

Lernportfolio

Moritz Dahlke

Matrikelnummer 11117538



Inhalt

[Workshop am 04.04.19 3](#_Toc7096290)

[Ziele des Kurses „Frameworks, Dienste und Daten im Web“: 3](#_Toc7096291)

[Vorhandenes Vorwissen: 3](#_Toc7096292)

[Plan und Fortschritt: 3](#_Toc7096293)

[Selbststudium zum Thema „Reactive Design“ ( Kapitel 1 & 2 ) 4](#_Toc7096294)

[Workshop am 11.04.19 9](#_Toc7096311)

[Plan und Fortschritt: 9](#_Toc7096312)

[Workshop am 25.04.19 ( Meilenstein 1 ) 9](#_Toc7096313)

[Ausgang: 9](#_Toc7096314)

[Plan und Fortschritt: 10](#_Toc7096315)

# Workshop am 04.04.19

## Ziele des Kurses „Frameworks, Dienste und Daten im Web“:

* Gezielter, selbstständiger Wissenserwerb mittels verschiedener Quellen
* Auswahl und Implementierung von Frameworks für serverseitige Anwendungen
* Verständnis über asynchrone, serverseitige Kommunikation und Framework Standards
* Verständnis über offene Daten im Web

## Vorhandenes Vorwissen:

Die Wahl des Szenarios wurde vor allem wegen vorherigen Arbeiten mit serverseitigem NodeJS und Telegram Bots getroffen. Ein dafür benötigtes, brauchbares Javascript Skillset ist vorhanden und kann angewendet werden.

Es war kein Vorwissen in Sachen Kommunikationsprotokolle wie XMPP oder RabbitMQ vorhanden, was auf jeden Fall durchs Selbststudium geändert werden sollte.

Grundlagen der Softwaremodellierung waren aus dem Fach Softwaretechnik 1 bereits bekannt, allerdings gab es Schwierigkeiten beim Erstellen eines Domänenmodells, da es unterschiedliche Varianten zu geben scheint.

## Plan und Fortschritt:

* Erstellung eines angemessenen Domänenmodells als Voraussetzung zur Zielsetzung

Die Erstellung eines Domänenmodells bedarf mehrere Anläufe, jedoch konnte durch etwas Recherche und Kommunikation mit dem Kursleiter ein entsprechendes Modell angefertigt werden, mit dem wir zufrieden waren.

* Zielsetzung

Die Zielsetzung für den Workshop am 11.04.2019 bestand aus der Aufbereitung der vorgeschlagenen Kapitel des Buches „Reactive Design Patterns“, sowie NodeJS (Javascript) und der Erhaltung von Informationen über Kommunikationsprotokolle wie XMPP oder RabbitMQ, um sich später auf eines festlegen zu können. Es standen nur diese Protokolle zur Auswahl, da diese einen relativ hohen Stellenwert besitzen.

# Selbststudium zum Thema „Reactive Design“ ( Kapitel 1 & 2 )

Das Selbststudium erfolgte in Gruppenarbeit, um ein ausreichendes Verständnis zu ermöglichen.

## Computer Standards:

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

## Sharding-Patterns:

* Replikation/Kopie eines Systems um Verfügbarkeit zu garantieren -> Replicate
* Active-Passive Replication:
  + Festlegen welches (aktive) Replikat Updates akzeptiert
    - Leitet Updates an die Anderen (passiven) weiter
    - Bei Ausfall übernimmt ein passives Replikat
* Consensus-based Multiple-master Replication:
  + Jedes Update wird von allen Replika bestätigt und übernommen
  + Damit sind zwar alle konsistent (Consistent), aber anfällig für Verzögerungen (Latency) oder nicht verfügbar (Availability) auf Grund des höheren Workloads
* Optimistic Replication (with conflict detection and resolution):
  + Mehrere aktive Replika 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:

* Anwendung läuft lokal und speichert Daten 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 / Non-Essential
* 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 (Normalerweise synchron – Kann Funktionalität der App beinflussen)
* Daher wichtig: Resource-Management Patterns
  + Ständiger retrofit (Ausbau) des Systems
  + Extra Threads, Prozesse oder Maschinen, wenn nötig
  + Festlegen was bei einem Overload passieren soll
  + Auskapseln von APIs und sie auslagern 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

## 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 (Engpass beim Transport von Daten)
* 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 prozessiert zu werden, benötigt man immer eine Art von Queue Data Structure (wie Stacks, aber an beiden Enden offen)
* 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 connection 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:

* Die meisten Programme erwarten, dass ihre Funktionen eine Antwort liefern, tun sie dies nicht funktioniert das ganze Programm nicht
* Lösung:
  + Sub-Tasks einzeln betrachten und parallel schalten -> Die Aufgabe dauert nur so lange wie die Längste Sub-Task
  + 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, sich wieder zurück in eine vergangene Form sich zu bringen
  + Die Kraft sich schnell von Schwierigkeiten oder Anstrengungen zu erholen
* Wichtige Teilaufgaben eines Systems delegieren
* Distribute and Compartmentalize (Verteilen und Gliedern)
* Wichtige Daten in kopierter Form sichern
* Wichtige Kopien von Systemzentren auf verschiedene Stromnetze oder gar Länder oder Kontinente verteilen

## Compartmentalization and bulkheading

* Falls die Compartments nicht hundert prozentig voneinander 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 Re-Connection Versuche ergeben Sinn?
  + Wie lange sollte man zwischen ihnen warten?

