ingenieur
wissenschaften
htw saar

Hochschule für
Technik und Wirtschaft
des Saarlandes
University of
Applied Sciences

Facharbeit-Thesis

an der Hochschule für Technik und Wirtschaft des Saarlandes
im Studiengang Praktische Informatik
der Fakultät für Ingenieurwissenschaften

Einführung in Microservices

vorgelegt von
Philipp Tull, Pascal Niedermeyer, Lukas Reichmann

betreut und begutachtet von
Prof. Dr. Markus Esch

Saarbrücken, 23. März 2020

Zusammenfassung

Die vorliegende Arbeit beschäftigt sich mit dem Software Architekturstil „Microservice“. Ziel dieser Arbeit ist es den Architekturstiel Microservice genauer zu betrachten, d.h. es wir geschaut was sich hinter diesem Architekturstiel verbirgt, wie dieser funktioniert bzw. wie dieser Aufgebaut ist und weshalb man diese Architektur verwenden sollte bzw. in welchen Situationen es von Vorteil ist diese Architektur zu verwenden.

Zu Beginn wird eine kurze Einführung geliefert. Diese bietet dem Leser eine Übersicht darüber mit was sich diese Arbeit beschäftigt und worauf der Fokus gelegt wird. Nach der Einführung wird ein überblicke über den Microservice an sich sowie die Funktionsweise gegeben. Anschließend wird die Microservice-Architektur mit anderen Architekturstielen verglichen.
In den beiden darauffolgenden Kapiteln wird sowohl auf die Vorteile als auch auf die Herausforderung bei der Nutzung von Microservices geschaut. Das Kapitel Daten beschäftigt sich mit den Problemen und Herausforderungen mit Daten mit denen Entwickler bei der Arbeit mit Microservices konfrontiert werden. Die Fallstudie zeigt ein kleines Beispiel für einen möglichen Microservice wie er in der Geschäftswelt auftreten kann. Das Fazit beschäftigt sich mit einem Rückblick über die Arbeit, sowie einem Ausblick für die zukünftige Nutzung von Microservices.

Inhaltsverzeichnis

1 Einleitung

In der modernen Welt nutzen immer mehr Menschen immer komplexere Services, die über das Internet bereitgestellt werden. Firmen wie Google, Netflix und Amazon müssen täglich Millionen von Anfragen bearbeiten. Solche großen Systeme stoßen sehr schnell an die Grenzen eines einzigen Servers, da es nicht möglich ist genügend Rechenleistung und Speicher zu verbauen, um eine reibungslose Nutzung zu ermöglichen. Der klassische, monolithische Ansatz, bei dem ein einziges System auf einem einzigen Server läuft, funktioniert also nicht mehr. Stattdessen müssen verteilte Systeme, die die Leistung von vielen Servern (bis hin zu ganzen Rechenzentren) nutzen können, eingesetzt werden. Diese, verteilte Systeme, sind weitaus skalierbarer als klassische monolithische Systeme.

Microservices sind ein Architektur-Pattern, das genutzt werden kann, um die Funktionalität eines Systems in kleine Unter-Systeme aufzuteilen. Mithilfe dieser wird dann das große Gesamtsystem aufgebaut. Die Skalierbarkeit entsteht dadurch, dass jeder Microservice, bei Notwendigkeit, auf einem eigenen Server laufen kann. Im Folgenden wird genauer auf das grundlegende Konzept von Microservices eingegangen, sowie auf die Vorteile und Herausforderungen, die eine Microservice Architektur mit sich bringt. Letztendlich wird in einem Minimal-Fallbeispiel die Architektur gezeigt.

2 Was sind Microservices?

Prinzipiell ist es schwierig Microservices einheitlich zu definieren. Gründe dafür sind zum einen, dass das Prinzip von Microservice-Architekturen zwar klar ist, es aber dennoch unterschiedliche Ansichten und Verständnisse diesbezüglich in der Fachliteratur gibt. Zum anderen gibt es keine einheitliche Architektur von Microservices, bzw. das Grundprinzip ist klar, dies wird allerdings unterschiedlich umgesetzt. Genaueres zu der uneinheitlichen Architektur von Microservices wird in 3 beschrieben.

„Eine Microservicearchitektur besteht aus einer Sammlung kleiner, autonomer Dienste. Jeder Dienst ist eigenständig und sollte eine einzige Geschäftsfunktion implementieren.“[5]

„Die Microservice-Architektur - oder einfach nur Microservices - ist eine eigene Entwicklungsmethode für Softwaresysteme, die sich auf die Entwicklung von Modulen mit nur einer Funktion mit gut definierten Schnittstellen und Operation zu konzentrieren versucht.“[3]

„A microservice, in my mind, is a small application that can be deployed independently, scaled independently, and tested independently and that has a single responsibility. It is a single responsibility in the original sense that it's got a single reason to change and/or a single reason to be replaced. But the other axis is a single responsibility in the sense that it does only one thing and one thing alone and can be easily understood.“[7]

Man sieht, das es keine standardisierte, formalisierte Definition von Microservices gibt. Dennoch gibt es grundlegende charakteristische Merkmale die diese Architektur beschreiben.
Microservices ist eine Methode zur Entwicklung von Softwareanwendungen als Umgebung von unabhängigen, voneinander einsetzbaren, modularen Diensten. Jeder Service soll so konfiguriert sein, das er als alleiniger Dienst ausgeführt werden kann, Daten unabhängig von anderen Diensten verarbeitet und nur über genau definierte Schnittstellen mit anderen Services kommuniziert.[6]

Genauer gesagt Microservice ist ein Ansatz zur Modularisierung von Software. Das heißt große Software-Systeme sollen in kleine Module unterteilt und entwickelt werden.
Dadurch soll es möglich sein Software einfacher zu erstellen, zu verstehen und weiterzuentwickeln. Des Weiteren sollen Microservices möglichst klein, unabhängig und lose gekoppelt sein. Ein Dienst wird von einem kleinen Entwicklerteam geschrieben und verwaltet. Microservices werden unabhängig von einander bereitgestellt, getestet und deployt.

Das Konzept des Microservice verfolgt den gleichen Ansatz wie die Unix-Philosophie[2]

- Ein Programm soll genau eine Aufgabe erledigen. Diese soll aber richtig erledigt werden.
- Programme sollen zusammen arbeiten können.
- Es sollen universelle Schnittstellen genutzt werden.

3 Wie funktionieren Microservices?

Wie schon im vorherigen Abschnitt erwähnt sind Microservices modular aufgebaut und sollen nur über geeignete Schnittstellen miteinander/mit dem Programm kommunizieren. Daher wird jeder Microservice auf einem eigenen Server betrieben, wobei die Services voneinander nichts wissen sollen. Die Funktionsweise bzw. der genaue Ablauf eines Microservices wird in 3.1 beschrieben.

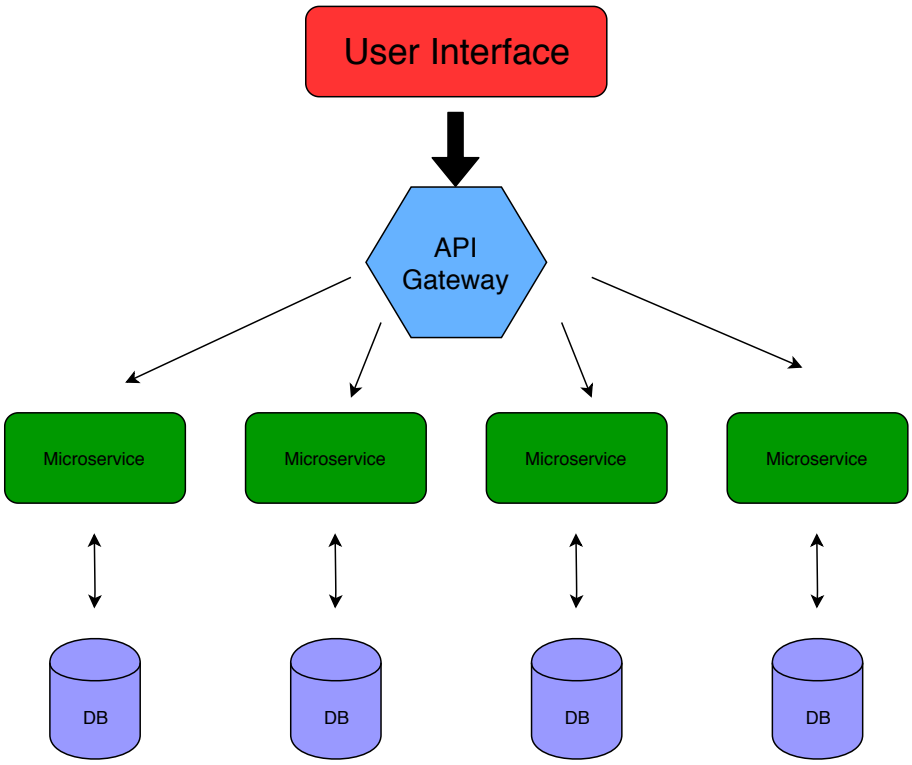


Abbildung 3.1: Microservice

3.1 zeigt einen grundlegenden Aufbau eines Microservices. Wie im vorherigen Abschnitt erwähnt ist es schwierig eine einheitliche Architektur für Microservices zu definieren. Dies wird im folgenden Abschnitt beschrieben. Wie man dennoch zu einer guten Architektur kommt, die auf die Organisation und oder das Programm passt und welche Herausforderungen dabei beachtet und bewältigt werden müssen wird in 6 beschrieben.

Vom Grundgedanken her besitzt jeder Microservice sein eigenes User Interface, durch welches der Nutzer mit der Software kommunizieren soll. Da Microservices allerdings modular aufgebaut sind und im besten Fall unabhängig von einander sind müssen die einzelnen User Interfaces der Microservices noch einmal in einer großen Master IU zusammen gefasst werden. Diese Master UI dient dazu, die einzelnen User Interfaces der Microservices zu bündeln und dem Nutzer so eine UI zur Verfügung zu stellen, sodass es den anschien hat das es eine einzige UI ist. Dieser Architektur-Ansatz wird allerdings eher selten in der Praxis genutzt, da es ohne ausreichender Kommunikation unter den Teams bzw. ohne eines grundlegenden Designs der Oberfläche zu Inkonsistenz und unterschiedlich aussehenden Bausteinen der UI kommen kann. Daher wird bei der Erstellung von Microservices eher auf den Aufbau wie in 3.1 zurückgegriffen, d.h es gibt ein gemeinsames User Interface das von einem Team erstellt und verwaltet wird. Dadurch entsteht zwar im Umkehrschluss eine erhöhte Kommunikation der einzelnen Microservice Teams und des UI Teams, wenn neue Features implementiert werden müssen. Allerdings ist auf der anderen Seite eine einheitliche und runde Oberfläche gewährleistet. Dies ist gerade für den Endnutzer von Vorteil, da er dadurch ein angenehmeres Gefühl bei der Arbeit mit der Software hat.

Wie am Anfang des Abschnitts beschrieben liegt jeder Microservice auf einem eigenen Server und kann nur über eine Schnittstelle nach außen kommunizieren. An dieser Stelle kommt das API-Gateway ins Spiel. Das API-Gateway dient als Einstiegspunk für die Clients. Das bedeutet anstatt den Service direkt aufzurufen, rufen die Clients das API-Gateway auf. Dieses leitet anschließend den Aufruf an den geeigneten Service im Back-End weiter. Dies hat unter anderem die Vorteile, dass[5]:

- Es entkoppelt die Clients von den Diensten. Für Dienste kann eine Versionierung oder Umgestaltung durchgeführt werden, ohne dass sämtliche Clients aktualisiert werden müssen.

- Dienste können nicht web fähige Messagingprotokolle verwenden, z.B. AMQP. (Was ist das)

- Das API-Gateway kann weitere übergreifende Funktionen ausführen, beispielsweise Authentifizierung, Protokollierung, SSL-Terminierung und Lastenausgleich.

Allerdings ist das API-Gateway kein Muss bei der Implementierung einer Microserice-Architektur sondern eine Möglichkeit seine Architektur übersichtlicher zu gestalten. Gerade bei kleineren Software-Systemen kann man auf ein API-Gateway verzichten und die Kommunikation zwischen UI und Microservice über Frameworks wie zum Beispiel REST gestalten. Diese Art der Kommunikation wird auch im Kapitel 8 genutzt, um zu zeigen wie solch eine Architektur in einem System funktionieren kann.

Jeder Microservice hat seine eigenes Back-End in dem die Bussines-Logic enthalten ist. Dort wird genau eine Funktionalität implementiert und es soll möglichst darauf geachtet werden das kein Service von anderen Services abhängig ist geschweige denn Funktionalitäten von anderen Services enthält. Auf dieses Problem wird noch einmal genauer in Kapitel 6 eingegangen, wie damit umzugehen ist.

Prinzipiell hat jeder Microservices seine eigene Datenhaltungsschicht auf die nur er zugreifen kann/soll. Allerdings gibt es auch Ansätze bei denen sich mehrere Microservices eine gemeinsame Datenhaltung teilen bzw. auf mehrere Datenhaltungen zugreifen. Dieser Ansatz wird auch in 8 genutzt. Welche Art der Datenhaltung genutzt wird, sollte Projekt und Architekt abhängig betrachtet werden, da es bei Zugriffen von mehreren Services auf dieselbe Datenhaltung schnell zu Problemen kommen kann. Welche Probleme an dieser Stelle genau auftreten können und wie diese zu lösen sind wird genauer in 7 beschrieben.

3.1 Funktionsweise

Ein Client nutzt die Software welche aus einer Microservice-Architektur besteht. Über das User-Interface kann er mit dem System kommunizieren. Möchte er nun eine Funktionalität der Software nutzen, wird zuerst die Anfrage an das API-Gateway geleitet. Dieses schaut welche Microservices für diese Anfrage zuständig sind und leitet anschließend die Anfrage an den/die entsprechenden Services weiter. Diese bearbeiten die Anfrage und geben das Ergebnis zurück an das API-Gateway sodass das Ergebnis nun auf der UI dargestellt werden kann. Die genaue Kommunikation kann zum Beispiel über eine REST-Schnittstelle laufen welche JSON-Objekte empfängt und versendet.

4 Vergleich mit anderen Architektur-Stilen

In diesem Abschnitt beschäftigen wir uns mit dem Vergleich der Microservice-Architektur gegenüber dem herkömmlichen Architektur-Stil des Monolithen, sowie der ähnlich aufgebauten Architektur der Service Oriented Architecture. Kurz SOA. Dabei wird zuerst darauf geschaut was sich hinter diesen beiden Architektur-Stilen verbirgt und wie diese funktionieren. Anschließend wird ein Vergleich zwischen dem Microservices und der jeweiligen Architektur gezogen.

4.1 Vergleich Monolith

Um zu schauen in wie fern sich die Architektur eines Microservice mit der Architektur eines Monolithen unterscheidet, muss zuerst geklärt werden was unter einem Monolithen verstanden wird.

Ein Monolith ist ein Software-System welches als eine große untrennbare Einheit gestaltet wird. Die monolithische Architektur folgt keiner expliziten Gliederung in Teilsysteme sondern wird als ein ganzes Software-System betrachtet.[1] Dieses Software-System, häufig auch als Legacy-System bezeichnet, besteht im Endeffekt aus einer einzelnen logisch ausführbaren Datei. Diese beinhaltet die gesamte Logik der Software und wird serverseitig verwaltet. Die Software läuft meist in einem einzigen Prozess auf einem Server, auf den von außerhalb zugegriffen werden kann. Da es sich bei dem Monolithen um eine ausführbare Datei handelt, muss bei jeder Änderung an dem System der Aufbau und die Bereitstellung einer neuen Version, für die serverseitige Anwendung gewährleistet sein.[4]
4.1 zeigt den prinzipiellen Aufbau und die Funktionsweise eines Monolithen. Dieser ist in drei Schichten aufgeteilt: dem User Interface, der Backend und der Datenhaltungsschicht. Das User Interface bietet die Möglichkeit der Kommunikation mit dem System. Es empfängt Ereignisse und leitet diese an das Backend weiter. Im Backend befindet sich die gesamte Business-Logik des Software-Systems. Es verarbeitet die empfangenen Nachrichten und greift auf die entsprechend benötigten Daten in der Datenhalten zu. Die Datenhaltungsschicht beinhaltet alle benötigten Daten die das Software-System verarbeitet.

Vergleicht man nun die Architektur des Microservices mit der Architektur des Monolithen gibt es auf der einen Seite Ähnlichkeiten in Bezug auf die einzelnen Komponenten.
Man könnte sagen das jeder Microservice ein sehr kleiner Monolith sei, da jeder Microservice je nach Architektur sein eigenes User Interface besitzt. Ein eigenes Backend sowie eine eigene Datenhaltung hat. Dies steht aber im Widerspruch mit der Bedeutung eines Monolithen, bei dem es sich um eine große Einheit handelt und ein Microservice meist nur aus wenigen Funktionalitäten. Dennoch ist der Vergleich naheliegend.
Im Vergleich zu dem Monolithen liegen die einzelnen Microservices auf unterschiedlichen Servern, wodurch die Kommunikation über spezielle Schnittstellen umgesetzt wird,

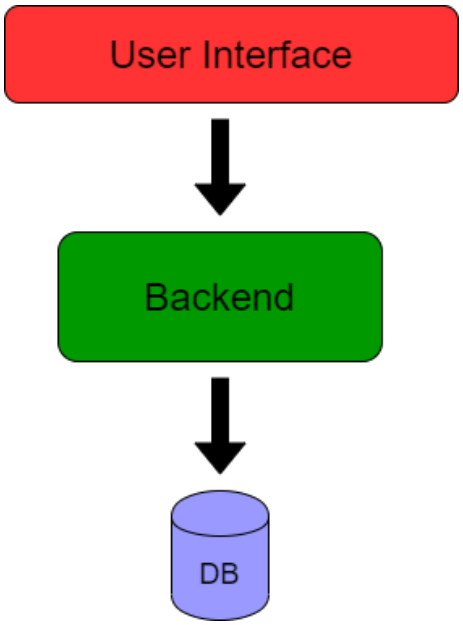


Abbildung 4.1: Monolith

welche über das Internet miteinander kommunizieren. Bei einem Monolithen ist dies nicht der Fall, da die gesamten Funktionalitäten auf einem Server liegen und dadurch direkt aufgerufen werden können. Auch die Datenhaltung unterscheidet sich bei einem Microservice und dem Monolithen, da jeder Microservice meist seine eigene Datenhaltung besitzt und ein Monolith meist nur auf eine oder wenige Datenhaltungen zugreift.
Ein wesentlicher Unterschied zwischen den beiden Architektur-Stilen ist der Umgang mit Abhängigkeiten. Da jeder Microservice auf einem eigenen Server läuft und nur für einen einzige Funktionalität zuständig ist, ist es wichtig das wenige, im Idealfall, keine Abhängigkeiten zwischen Microservices entstehen, da dies zu erheblichen Problemen führen kann. Genaueres zu dieser Problematik in Kapitel 6.
Bei einem Monolithen sollte zwar auch Wert auf eine saubere Architektur gelegt werden und dadurch sehr genau auf Abhängigkeiten geachtet werden. Dennoch ist es an dieser Stelle nicht so gravierend bzw. Abhängigkeiten werden sogar benötigt. Ein weiterer Unterschied ist die Aufteilung der Entwicklerteams. Bei einem Monolithen werden die Teams auf der technischen Ebene aufgeteilt. Das heißt, ein Team ist für die Datenhaltung zuständig, ein anderes Team kümmert sich um die Business-Logik und ein weiteres Team ist für die graphische Oberfläche verantwortlich.
Bei einem Microservices werden die Teams nach Fachlichkeit des Geschäftsprozesses aufgeteilt.
Hier ist ein Team zum Beispiel für die Artikelsuche eines Online-Shops zuständig. Ein weiteres für den Warenkorb und ein drittes Team für den Kaufabschluss. Dadurch ist jedes Team für einen Geschäftsprozess alleine verantwortlich und muss sich sowohl um die Nutzeroberfläche als auch um die Business-Logik und die Datenhaltung kümmern.

4.2 Vergleich SOA

Schaut man sich die beiden Architektur-Stile Microservice und SOA an, scheinen sie auf dem ersten Blick recht identisch zu sein. Bei beiden Ansätzen steht die Aufteilung großer Systeme in kleinere Services im Vordergrund. Daher werden in der Literatur Microservices und SOA häufig als dasselbe bezeichnet, da sich beide Ansätze mit der Aufteilung von Anwendungen in Services beschäftigen. Schaut man sich die beiden Ansätze nun aber genauer an stellt man fest, das sie sich in bestimmten Punkten weit aus mehr unterscheiden als zuvor angenommen.[8]
Für Service Oriented Architecture oder kurz SOA gibt es, wie für Microservices, keine einheitliche Definition. Dennoch gibt es zentrale Bestandteile dieser Architektur, welche große Ähnlichkeiten mit Microservices aufweisen. Unter anderem besteht SOA wie Microservices aus Services welche eine Funktionalität umfassen sollen, sowie das der Service eigenständig genutzt werden kann. Des weiteren sollen die Services über das Internet und über eine geeignete Schnittstelle miteinander kommunizieren. Auch ist die Nutzung von verschiedenen Programmiersprachen sowie Frameworks bei der Entwicklung von SOA möglich.[8][9]

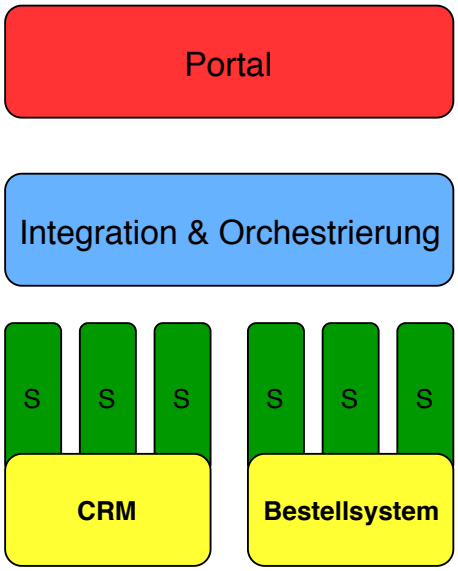


Abbildung 4.2: SOA

4.2 zeigt einen möglichen Architekturansatz und die Funktionsweise eines SOA-Systems. SOA besteht aus einem Portal, einer Integrations- und Orchestrierungsschicht sowie aus mehreren größeren Services welche zu einer bestimmten Fachlichkeit des Geschäftsprozesses gehören. Jeder Service besteht noch einmal aus mehreren kleineren Services.
Das Portal ist vergleichbar bzw. das gleiche wie ein User Interface welches bei den beiden anderen Ansätzen zum Einsatz kommt. Es ist dafür zuständig, Nutzern eine Oberfläche zu bieten, mit der die Services verwendet werden können. Es können auch mehrere Portale koexistieren, jeweils für unterschiedliche Nutzer des Systems. Zum Beispiel für Kunden, den eigenen Support oder auch interne Mitarbeiter wie Verwaltung oder Management. Aber auch für unterschiedliche Plattformen wie Desktop oder mobile Apps. Egal welche Art von Portal, für die Architektur macht dies keinen Unterschied.
In der Integrations- & Orchestrierungsschicht sind die einzelnen Services in einem Verzeichnis registriert. Sie übernehmen die Anfragen der Clients, suchen nach den entsprechenden Services die für dies Anfrage benötigt werden und rufen diese mit den entsprechenden Daten auf.

Die größeren Services sind wie schon erwähnt in Fachlichkeiten des Geschäftsprozesses unterteilt und beinhalten wiederum mehrere kleinere Services, vergleichbar mit Microservices, welche zu Umsetzung einzelner kleinere Geschäftsprozesse dienen. Als Beispiel beinhaltet ein Service das Customer Relationship Management (CRM), welches sich um die Kundenverwaltung kümmert. Die einzelnen Services sind dann für die einzelnen Prozesse des CRM zuständig wie zum Beispiel, Kunde registrieren, Kunde löschen oder Kunde ändern. Jeder größere Service beinhaltet eine eigene Datenhaltung, auf die, die einzelnen Services zugreifen können.

Vergleicht man nun beide Ansätze stellt man fest, das es wie schon erwähnt viele Gemeinsamkeiten gibt, aber auch große Unterscheide, wodurch es schwierig wird Microservices und SOA als das gleiche zu betrachten.
Beide Ansätze streben die Aufteilung von Anwendungen in Services an, sowie die Kommunikation der einzelnen Services über das Netzwerk.
Aber schon bei der Kommunikation bzw. der Integration fangen die Unterschiede an. Bei SOA funktioniert die Integration auch für Orchestrierung. An dieser Stelle wird ein Geschäftsprozess aus den einzelnen Services zusammengesetzt. Die Orchestrierung soll so eine gewisse Flexibilität in das Programm bringen. Bei Microservices hat die Integrationslösung keine eigene Intelligenz. Die einzelnen Services kommunizieren untereinander und müssen sich kennen. Microservices versuchen hingegen die Flexibilität durch einfache und schnelle Änderung sowie Ersetzung von Services zu gewährleisten.
Ein weiterer Unterschied ist die Aufteilung der Teams in Fachlichkeit und Technik. Bei SOA gibt es ein Team welches für die Oberfläche bzw. das Portal zuständig ist. Ein Team welches sich um die Integration & Orchestrierung kümmert. Diese beiden Teams sind auf technische Weise aufgeteilt. Andere Teams kümmern sich um die einzelnen fachlichen Aspekte des Systems, wobei diese für eine Sammlung von Services des jeweiligen Geschäftsprozesses zuständig sind. Durch die Integration und Orchestrierung muss eine unternehmensweite Integrationslösung eingeführt werden, da es gerade bei dem Zugriff auf Daten zu Problemen führen kann. Bei Microservices ist dies leichter zu handhaben, da es keine einheitliche Integrationstechnologie gibt. Die Technologie der Integration und Kommunikation ist auf den Service begrenzt. Gerade bei Datenreplikationen ist dies von Vorteil. Genaueres hierzu in 7.
Auch die Kommunikation zwischen mehreren Teams ist bei SAO schwieriger als bei Microservices. Durch die Aufteilung auf technischer Ebene kann es bei Änderungen eines Service dazuführen, das Anpassungen an anderer Stelle, für die ein anderes Team verantwortlich ist, getätigt werden müssen. So muss zum Beispiel bei der Änderung eines Services auch die Oberfläche entsprechend geändert werden, wodurch das Team Service mit dem Team Portal kommunizieren muss.[8]

5 Warum nutzt man Microservices?

In diesem Abschnitt wird darüber gesprochen, wieso es sinnvoll ist eine Microservice-Architektur zu verwenden und welche spezifischen Vorteile solch ein Architektur Ansatz mit sich bringt. In den vorherigen Abschnitten wurde bereits ein Teil der Vorteile leicht angeschnitten. Nun werden diese detaillierter dargestellt. Da es sich bei den Vorteilen nicht nur um technische Aspekte handelt, werden diese in drei Kategorien unterteilt:

- Technische Vorteile

- Organisatorische Vorteile

- Geschäftliche Vorteile

5.1 Technische Vorteile

Wie schon in den vorherigen Abschnitten erwähnt sind Microservices modular aufgebaut und jeder Service liegt auf einem eigenen Server. Dadurch entsteht eine verteilte Kommunikationsstruktur. Diese ermöglicht es einem Service, mit anderen Services zu kommunizieren bzw. diese aufzurufen. Diese Kommunikationsinfrastruktur muss dementsprechende Möglichkeiten enthalten um die Kommunikation zwischen den Services zu ermöglichen. Ein Entwickler muss diese Infrastruktur genau beschreiben. Dadurch ist es schwieriger das Abhängigkeiten zwischen zwei oder mehreren Services entstehen, da der Entwickler diese explizit einbauen muss, um die Kommunikation zu gewährleisten. Falls sich doch einmal solch eine Abhängigkeit ungewollt einschleichen sollte, gibt es spezielle Architekturmanagement-Werkzeuge um diese zu finden. Gerade bei Microservices ist dies leicht zu korrigieren, wenn man die Abhängigkeit frühzeitig entdeckt, da der Service schnell abgeändert werden kann oder komplett neu geschrieben werden kann. Genaueres zu dem Umgang mit Abhängigkeiten wird im Kapitel **??** beschrieben.[8]

Ein weiterer Vorteil ist wie gerade beschrieben, die mögliche Ersetzung eines Microservices. Gerade der Umgang mit alten Software-Systemen ist eine große Herausforderung. Oftmals ist die Codequalität recht schlecht, da die Software über Jahre hinweg von unterschiedlichen Entwicklern weiter entwickelt wurde und an vielen Stellen Hilfen eingebaut wurden, um temporäre Probleme zu lösen, die der Entwickler später wieder beheben wollte, dies aber nie tat. Diese Software weiter zu entwickeln oder gar neu zu schreiben ist ein fast unmögliches Unterfangen, da an vielen Stellen gar nicht mehr bekannt ist was einzelne Funktionen oder Codeabschnitte genau machen und der Entwickler der sie geschrieben hat, nicht mehr in dem Unternehmen tätig ist. Je größer die Software ist, desto größer ist der Aufwand. Beinhaltet die Software zusätzlich noch wichtige Geschäftsprozesse ist es praktisch unmöglich die Software noch zu ändern, da ein Ausfall der Software einen erheblichen finanziellen Schaden bedeuten kann.
Microservices bieten an dieser Stelle den Vorteil das die Entwickler zum einen nicht an bestimmte Technologien gebunden sind, sondern können beliebige andere Technologien nutzen um das Problem zu lösen. Dadurch das ein Microservice im Idealfall auf der

fachlichen Ebene eigenständig agiert, muss nicht genau bekannt sein wie die einzelnen Funktionen genau funktionieren. Dem Entwickler muss nur bekannt sein welche fachlichen Funktionalitäten der Microservices haben soll und welche Geschäftsprozesse er beinhaltet. Danach kann er diesen einfach durch einen neuen Microservice ersetzten.[8]

Im Umgang mit Legacy-Systemen bilden Microservices einen weiteren Vorteil. Wie bei dem Vorteil der Ersetzung eines Microservices beschrieben, sind Legacy-Systeme oft recht unübersichtlich je größer sie werden. Mit Hilfe von Microservices können einzelne Codeabschnitte oder Funktionalitäten der Software einfach an einen Microservice ausgelagert werden. Hierzu muss einfach eine Schnittstelle an die Stelle der Funktionalität gebaut werden, welche ersetzt werden soll. Diese leitet die Anfrage anschließend an den Microservice weiter der diese dann verarbeitet. Anschließend kann er das Ergebnis wieder an das Legacy-System zurück senden, welches wiederum mit dem Ergebnis weiter arbeiten kann. Dadurch können kritische, schwer zu ändernde Codeabschnitte gezielt umgangen und ausgelagert werden, ohne das es zu Problemen mit dem System kommt.[8]

Continous Delivery brint Software durch einen einfachen reproduzierbaren Prozess regelmäßig in Produktion. Dazu dient eine Continuous-Delivery-Pipeline.

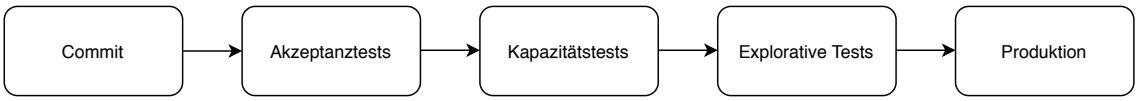


Abbildung 5.1: Continuous-Delivery-Pipeline

- In der Commit-Phase wird die Software kompiliert, die Unit-Tests werden durchgeführt und es wird gegebenenfalls eine statische Code-Analyse durchgeführt.

- Die automatisierten Akzeptanztests in der nächsten Phase stellen sicher, dass die Software fachlich korrekt ist, sodass sie vom Kunden akzeptiert wird.

- Kapazitätstests überprüfen, ob die Software performant genug ist, um die erwartete Anzahl Nutzer zu unterstützen. Auch diese Tests sind automatisiert.

- Die explorativen Tests hingegen sind manuell und dienen dazu, bestimmte Bereiche des Systems zu testen. Dazu können neue Features zählen oder bestimmte Aspekte wie die Sicherheit des Systems.

- Zum Schluss wird die Software in Produktion gebracht. Auch dieser Prozess ist idealerweise automatisiert.

Die Software durchläuft jede einzelne Phase der CDP. Nach jeder erfolgreich abgeschlossenen Phase wird sie in der nächsten Phase weiter getestet. An dieser Stelle kann es passieren, dass die Software eine Phase erfolgreich abschließt, dann aber in der nächsten Phase nicht akzeptiert wird, weil zum Beispiel die Laufzeit zu lange dauert.
Gerade hier liegt der Vorteil bei einer Microservice-Architektur. Microservices sind eigenständige Deployment-Einheiten. Daher können sie unabhängig von anderen Services getestet und in Produktion gebracht werden. Dies hat erhebliche Auswirkungen auf die CDP im Vergleich zum Testen von monolithischen Strukturen. Der Durchlauf durch die einzelnen Tests ist wesentlich schneller, da nur ein kleiner Microservices in Produktion gebracht wird. Dadurch erhält man schnell Feedback, wodurch der Fehler bzw. das Problem behoben werden kann und sofort wieder getestet wird. Dadurch wird eine Menge zeit gespart. Denn wenn der Entwickler erst nach mehreren Tagen oder Wochen erfährt das sein Code fehlerhaft ist, ist es schwieriger sich wieder in den Code einzuarbeiten und den Fehler zu beheben.
Auch ist der zu testende Aufwand wesentlich bei einem Microservice geringer als bei einem Monolithen. Bei einem Monolithen muss der gesamte Code bei einer Änderung wieder deployed und anschließen ausgeliefert werden. Bei einem Microservice ist es nur der Microservice selbst, ohne das andere Teiler der Software getestet und ausgeliefert werden müssen.[8]

Auch in Bezug auf Skalierbarkeit haben Microservices Vorteile. Dadurch das jeder Service auf einem eigenen Server betrieben wird, können mehrere Instanzen des selben Microservice betrieben werden. Das heißt es gibt mehrere Services eines Microservices auf verschieden Servern. Dadurch ist eine gute Skalierbarkeit gewährleistet, da die Last auf mehrere Server verteilt werden kann. Auch ist es dadurch möglich Microservices auf unterschiedlich schnellen Servern laufen zu lassen. So kann jeder Microservice seine eigene Skalierung beinhalten. Des weiteren können so Microservices an verschiedenen Stellen im Netzwerk betrieben werden, um so einen geographische Skalierung zu erzeugen.
An dieser Stelle kommt auch die Robustheit eines Microservices in Spiel. Prinzipiell sollten Microservices anfälliger für Ausfälle sein, da ein Ausfall nicht nur auf der Hardware-Ebene passieren kann, sondern auch ein Ausfall auf der Netzwerk-Ebene. Um trotzdem eine hohe Verfügbarkeit zu gewährleisten muss eine Art Firewall zwischen den Microservices gebildet werden. Ein Ausfall eines Services darf sich nicht auf andere Services ausbreiten und diese ebenfalls zu einem Ausfall bringen. Hierzu kann zum einen bei einem Ausfall ein Microservice mit einem Default-Wert weiter arbeiten oder an einen anderweitig reduzierten Services weiter geleitet werden. Zum anderen kann wie bei Skalierbarkeit beschrieben, bei einem Ausfall eines Microservices einfach ein anderer Service eingesetzt werden, der die gleicher Funktionalität besitzt. Dies wird solange betrieben bis der ursprüngliche Services wieder verfügbar ist.[8]

Wie schon in vorherigen Abschnitten beschrieben herrscht bei Microservices eine Technologiefreiheit. Jeder Services kann in einer anderen Programmiersprache geschrieben sein, mit unterschiedlichen Frameworks zusammen arbeiten oder aber auch an bestehende Sprachen angepasst werden. Dies bietet zum einen den Vorteil, das Funktionalitäten mit bestimmten Anforderungen in dafür geeigneten Sprachen geschrieben werden können. Nehmen wir an es wird viel Wert auf die Laufzeit gelegt, so kann der Service in C oder C++ geschrieben werden. Des weiteren ist es so möglich, leichter Microservices zu ersetzen, da man den Service nicht in der ursprünglichen Sprache ersetzten muss, sondern eine

beliebige wählen kann. Falls zum Beispiel ein Service in Python geschrieben wurde und der Entwickler die Firma verlassen hat, kann der Service in einer anderen Sprache neu geschrieben werden, falls sich niemand mit Python auskennen sollte. So kann einiges an Einarbeitungszeit gespart werden. Auch die Arbeitsmoral und Motivation kann dadurch gesteigert werden. Durch die technologische Wahlfreiheit können neue Frameworks oder Sprachen getestet werden, ob diese für das Unternehmen lukrativ sind bzw. ob man diese auch in Zukunft für andere Projekte nutzen kann.
Normalerweise würde in einem Unternehmen die Zeit fehlen, das sich Entwickler in neue Technologien einarbeiten können, um zuschauen welchen Mehrwert diese für die Firma haben. Des weiteren sorgt die Möglichkeit der Wahlfreiheit für eine gewisse Abwechslung für die Entwickler. Diese müssen nicht Tag für Tag mit der selben Sprachen etc. arbeiten, sondern können sich auch mit neuen Technologien beschäftigen. Dadurch kann zumindest teilweise ein gewisser Trott vermieden werden und die Mitarbeiter neue Impulse setzten.[5]

5.2 Organisatorische Vorteile

Durch die technische Unabhängigkeit kann ein Team, welches für einen Microservice voll zuständig ist, die Unabhängigkeit komplett ausnutzen. Dadurch wird vor allem die Selbstständigkeit des Teams gestärkt. Des weiteren müssen die einzelnen Teams weniger koordiniert werden, da es zum Beispiel nicht wichtig ist das unterschiedliche Bibliotheken oder unterschiedliche Versionen von Bibliotheken genutzt werden. Die einzelnen Teams sind dafür verantwortlich welche Architektur welche Sprache und weiches Framework genutzt wird. Daraus folgt aber auch, dass das Team die vollen Konsequenzen tragen muss, falls etwas schief läuft. Dies steigert zum einen die Selbstorganisation und zum anderen werden die Teammitglieder zusätzlich motiviert fokussiert zu arbeiten und ihre Entscheidungen zu reflektieren.[8]

5.3 Geschäftliche Vorteile

Die bereits erwähnten Vorteile aus der organisatorischen Sicht führen zu geschäftlichen Vorteilen. Das Risiko der Projekte sinkt, da jedes Team eine hohe Eigenverantwortlichkeit trägt und die Koordinierung zwischen den Teams wird weniger, sodass die Teams effizienter arbeiten können. Durch die Aufteilung in Mikroservices können die Teams unabhängig von einander an den Services Arbeiten, was eine parallele Arbeit ermöglicht. Teams müssen nicht auf Ergebnisse anderer Teams warten , um selbst an ihren Services weiter arbeiten zu können. Dadurch kann auch eine Skalierung auf der Ebene der Entwicklung erzeugt werden. Auch müssen die einzelnen Teams viel weniger untereinander Kommunizieren, was dazu führt das sie schneller Ergebnisse erzielen und ihr Projekt abschließen. Folgen für das Unternehmen aus geschäftlicher Sicht, sind zum einen schnellere Ergebnisse was wiederum zu einer Gewinnmaximierung führt. Zum anderen weniger Verwaltungsaufwand was auch wieder zu einer Gewinnmaximierung führt.[8]

6 Herausforderungen

In dem folgendem Kapitel werden die Herausforderungen und Schwierigkeiten von Miroservices beschrieben. Dabei werden die Anforderungen an das Netzwerk beschrieben, welches mit den Lasten von Verteilten Systemen zurecht kommen muss (6.1). Darauf folgend wird auf die vermutlich größte Herausforderung eingegangen, der Architektur. Eine schlechte Architektur kann den Vorteil von Microservices zunichte machen und muss daher fortlaufend angepasst werden (6.2). Danach wird die benötigt Infrastruktur beschrieben, die bei Microservices um ein vielfaches größer ist und mehr Aufmerksamkeit bedarf als bei einer mololithischen Entwicklung (6.3).

6.1 Netzwerk

Da Microservices verteilte Systeme sind, werden Aufrufe untereinander über das Netzwerk verschickt. Damit verbunden sind Latenz und Antwortzeiten die mit steigender Netzwerkauslastung exponentiell steigen.

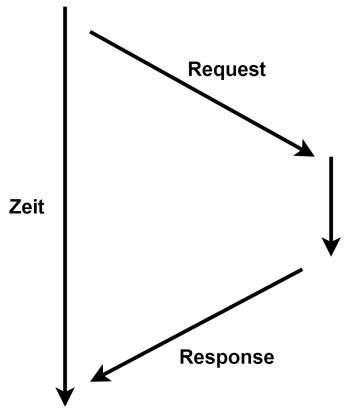


Abbildung 6.1: Latenz bei einem Netzwerkaufruf

In Fig. 6.1 ist ein typischer Aufruf über ein Netzwerk zu sehen. Während die Anfrage über das Netzwerk verschickt, von einem anderen Service bearbeitet und die Antwort wieder zurück geschickt wird muss der ursprüngliche Service warten. Selbst wenn die Latenz einer Nachricht nur 0,5 ms beträgt, könnte ein Prozessor mit 3 GHz in dieser Zeit 1,5 Millionen Operationen durchführen. Daher sollte auf die Entfernung der einzelnen Services geachtet werden und diese so nah wie möglich beieinander zu halten. Auch sollte darauf geachtet werden, dass das Netzwerk nicht überlastet wird. Subnetze bieten sich bei dieser Problemstellung an.

Eine hohe Kommunikation zwischen Microservices spricht jedoch auch für eine schlechte Skalierung der Services und für zu vielen Abhängigkeiten. Solche Abhängigkeiten sollen jedoch vermeiden werden, da dadurch der Vorteil von Microservices verloren geht -

das unabhängige Deployment. Mehr zum Thema Architektur im folgendem Unterkapitel (6.2).

Ein weiteres Problem welches bei verteilten Systemen auftreten kann, ist der Ausfall eines Teilsystems. Wenn dies geschieht, müssen unterschiedliche Maßnahmen getroffen werden, damit nicht das gesamte System zum Erliegen kommt. So muss es Pläne geben, was passiert, wenn eine Request keine Antwort bekommt. Dafür kann es verschiedene Gründe geben. Es kann beispielsweise daran liegen, dass die Nachricht im Netzwerk nicht richtig weitergeleitet wurde. Eine weitere wahrscheinliche Möglichkeit ist ein abgestürzter Gegenspieler. Im ersten Fall kann durch eine erneut gesendete Anfrage das Problem gelöst werden, im zweiten Fall würde dies jedoch ohne weiteren Maßnahmen zu einer Endlosschleife führen. Dies gilt es zu erkennen und geeignete Fehlerfälle mitzubeachten. Währenddessen muss der abgestürzte Microservice neu gestartet werden, am Besten wäre es, wenn mehrere Instanzen des Services aktiv wären und im Falle eines Absturzes diese kurzzeitig die zusätzlichen Anfragen bearbeiten könnten. Gleichzeitig wird im Hintergrund der Service automatisch neu gestartet und kann danach weiterarbeiten.
Im Falle von großen Lastwechseln kann dieser Mechanismus auch dazu genutzt werden bei Bedarf neue Instanzen eines Services zu starten, damit keine zu großen Warteschlangen entstehen und die Antwortzeiten gering bleiben. Ein Aufteilen der Requests auf die einzelnen Instanzen kann dabei dann mittels Service Discovery und Load Balancing geschehen.

6.2 Architektur

Bei Microservices ist es wichtig, dass die fachliche Struktur mit der Organisationsstruktur übereinstimmt (Gesetz von Conway). Die unterschiedlichen fachlichen Funktionen werden dabei in Microservices aufgeteilt. Ein Team der Organisation ist dabei für einen oder mehrere Microservices zuständig. Die Microservices anderer Teams werden dagegen als Black-Box gesehen, von denen nur die Schnittstelle bekannt ist. Daher können die Microservices auch in unterschiedlichen Technologien implementiert sein. Die einzige Gemeinsamkeit, die alle Services haben müssen, ist eine einheitliche Schnittstelle. Eine valide Option hierfür wäre eine REST Schnittstelle.
Durch die verschiedenen Technologien ist es schwer einen Gesamtüberblick über das System zu behalten. Falls dies jedoch benötigt wird, ist es ratsam einen Technologiestack vorzuschreiben. Dies kann auch partiell durchgeführt werden, falls beispielsweise einheitlich geloggt werden soll.

Änderungen Durch die Aufteilung in kleine Microservices sind Änderungen und Refactorings innerhalb eines Microservices sehr leicht umzusetzen. Bei größeren Änderungen oder Fehlern ist es auch leicht den gesamten Microservice zu ersetzen und neu zu implementieren.
Falls neue Anforderungen Änderungen an mehreren Microservices erfordern, ist dies sehr viel mehr Arbeit. Kommt dies öfters vor, kann dies auch für eine schlechte Architektur sprechen, die überarbeitet werden sollte. Bei Microservice-übergreifenden Änderungen bei denen mehrere Teams beteiligt sind, müssen diese koordiniert werden und sich miteinander abstimmen. Auch können die Services in unterschiedlichen Technologien entwickelt worden sein oder Bibliotheken in unterschiedlichen Versionen nutzen. Dies alles macht den Vorteil von Microservices zunichte und ist sehr Aufwändig. Daher ist das initiale Erstellen der Architektur ein essentieller Punkt, der über den Erfolg des Projektes maßgeblich mitentscheidet.

Anpassen der Architektur Wie bereits gezeigt, ist die Architektur ein essentieller Part eines Microservice Systems. Durch eine gute Architektur können Anforderungen nachhaltig schnell umgesetzt werden (Micorservice-intern) und es können die Technologien genutzt werden, die für den Anwendungsfall optimal sind.
Jedoch ändert sich die optimale Architektur mit neuen Anforderungen. Wenn die Architektur daraufhin (agil) angepasst wird, ergibt sich daraus auch kein Problem, jedoch wenn man an der alten Architektur festhält.
Gründe für eine Anpassung der Architektur können folgende sein:

- Ein Microservice ist zu groß und muss aufgeteilt werden. Zeichen hierfür ist beispielsweise eine schlechte Verständlichkeit oder auch eine Größe, mit der nicht einmal ein ganzes Team zurecht kommt. Ebenfalls kann es vorkommen dass ein Micorservice mehrere Themengebiete umfasst.

- Ein Microservice bearbeitet Funktionen, die in einem anderen Microservice besser aufgehoben wären. Erkennbar sind solche Funktionen durch viel Kommunikation zwischen den beiden Services. Ebenfalls können Bereiche in einem Microservice sehr wenig miteinander zu tun haben. Diese können dann aus Gründen der Übersichtlichkeit in neue Microservices getrennt werden.

- Wenn eine Funktion von mehreren Microservices genutzt werden soll, sollte eine Architekturänderung ebenfalls in Betracht gezogen werden.

Um diese Probleme zu lösen gibt es verschiedene Lösungsansätze:

Gemeinsame Bibliotheken Wenn Code gemeinsam genutzt werden soll, so kann dieser in Bibliotheken ausgelagert werden. Dies setzt voraus, dass die darauf zugreifenden Microservices in der selben Technologie entwickelt werden.
Dies bedeutet jedoch auch, dass die Microservices von einander abhängig werden und die Arbeit an der Bibliothek koordiniert werden muss.
Durch 3rd. Party Bibliotheken können so ebenfalls Probleme entstehen. So kann in manchen Laufzeitumgebungen jeweils nur eine Version einer Bibliothek genutzt werden. Das heißt, dass wenn in der neuen Bibliothek die externe Bibliothek XY v2.0 benötigt wird, muss der Code des Microservices, wenn er die Bibliothek XY benötigt, diese ebenfalls in der Version 2.0 nutzen.
Wegen diesen Problemen wird eine Wiederverwendung von Code in Microservices nicht forciert.

Code übertragen Ein anderer Weg eine Funktion in einem Microservice verfügbar zu machen, ist die Codeübertragung. Der Ansatz ist wie bei einer gemeinsamen Bibliothek, nur dass keine Abhängigkeiten entstehen. Dadurch kann eine lose Kopplung zwischen zwei Abhängigen und stark kommunizierenden Microservices wiederhergestellt werden. Dabei spielt es keine Rolle ob der Code im ursprünglichem Microservice bestehen bleibt oder nicht, wodurch jedoch Redundanzen entstehen. Dies bedeutet dass Bug Fixes an mehreren Stellen vorgenommen werden müssen, dagegen können sich die Microservices/Funktionen unabhängig voneinander in unterschiedliche Richtungen weiterentwickeln.
Es muss darauf geachtet werden, dass durch das Hinzufügen von Funktionen der Microservice nicht zu groß wird und zu einem Monolithen mutiert.

Gemeinsamer Service Anstatt den Code in eine Bibliothek auszulagern, kann dieser auch in einen neuen Microservice überführt werden. Dadurch werden auch verschiedenen Technologien unterstützt. Mit dieser Methode werden die Micorservices

auch klein gehalten, was vorteilhaft für die Übersichtlichkeit und Verständlichkeit des Systems ist. Diese neuen Microservices sind dann reine Backend-Services, da diese keine Benutzeroberfläche besitzen.
Dieses Verfahren kann auch gut dazu genutzt werden wenn ein Microservice zu groß wird. Durch Auslagern von Funktionen kann die Wartbarkeit wieder erhöht werden.

Neu schreiben Mit der Zeit kann es vorkommen dass ein Microservice schwer wartbar wird. Dies kann beispielsweise durch einen "historisch gewachsenen" Code geschehen, oder auch durch die Auswahl einer nicht praktikablen Technologie. Gerade bei solchen Problemen haben Microservices ihren Vorteil.
Durch ihre kleine Größe kann der gesamte Service ohne größeren Aufwand neu entwickelt werden. Dabei können neue Erkenntnisse über die Domäne direkt in die Neuimplementierung einfließen. Ein Wechseln der Technologie ist ebenfalls kein Problem, solang die Schnittstellen gleich bleiben.

6.3 Infrastruktur

Auch bei der Infrastruktur gibt es bei Microservices Hürden und Herausforderungen. So werden viel mehr Server oder Virtuelle Maschinen benötigt als bei einem Monolithen. Die verschiedenen Services müssen aufgrund ihrer Unabhängigkeit auf einzelnen Servern/VM's liegen. Dies kann bspw. an verschiedenen Versionen von Bibliotheken liegen oder auch an dem unabhängigen Deployment der Services.
Einer der großen Vorteile von Micorservices ist das lastabhängige Starten zusätzlicher Instanzen. Dies muss automatisch und auf einer neuen VM geschehen. Um jedoch überhaupt erstmal festzustellen, wie groß die Last ist, muss ein geeignetes Monitoring auf allen Services aktiv sein. Aufgrund der verschiedenen Technologien kann dies zu einem großen Problem werden.
Ebenfalls sollte ein Monitoring für die Problemfindung vorhanden sein. Dies muss auf die einzelnen Services angepasst sein, damit Schwachstellen erkannt und optimiert werden können.
Auch bei der Entwicklung gibt es im Vergleich zu einem Monolithen Nachteile. So muss für jeden einzelnen Microservice eine Versionskontrolle eingerichtet werden. Eine Build Pipeline pro Micorservice ist ebenfalls empfehlenswert. Diese Entwicklungsanforderungen bedeuten für das Unternehmen zusätzliche Infrastruktur wie auch erhöhte Administration.

7 Daten

Die Daten eines Systems, das eine Microservice-Architektur nutzt sind sehr stark verteilt. Dies hat den Grund, das jeder Microservice eine eigene Datenbank besitzt. Würde, wie bei einem Monolithen, alle Daten in einer einzigen Datenbank gespeichert werden, dann würde die Skalierbarkeit sehr stark darunter leiden. Schließlich war einer der Gründe für Microservices, dass jeder der Microservices auf einem eigenen Server laufen könnte, um das System skalieren zu können. Würden alle Daten in einer einzigen Datenbank gespeichert werden (die nur auf einem einzigen Server läuft), würde hier erneut ein Flaschenhals entstehen, da dieser Datenbank-Server nicht im selben Ausmaß skalieren kann, wie es die Microservices können.
Eins der größten Probleme mit den Daten bei einem Microservice-System ist, das durch die Verteilung der Daten keine Transaktionen mehr möglich sind sondern die Änderungen an den Daten anderen Microservices mitgeteilt werden müssen (beispielsweise muss das Anlegen eines Nutzers an andere Microservices mitgeteilt werden).

7.1 Synchrone Persistenz

Synchrone Persistenz ist ein Ansatz um das oben genannte Problem der Synchronisierung zu Lösen. Bei der Synchronen Persistenz wird jede Änderung an den Daten direkt an alle anderen Microservice, die von diesen Daten abhängen, weitergegeben. Wenn beispielsweise ein Microservice B von den Daten des Microservice A abhängt, dann würde B einen Endpunkt besitzen, den A aufruft, sobald sich Daten ändern. A wartet dann auf die Antwort von B, um sicherzustellen, dass die Daten auch tatsächlich in B aktualisiert wurden. Wenn mehrere Microservices von den Daten abhängen, dann muss jeder dieser abhängenden Services einzeln Benachrichtigt werden. Es muss beachtet werden, dass Transaktionen (nach ACID-Prinzip) dennoch nicht möglich sind. Beispielsweise sind Rollbacks nur schwer umsetzbar wenn fünf Microservices Benachrichtigt werden, dies aber nach dem dritten scheitert (es müsste also ein Rollback bei den ersten beiden durchgeführt werden). Desweiteren existieren die folgenden Probleme:

Hohe Kupplung Bei der Synchronen Persistenz existiert eine 1:N Beziehung zwischen den Microservices. Das bedeutet, dass im schlechtesten Fall (wenn jeder Microservice immer jeden anderen Microservice benachrichtigen muss) $\frac{n(n-1)}{2}$ Beziehungen existieren. Zudem steigt offensichtlich deren Anzahl sehr stark, wenn neue Microservices zum System hinzugefügt werden. Dies erhöht die Komplexität des Systems enorm und macht es für Entwickler sehr schwer, den Überblick zu behalten.

Service-Ausfälle Wenn ein einzelner Microservice ausfällt kann dieser nicht mehr Benachrichtigt werden. Hierdurch müsste entweder jeder Microservice speichern, welche Microservices noch Benachrichtigt werden muss (wobei die logik hierfür sehr komplex ist, und den Rahmen eines einzelnen Microservices sprengt), oder der gesamte Befehl scheitert, wodurch allerdings ein Rollback ausgeführt werden müsste, was, wie bereits weiter

oben beschrieben, problematisch ist. Zudem würde hierdurch das gesamte System deutlich anfälliger gegenüber Ausfällen werden, da in einem Benachrichtigungs-Baum (d.h. ein Microservice Benachrichtigt andere Microservices, die wiederum andere Microservices benachrichtigen etc.) ein einziger Ausfall eines Microservices den gesamten Baum zum Ausfallen bringt.

Netzwerk-Overhead Synchrone Persistenz ist nicht sehr performant, da immer gewartet wird, bis die benachrichtigen Services eine Antwort liefern (da sonst Fehlerbehandlung sehr schwierig wird, da nicht bekannt ist, ob die Services auch tatsächlich benachrichtigt wurden). Wenn hier allerdings ein Baum mit mehreren Ebenen existiert, dann muss der Wurzel-Service warten, bis alle Blatt Services geantwortet haben. Dies kann, je nach Netzwerk-Verbindung oder der Komplexität der Berechnungen bei einer Benachrichtigung, sehr lange dauern.

7.2 Ereignis gesteuerte asynchrone Persistenz

Ereignis gesteuerte asynchrone Persistenz (im folgenden nurnoch Event-Persistenz genannt) ist eine alternative zur Synchronen Persistenz. Der Name beschreibt das Prinzip bereits sehr gut. Es wird ein Ereignis-Basiertes System eingesetzt und, im Gegensatz zur synchronen Persistenz, werden Änderungen nicht sofort, sondern asynchron übernommen. Das bedeutet, dass ein benachrichtigender Microservice nicht auf die Antwort aller benachrichtigten Microservices warten muss, sondern diese möglicherweise erst zu einem späteren Zeitpunkt benachrichtigt werden (hierbei handelt es sich allerdings in der Regel um kleine Zeitverzägerungen, die nicht von einem Menschen bemerkbar ist). Der Ablauf bei der Event-Persistenz zum Benachrichtigen von anderen Microservices läuft ab, indem der benachrichtigende Microservice, auf einem bestimmten Kanal den Event-Manager benachrichtig, dass es neue Daten gibt. Hierbei existiert ein Kanal, jeweils einer pro möglicher Veränderung (beispielsweise gibt es einen Kanal für das Erstellen und einen Kanal für das Löschen eines Nutzers). Andere Microservices, die benachrichtigt werden müssen, tragen sich beim Event-Manager für einen bestimmten Kanal ein und werden dann automatisch von diesem benachrichtigt, wenn ein neues Event in dem jeweiligen Kanal ausgelöst wurde. Hierdurch wird die hohe Kupplung der Synchronen Persistenz verhindert, da jeder Microservice immer nur direkt mit dem Event-Manager kommuniziert und nicht von anderen Microservices abhängig ist. Zudem merkt der Event-Manager welche Services bereits benachrichtigt wurden und welche nicht. Wenn also ein einziger Service ausfällt (und demnach nicht benachrichtigt werden konnte), wird dieser lediglich erst später benachrichtigt. Hierdurch bringt ein einziger Ausfall nicht das gesamte System zum Einsturz, wie es bei der Synchronen Persistenz der Fall wäre. Wenn ein Service ausfällt, dann verzögert sich lediglich der Zeitpunkt bei dem der jeweilige Microservice die Datenänderung übernimmt, bis der Service wieder funktionsfähig ist (solche Verzögerungen sind aber ohnehin von vornherein ein Teil des Systems, da es asynchron ist). Ein großes Problem der Event-Persistenz ist allerdings, dass ein Ausfall des Event-Managers das gesamte System zum erliegen bringt, da dann keine Benachrichtigungen mehr gesendet werden können. Hier muss also dafür gesorgt werden, dass die Ausfallrate des Event-Managers möglichst gering ist. Zudem sind Transaktionen, aus den gleichen Gründen wie bereits bei der Synchronen Persistenz, nach wie vor nicht möglich.

7.3 2-Phase-Commit

Das 2-Phase-Commit (2PC) Protokoll ist ein Protokoll, das Transaktionen in einem Verteilten System ausführen kann. Es hat allerdings einige Probleme. Zum einen nutzt es einen Zentralen Manager, der die Transaktionen orchestriert[1]. Zum anderen ist das Protokoll sehr aufwändig, und hat die Laufzeit $O(n^2)$. Die Geschwindigkeit wird zudem weiterhin von Sperren auf Ressourcen eingeschränkt. Generell wird 2PC nicht bei Microservices verwendet.

7.4 Saga

Sagas[2] sind eine Möglichkeit, mit der sichergestellt werden kann, dass Daten aktualisiert werden, und mit der, bei einem Fehler, diese Änderungen wieder Rückgängig gemacht werden. Das Prinzip von Sagas ist das folgende: Jeder Microservice hat für jedes Datum, dass er aktualisieren kann, jeweils eine Funktion T und C (Transaction beziehungsweise Correction). T aktualisiert ein bestimmtes Datum in der lokalen Datenbank des Services. C macht das genaue Gegenteil, indem es eine Aktualisierung rückgängig macht. Für jedes zu aktualisierende Datum existiert eine Saga, die eine Liste von Microservices darstellt. Zum Aktualisieren eines Datums führt der erste Microservice in der Saga seine T-Funktion aus. Dannach wird der zweite Service benachrichtigt, der wiederrum seine T-Funktion ausführt. Dies wird solange fortgeführt, bis alle Services in der Saga abgearbeitet wurden. Falls während der Durchführung einer T-Funktion ein Fehler auftritt kommt die C-Funktion zum tragen. Denn anstatt, dass der nächste Service in der Sage benachrichtigt wird, wird der vorangehende benachrichtigt. Dieser wird dann seinen vorangegangenen aufrufen, der dann die C-Funktion aufruft und die erneut die gleiche Nachricht an seinen Vorgänger weitergibt. Hierdurch werden sämtliche Daten, die bis zu diesem Zeitpunkt aktualisiert wurden, wieder durch die C-Funktionen rückgängig gemacht. Sagas nutzen ebenfalls einen Event-Manager, um die Benachrichtigungen zwischen den Services in einer Saga zu verwalten. Hierdurch wird das gesamte System asynchron, und Ausfälle einzelner System sind weniger schwerwiegend. Die Reihenfolge, in der die Microservices in einer Saga auftreten, muss spezifisch für jede Saga festgelegt werden, hier gibt es keine einheitliche Regel. Ein Nachteil von Sagas ist, dass sie nicht so lose gekuppelt sind wie es bei der Event-Persistenz der Fall ist. Schließlich müssen bei einer Saga immer alle Services, die Teil dieser Saga sind, gespeichert werden. Vor allem müssen diese im vornherein gut geplant sein, um festzulegen, wie die Reihenfolge der Services innerhalb der Saga auszusehen haben.

[1] Wobei dieses Problem bei der Event-Persistenz ebenfalls existiert.
[2] Saga ist keine Abkürzung, sondern schlicht das normale Wort „Saga“.

8 Fallstudie

Szenario Für unseren Online-Shop soll eine neue Benutzerverwaltung entwickelt werden, nachdem die alte nicht mehr Wartbar ist. Dazu zählt das Anmelden und neu Registrieren von Nutzern wie auch das Ausgeben der Benutzerdaten zu einem Nutzer.

Gründe für einen Microservice Da man aus den Problemen der vorherigen Benutzerverwaltung gelernt hat, soll die neue Verwaltung auch langfristig Wartbar bleiben.
Auch soll jedes Teilgebiet der Anwendung von einem einzelnen Team umgesetzt werden. Dabei hat jedes Team andere Präferenzen welche Technologie genutzt werden soll. So möchte ein Team bei den alt bewährten Technologien bleiben und entwicklet in Java. Ein Anderes Team möchte sich in neuere Technologien einarbeiten und versucht ihren Microservice mit Webtechnologien umzusetzen.
Ein weiterer Grund für die Nutzung ist die erleichterte Integration neuer Systeme sowie die Integration in ein anderes System. Gerade im Falle einer Benutzeranmeldung, die in mehreren Systemen zum Einsatz kommen kann. Je nach System können die hier entwickelten Grundbausteine erhalten bleiben und neue Funktionen mit weiteren Microservices hinzugefügt werden.

Planung Wir sind folgendermaßen vorgegangen: Zuerst haben wir geplant wie unser beispielhaftes Microservice -System aussehen soll. Dabei haben wir uns auf ein Login-System festgelegt, da dieses Szenario einen großen Anwendungsbezug hat - auf fast jeder Webseite befindet sich ein Anmeldesystem.
Um möglichst viele Aspekte von Micorservices zu erfahren und aufzeigen zu können haben wir uns dafür entscheiden, möglichst kleine Micorservices zu planen. Das bedeutet: in einem realen System wären die unterschiedlichen Funktionen wahrscheinlich in einem Service zusammengefasst, in diesem Beispiel sind diese jedoch aus Anschauungsgründen in unterschiedliche Microservices aufgeteilt.
Schon an diesem kleinen Beispiel ist erkennbar, dass die Planung der Architektur sehr Schwierig ist, da wir uns auch schon bei der Einteilung der Services nicht immer sicher waren wie etwas am Besten aufzuteilen und umzusetzen ist. Bei größeren Systemen wäre dieses Problem um Längen schwerer zu lösen.
Nachdem die einzelnen Services den Teams/Personen zugeordnet waren, wurden diese, völlig voneinander getrennt, entwickelt. Wir haben keine Technologie vorausgesetzt und jeder durfte für sich entscheiden, welche er als am sinnvollsten hielt bzw. worin die Person Erfahrung sammeln wollte.
Bei der Architektur haben wir uns dafür entschieden die Backendservices, die auf die Datenbanken zugreifen, von der Benutzeroberfläche zu trennen. Dadurch hatten wir mehr Flexibilität bezüglich Ausführbarkeit der Oberfläche im Browser.

Durchführung Nach dem Aufteilen der Services hat jeder für sich die Services entwickelt ohne dass dafür zusätzliche Absprachen benötigt wurden. So wurde die Benutzeroberfläche mit Typescript und React.js als Webapp im Material Design entwickelt. Der Microservice zum Registrieren neuer Nutzer wurde mittels Java umgesetzt. Die Services zum anzeigen aller Daten und zum anmelden läuft auf node.js und wurde in Typescript

geschrieben. Für die Datenbanken wurde zwei mal MySQL benutzt, was jedoch nicht abgesprochen wurde. Die Kommunikation zwischen den Microservices läuft, wie es auch in vielen größeren Microservice-Systemen üblich ist, über HTTP und basiert auf REST.
Es ist erkennbar wie die Wahl der Technologien bei der Entwicklung keine Rolle gespielt hat und neue Sprachen ausprobiert werden konnten.

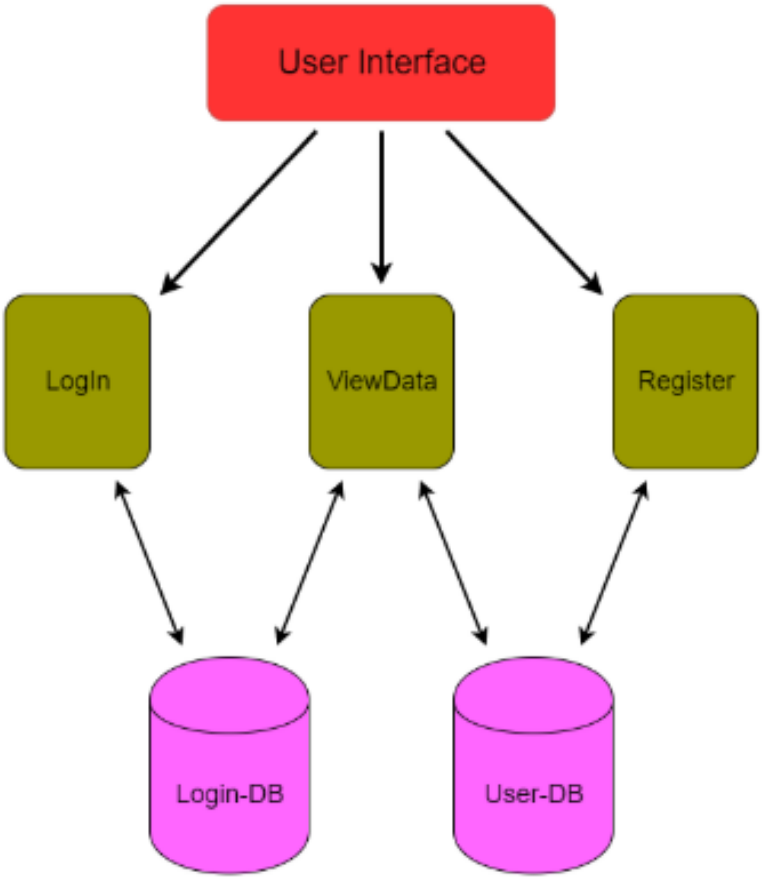
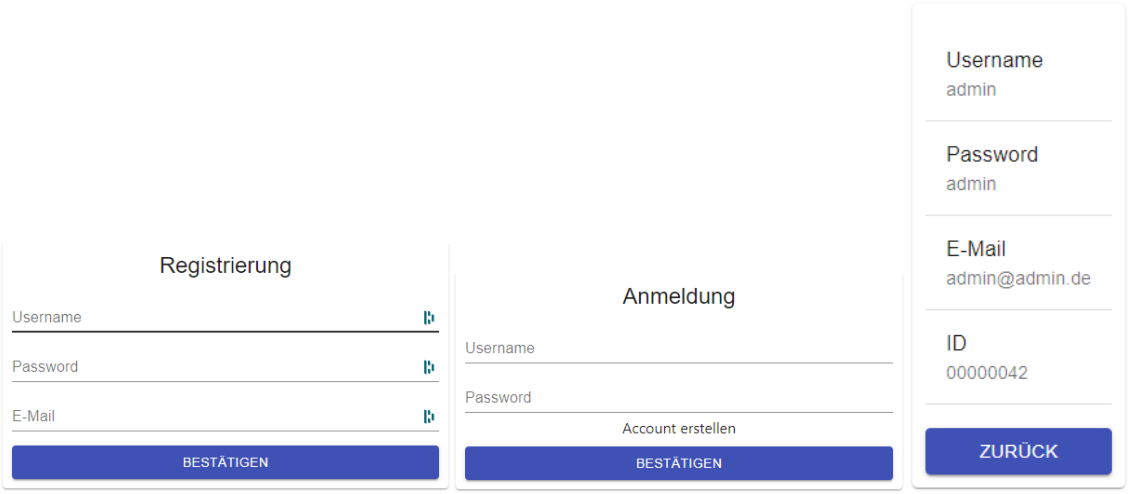


Abbildung 8.1: Die Architektur der Fallstudie

Funktionsweise In der **User-DB** sind die Daten zu den Nutzern gespeichert wie Nutzername, Email und Passwort. Zum identifizieren der Nutzer wird jedem Nutzer eine ID zugewiesen die ebenfalls in dieser Datenbank gespeichert wird. Diese Daten werden von dem **Register** Microservice angelegt, sobald sich ein Nutzer registriert.
Gelesen werden diese Daten vom **ViewData** Microservice, wenn dem Nutzer seine gesamten Daten angezeigt werden. Dazu zählen ebenfalls Daten aus der Login-DB.
In der **Login-DB** wird gespeichert, ob ein Nutzer derzeit angemeldet ist. Beim anmelden genereirt der **LogIn** Microservice ein Token, welches in der Datenbank gespeichert wird und ebenfalls dem Nutzer zurückgegeben wird. In der Datenbank wird dem Nutzer das Token mittels der ID zugeordnet.
Mit diesem Token kann der Nutzer den **ViewData** Microservice aufrufen der zu dem Nutzer die Daten ausgibt (inclusive ID und Token aus Login-DB).
Die gesamte Steuerung kann der Nutzer über die Benutzeroberfläche einfach und intuitiv vornehmen. Dazu gibt es 3 Verschiedene Ansichten.



(a) Registirerung (b) Anmeldung (c) Datenausgabe

Abbildung 8.2: Die Benutzeroberfläche

Herausforderung Beim Entwicklungsprozess sind uns selbst in einem so kleinen und primitiven Projekt viele der Herausforderungen aufgefallen. So ist uns das sinnvolle Aufteilen der Funktionen und Datenbanken schwer gefallen. Bei großen Projekten ist dies wahrscheinlich exponentiell komplexer.
Auch wäre eine realistische Infrastruktur aufgrund des Initalen Administrationsaufwandes nicht möglich gewesen (Router konfigurieren, pro Micorservice eine VM aufsetzen ...).

9 Fazit

Letztendlich erreicht die Microservice Architektur ihr Ziel: Systeme können sehr gut auf viele Server verteilt und damit skaliert werden. Allerdings ist die Architektur nicht in jeder Situation optional, denn für jeden Vorteil, den die Architektur gegenüber der klassischen Monolith Architektur hat, existiert ein Nachteil, der andere Probleme macht. Beispielsweise sind die Daten zwar sehr verteilt, dafür ist es allerdings deutlich schwerer die Daten als Ganzes zu verwalten und Transaktionen sind praktisch nur sehr schwer umsetzbar. Letztendlich bleibt jedoch der Vorteil der Skalierbarkeit bestehen und wenn bei einem großem System die Wahl zwischen den skalierenden Microservices und einem nicht skalierendem Monolithen besteht, dann müssen selbstverständlich die Microservices gewählt werden, da das System sonst schlicht nicht umsetzbar ist. Ein Monolith ist allerdings eine völlig valide und gerechtfertigte Wahl, wenn die Skalierbarkeit eines Microservice-Systems nicht benötigt wird.

Literatur

[1] Andreas Fink. *Monolithisches IT-System*. Internetseite. Okt. 2012. URL: https://www.enzyklopaedie-der-wirtschaftsinformatik.de/wi-enzyklopaedie/lexikon/is-management/Systementwicklung/Softwarearchitektur/Architekturparadigmen/Monolithisches-IT-System.

[2] Mike Gancarz. *The UNIX Philosophy*. Hrsg. von Digital Press. Digital Press, 1995.

[3] Tom Huston. *Was ist eine Microservice-Architektur?* Internetseite. März 2020. URL: https://smartbear.de/learn/api-design/what-are-microservices/.

[4] James Lewis Martin Fowler. „Microservices: Nur ein weiteres Konzept in der Softwarearchitektur oder mehr?“ In: *www.objektspektrum.de* (2015).

[5] Microsoft. *Architekturstil für Microservices*. Internetseite. Okt. 2019. URL: https://docs.microsoft.com/de-de/azure/architecture/guide/architecture-styles/microservices.

[6] Keyhole Software. *Microservices: Patterns for Enterprise agility and scalability*. 2017.

[7] Johannes Thönes. *Microservices*. Internet IEEE Software. Feb. 2015. URL: https://ieeexplore.ieee.org/abstract/document/7030212/metrics#metrics.

[8] Eberhard Wolff. *Microservices Grundlagen flexibler Softwarearchitekturen*. Hrsg. von dpunkt.verlag. dpunkt.verlag, 2018.

[9] Liang jie Zhang. „SOA and Web Services“. In: *SOA and Web Services*. IEEE. IEEE, Sep. 2006. DOI: 10.1109/SCC.2006.94.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

Anhang

A Erster Abschnitt des Anhangs

In den Anhang gehören „Hintergrundinformationen“, also weiterführende Information, ausführliche Listings, Graphen, Diagramme oder Tabellen, die den Haupttext mit detaillierten Informationen ergänzen.
Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. Der Text gibt lediglich den Grauwert der Schrift an. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. Er muss keinen Sinn ergeben, sollte aber lesbar sein. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln. Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. Der Text gibt lediglich den Grauwert der Schrift an. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. Er muss keinen Sinn ergeben, sollte aber lesbar sein. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln. Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. Der Text gibt lediglich den Grauwert der Schrift an. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. Er muss keinen Sinn ergeben, sollte aber lesbar sein. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln.

Kolophon

Dieses Dokument wurde mit der LaTeX-Vorlage für Abschlussarbeiten an der htw saar im Bereich Informatik/Mechatronik-Sensortechnik erstellt (Version 2.1). Die Vorlage wurde von Yves Hary und André Miede entwickelt (mit freundlicher Unterstützung von Thomas Kretschmer, Helmut G. Folz und Martina Lehser). Daten: (F)10.95 – (B)426.79135pt – (H)688.5567pt