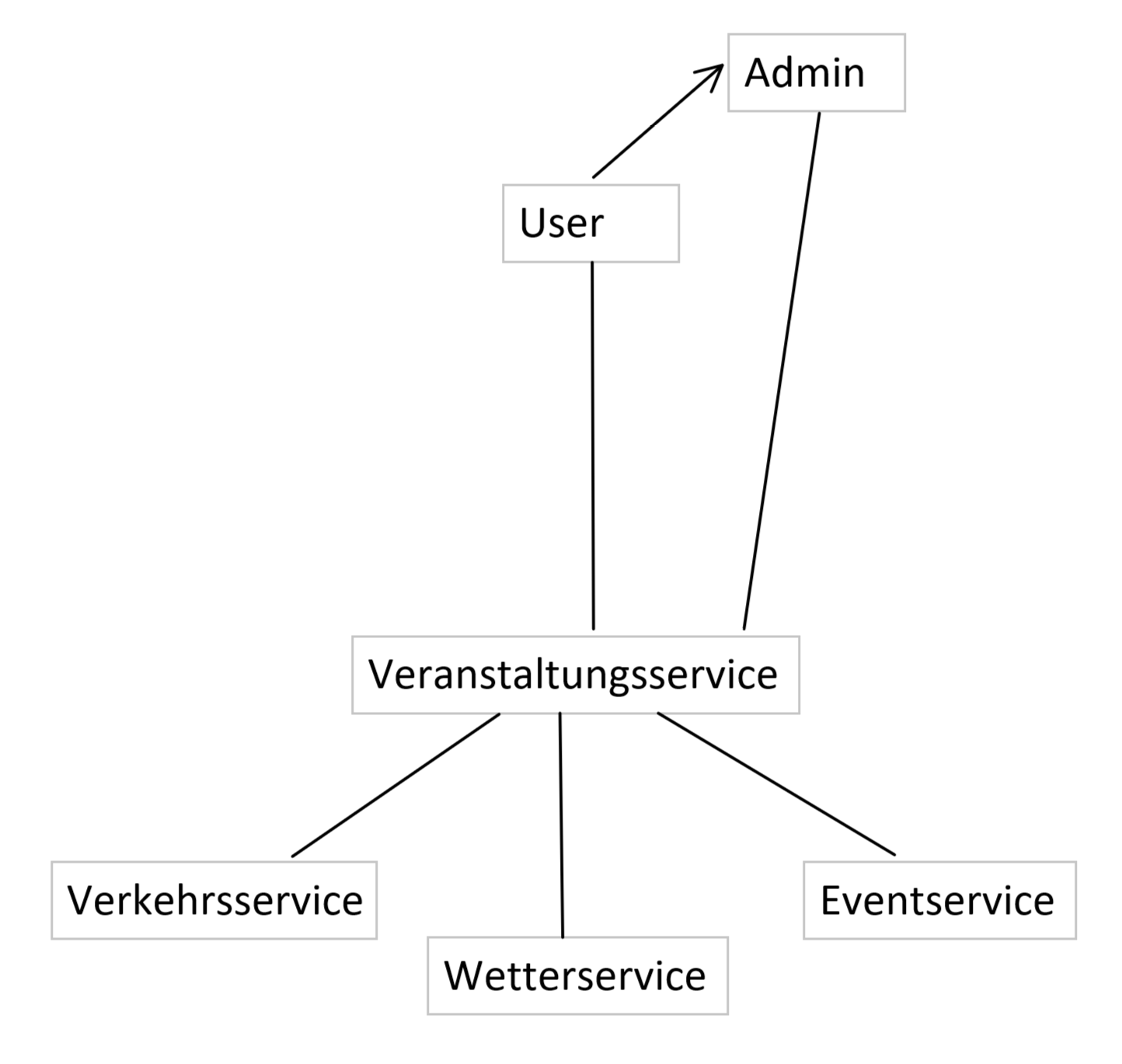
*Termin wurde gelöscht (a)*

*Teilnehmer sind selbst verantwortlich für ihre Reiseroute.*

Daraus ergeben sich die für uns relevanten Aktoren unseres Domainklassendiagramms.

* *Nutzer (Admin) / Teilnehmer (User)*
* *Veranstaltungsservice*
* *Verkehrsservice*
* *Wetterservice*
* *Eventservice*

## Vereinfachtes Domainenmodell:



## Recap:

Nach dem wir uns das Verhalten des Systems vorstellen konnten, stellten wir uns allerdings die Frage wie wir weiter vorgehen sollen. Natürlich könnten wir weiter Modelle bauen, aber das würde unsere nicht vorhandenen Kenntnisse über XML/XMPP nicht besser machen.

Wir haben vor die Tutorials zu Javascript und XML zu erarbeiten:

* <https://www.w3schools.com/Js/js_intro.asp>
* <https://www.w3schools.com/xml/default.asp>
* <https://www.w3schools.com/nodejs/nodejs_intro.asp>
* GDW Inhalte aufarbeiten (da noch nicht teilgenommen)
* Reactive Design Patterns lesen

### Proof of Concepts:

* JavaScript/Java -> Programmiersprache?
* XMPP/XML -> Chat austausch und Veranstaltung

### Anforderungen:

* Eventbasiert – zumindest weitgehend
  + Asynchrone Kommunikation zwischen Veranstaltung und User
* Open Data Portale
  + Synchrone Datenabfrage jedoch asynchrone Weiterleitung

### Versprechungen vom Dienst:

* Nachrichtenaustausch über eine Gruppe der Veranstaltung
* Nachrichten direkt von der Veranstaltung
* Statusupdates der Verkehrs- und Wetterlage oder anderen Events im Umkreis
* Nachträgliche Modifikationen an den Veranstaltungsdetails

# Selbststudium mit dem Buch „Reactive Design Patterns“

## Notizen zu dem Buch

* Responsive = Reaktion auf den User
* Resilient = Reaktion auf Versagen des Systems und das überkommen und aktiv bleiben dessens
* Elastic = Flexibel und Funktionstüchtig bei der Menge des Workloads
* Message-Driven = Reaktion auf verschiedene Inputs

### Sharding-Patterns:

* Kopieren eines Systems um Verfügbarkeit zu garantieren -> Replicate
* Active-Passive Replication:
  + Festlegen welches Replikat Updates akzeptiert
  + Dieser leitet dann das Update weiter an die Anderen
  + Erst wenn dieses ausfällt übernimmt ein anderes Replikat
* Consensus-based Multiple-master Replication:
  + Jedes Update wird von allen Replikas bestätigt und übernommen
  + Damit alle konsistent (Consistent) sind aber dadurch sind die anfällig für Verzögerungen (Latency) oder nicht Verfügbar(Availability) auf Grund des höheren Workloads
* Optemistic Replication (with conflict detection and resolution):
  + Mehrere aktive Repliken verbreiten Updates und führen Transaktionen zurück, wenn Konflikte auftreten oder verwerfliche Updates verworfen werden, die während einer Netzwerkpartition durchgeführt wurden
* Conflict Free Replication:
  + Dieser Ansatz schreibt Zusammenführungsstrategien so vor, dass Konflikte nicht definitionsgemäß entstehen können, jedoch nur die eventuelle Konsistenz erfordert und besondere Sorgfalt bei der Erstellung des Datenmodells erforderlich ist

### Responsives System entwickeln:

* Applikation läuft lokal und speichert Emails und bla lokal
* Synchronisiert mit Server/-n
  + Erzeugt Spannung zwischen dem Wunsch das System zu verteilen und responsiv zu bleiben
* Circuit Breaker Patterns
  + Beobachtet das Anfrageverhalten des Nutzers und schaltet sich zwischen, wenn zu viele Anfragen gemacht werden
  + Zusätzliche Features:
    - Flow Control
    - Backlogs
  + Anwendungsbereich von Circuit Breaker Patterns
    - Zwischen Anwendungsebene des Users und Webserver
    - Zwischen Webserver und Backend Services
* Wichtig ist auch die Einstufung des Features: Essential / Nonessential
* Avoiding the ball of mud:
  + Klassische Visualisierung eines Systems ist Frontend Backend
  + Jedoch durch die Erweiterung/Verteilung des Backends auf verschiedene Services
  + Kann ein Big ball of mud entstehen wenn die Kommunikationswege
  + Nicht frei sind
* Spezielle Designform: Message-Flow / Message-Driven

### Integrating nonreactive Components:

* Device Drivers
* API (Eher Synchron und daher können sie die Funktionalität der App blockieren)
* Daher wichtig: Resource-Management Patterns
  + Ständiger retrofit des Systems
  + Extra Threads, Prozesse oder Maschinen wenn nötig
  + Festlegen was bei einem Overload passieren soll
  + Auskapseln von APIS und sie auslagern so gut es geht um crashes zu vermeiden
  + IPC = Inter Process Communication
    - Pipes
    - Sockets
    - Shared Memory
* Verzögerungen beobachten und im Falle dieser:
  + Temporäre Fehler Meldungen
  + Warnungen in der Antwort
  + Nutzer benachrichtigen, dass
    - Später erneut versuchen
    - Oder über die Verzögerung
* Reactive Manifesto

### Reacting to Users:

* Nicht nur einzelne Komponenten betrachten sondern auch das Innere dieser
* Zusätzlich das Ziel haben sie so responsive wie nötig zu halten

### Understanding the traditional approach:

* Wenn Sie die Leistung eines Systems wie dieses optimieren, ist einer der Schlüsselparameter das Verhältnis von Anforderungsthreads zu Einträgen des Verbindungspools. Es macht nicht viel Sinn, den Verbindungspool größer als den Request-Thread-Pool zu machen.
* Verbindungspool zu klein -> Bottleneck
* Die beste Antwort für eine gegebene Last liegt irgendwo zwischen den Extremen. Der nächste Abschnitt befasst sich mit dem Finden eines Gleichgewichts.

### Analyzing latency with shared resource:

* Bei Anfragen die darauf warten prozeduiert zu werden, benötigt man immer eine Art von Queue Date Structure
* Lösungsansätze:
  + Controller
  + TCP Buffer
  + Cache
  + Fallback
* WICHTIG: Immer auf Anfragen Overload vorbereiten und System beschützen

### Limiting maximum latency with a queue:

* Erster einfacher Schritt: If no database connaction available -> Return null
* Einführung einer Explicit Queue
  + Statt direkt einen Error zu senden können hier noch Anfragen zwischen geparkt werden
  + Erst wenn diese Queue voll ist wird abgewiesen

### Exploiting parallelism:

* Meisten Programme erwarten dass ihre Funktionen eine Anwort liefern, tun sie dies nicht funktioniert das ganze Programm nicht
* Lösung:
  + Subtasks einzeln betrachten und parallel schalten -> Die Aufgabe dauert nur so lange wie die Längste Subtask
  + Und das Ergebnis nicht als zwingend passend betrachten -> Try Catch
  + Fehlerwerfende Threads suspendieren um den Platz für neue Threads freizugeben

### The limits of parallel execution:

* Um von diesem Wachstum zu profitieren, müssen Sie die Berechnungen sogar auf einer einzigen Maschine verteilen.
* Ein herkömmlicher Ansatz ist wenn ein gemeinsamer State verwendet wird, der auf gegenseitigem Ausschluss durch Sperren basiert, werden die Kosten für die Koordinierung zwischen Kernen sehr CPU-bedeutend.

### Reacting to failure

* Software will fail
* Hardware will fail
* Humans will fail
* Timeout is failure
* Resillience:
  + Die Fähigkeit einer Substanz oder Objekt wieder zurück in eine vergange Form sich selbst zu bringen
  + The Kraft sich schnell von Schwierigkeiten oder Anstrengungen zu erholen
* Wichtige Teilaufgaben eines Systems zu deligieren
* Distribute and compartmentalize
* Wichtige Daten in kopierter Form zu sichern
* Wichtige Kopien von Systemzentren auf verschiedene Stromnetze oder gar Länder oder Kontinente zu verteilen

### Compartmentalization and bulkheading

* Falls die Compartments nicht hundert prozentig von einander getrennt sind, kann ein Systemfehler in einem von ihnen das gesamte System zum crashen bringen.

### Using circuit breaker

* Wenn die Antwort einer Anfrage bereits nicht mehr benötigt wird, ist es sinnlos sie überhaupt noch zu bearbeiten
* Falls der Circuit Breaker eine Verzögerung im Systemablauf feststellt kann er Anfragen von akzeptiere-alle auf wartet-hier-und-kommt-nach-der-wartezeit-dran oder gar ich-nehme-nichts-mehr-an umstellen um dem System Zeit zu geben sich zu erholen und die eigenen Warteschleifen zu leeren
* Wie viele Reconnection-Attempts machen Sinn?
* Wie lange sollte man zwischen ihnen warten?

### Losing strong consistency

* Nimm zwei von drei:
  + Konsistenz – Alle Kopien beeinhalten identischen Inhalt
  + Hohe Verfügbarkeit der Daten für Updates
  + Toleranz zu Netzwerkpartitionen

### BASE

* Verteilte Systeme sind meist auf verschiedenen Prinzipien gebaut
* Eins davon heißt BASE
  + Basically available
  + Soft State (Ein Zustand der aktiv versucht erreicht zu werden statt von einem Default Zustand auszugehen)
  + Eventually Consistent
* Der Letzte Punkt bedeutet, dass zwar zu manchen Zeitpunkten ein inkonsestenter Zustand herrschen kann, dieser jedoch mit der Zeit wieder konsestent wird.
* BASE setzt sich zusammen aus
  + Associative
    - Jede Aktion kann in Stapeln erfolgen (Assoziativ)
  + Commutative
    - Jede Aktion kann in jeder Reihenfolge erfolgen (Kommutativ)
  + Idempotent
    - Jede Aktion kann beliebig oft durchgeführt werden
  + Distributed

### Accepting Updates

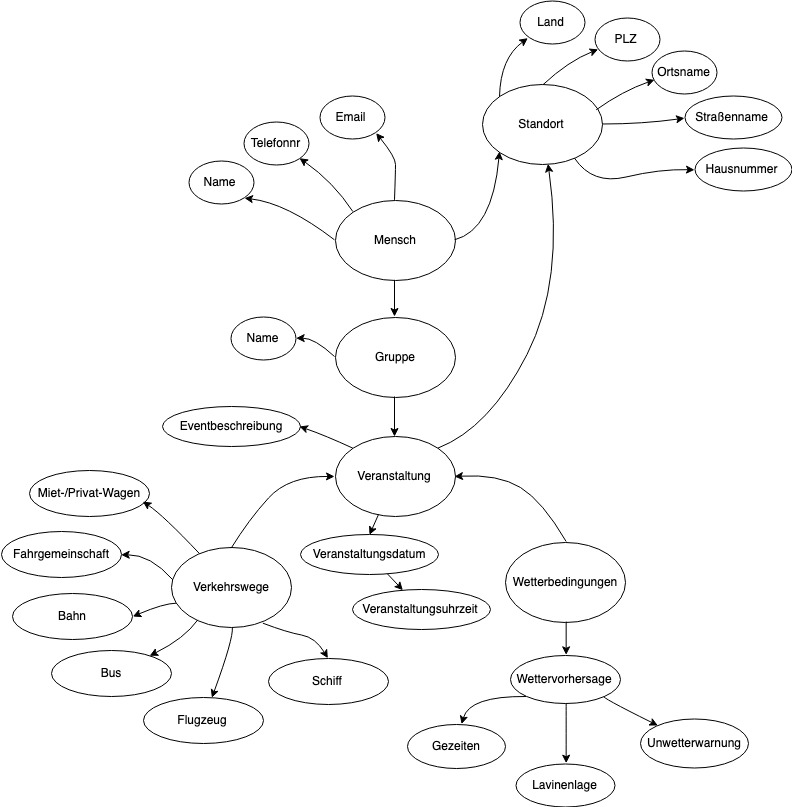
* Wichtig hierfür sind CRDTs (Conflict-Free-Replicated-Data Types)
* Wie man zwei Dokumente wieder konsistent bekommt
  + Die beiden Änderungsketten nicht simultan bearbeiten sondern über einander legen
  + Erst die eine Kette anwenden und dann die Andere
  + Bis beide Partitionen „geheilt“ sind
* Wenn das System offline oder einfach unerreichbar ist, ist die Lösung dem Nutzer immernoch manche Feautures anzubieten.
* Man könnte auch das ganze so sehen, dass das System auf einen ungefähren Zustand wechselt bis eine Verbindung wieder hergestellt werden konnte

### The need for Reactive Design Patterns

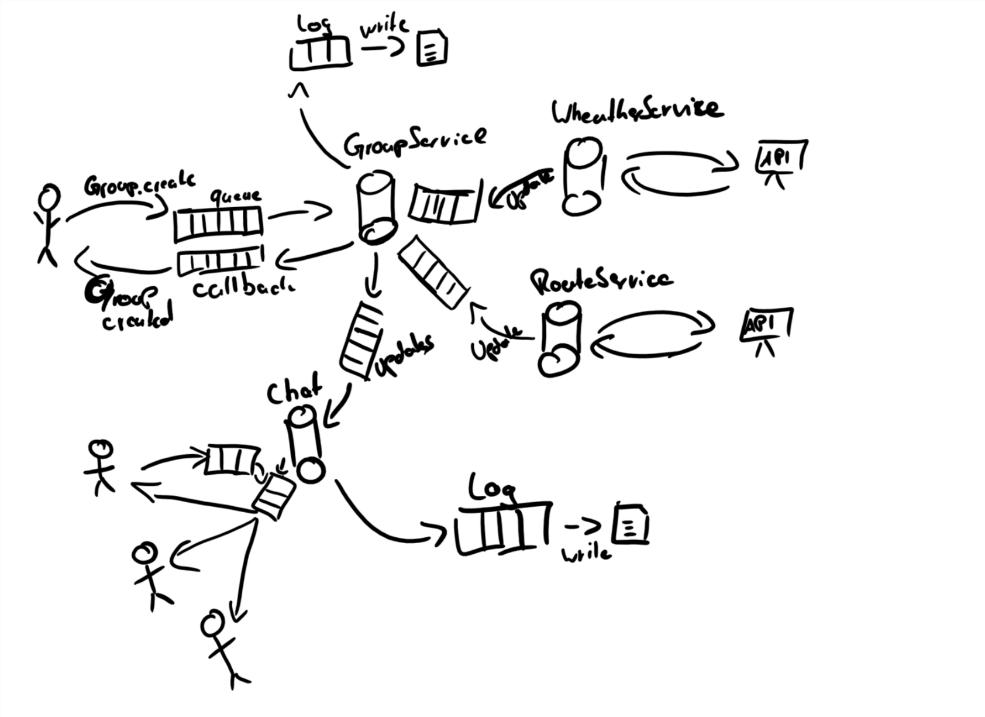
* SOA = Service-Oriented Architecture
* Managing Complexity
  + Essential Complexity = Ausgelöst durch ein Problem
  + Incidential Complexity = Ausgelöst durch die Lösung
  + Message oriented Development führt genauso zu einem konsistenten Service wie die Synchronen Entwicklungen und weil sie deutlich leichter sind zu implementieren werden sie immer öfter benutzt

# Workshop 11.04.19

### Domaindiagramm:



### Erster Aufbau des Services:



### Lernziele bis zum 25.4:

* Node JS + RabbitMQ zum laufen bringen
  + Nodejs lernen und alles nötige installieren
  + Client aufstellen (proto)
  + Servicemodule aufstellen (proto)
  + Beschränkte API-Einbindung erreichen (issues)
* Gedanken machen wie wir den Service aufbauen
* Domainmodell fertigstellen

### Lernziele am 25.4:

* Servicemodule und Client komplexieren
* Alle auf denselben Stand bringen
* Lernportfolio fertigstellen