## Losing strong consistency

* Konsistenz – Alle Kopien beinhalten 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 Standard Zustand auszugehen)
  + Eventually Consistent
* Der Letzte Punkt bedeutet, dass zwar zu manchen Zeitpunkten ein inkonsistenter Zustand herrschen kann, dieser jedoch mit der Zeit wieder konsistent 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 immer noch manche Features 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
  + Zusammensetzung (Orchestrierung) von mehreren Services
* Managing Complexity
  + Essential Complexity = Ausgelöst durch ein Problem
  + Incidential Complexity = Ausgelöst durch die Lösung
  + Message-Oriented Development führt genauso zu einem konsistenten Service wie synchrone Entwicklungen und weil sie deutlich leichter zu implementieren sind, werden sie immer öfter benutzt

# Workshop am 11.04.19

## Plan und Fortschritt:

* Festlegung des Kommunikationsprotokolls: RabbitMQ
  + Bietet umfassende Dokumentation und Tutorials
  + Anwendung in vielen verschiedenen Programmiersprachen möglich
    - Eine davon: Javascript via NodeJS

Nach der Installation eines lokalen RabbitMQ Servers war ein kleines Script mit „Sender“ und „Empfänger“ mittels dokumentierter Tutorials und vorhandenen Kenntnissen schnell lauffähig. Die Tutorials wurden im Selbststudium gelesen, verstanden, sowie umgesetzt und nicht einfach kopiert. Das hat dazu geführt, dass der „Code“ sehr leicht anzupassen war.

* Festlegung von Lernzielen bis zum 25.04.2019

Nach der Erstellung des Tutorial-Scripts war beispielsweise nicht auf Anhieb ersichtlich, wie der „Empfänger“ wieder mit dem „Sender“ kommunizieren kann. Dies war darauf zurückzuführen, dass lange Zeit unklar war, was der Prototyp des Systems am Ende genau darstellen und wie dieser umgesetzt werden sollte.

Das Ziel zum Workshop am 25.04.2019 sollte ein „Prototyp“ sein, der bereits eine simple, realistische Kommunikationskette darstellt. Das sollte wie folgt aussehen:

* Kommunikation einzelner, unabhängiger „Service“-Module mit einem „Hauptmodul“
  + Server-Modul (Hauptmodul)
  + Wetter-Modul (Servicemodul)
  + Telegram-Modul (Servicemodul)
* Keine Kommunikation unter Servicemodulen – ausschließlich mit dem Hauptmodul
* Kommunikation des Hauptmoduls mit den jeweiligen Servicemodulen

Dieser „Prototyp“ sollte also ein Hauptmodul starten, welches auf die Nachrichten von Service-Modulen wartet. Ein Wetter-Modul prüft dafür beispielsweise jede Minute die Temperatur am Veranstaltungsort und sendet eine Nachricht an das Hauptmodul, wenn eine Veränderung vorliegt. Des Weiteren sollte ein Telegram-Modul das Ganze etwas realitätsnaher gestalten. In diesem Fall ermöglicht also ein Telegram-Bot die Kommunikation mit dem Hauptmodul.

# Workshop am 25.04.19 ( Meilenstein 1 )

## Ausgang:

Der geplante „Prototyp“ wurde bis zum 25.04.2019 fertiggestellt. Während des intensiven Beschäftigens mit der Materie ergeben sich durch Überblick und Fokus plötzlich auch einfache Dinge, wie zum Beispiel die Antwort auf die Frage wie ein „Empfänger“ auch wieder eine Nachricht an den „Sender“ leiten kann. Da wundert man sich gleich, wieso solch eine Frage überhaupt aufkommt.

Das Einbinden einer Wetter-API und das Aufsetzen eines Telegram-Bots war durch vorhandenes Vorwissen absolut kein Problem.

## Plan und Fortschritt:

* Anlegen eines Lernportfolios

Das zuvor noch nie ein Lernportfolio angelegt wurde führte zu leichten Schwierigkeiten und zu einem dementsprechenden Ergebnis. Der Fokus lag durch die vorgegebene Aufgabenstellung auf der Dokumentation zum Thema „Reactive Design“.

* Erlangung eines ersten Verständnisses für das Thema "offene Daten"
  + Daten, die von jedem für jegliche Zwecke verwendet werden können

# Selbststudium zum Thema „Reactive Design“ ( Kapitel 4-6 )

## Kapitel 4:

### 4.1 Messages

## Kapitel 5:

123

## Kapitel 6: