ingenieur
wissenschaften
htw saar

Hochschule für
Technik und Wirtschaft
des Saarlandes
University of
Applied Sciences

Projektarbeit

im Modul Software-Architektur
an der Hochschule für Technik und Wirtschaft des Saarlandes
im Studiengang Praktische Informatik
der Fakultät für Ingenieurwissenschaften

Cloud-Native Architekturen

vorgelegt von
Tristan Gläs und Carolin Becker

betreut und begutachtet von
Prof. Dr. Markus Esch

Saarbrücken, 22. März 2021

Selbständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit (bei einer Gruppenarbeit: den entsprechend gekennzeichneten Anteil der Arbeit) selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich erkläre hiermit weiterhin, dass die vorgelegte Arbeit zuvor weder von mir noch von einer anderen Person an dieser oder einer anderen Hochschule eingereicht wurde.

Darüber hinaus ist mir bekannt, dass die Unrichtigkeit dieser Erklärung eine Benotung der Arbeit mit der Note „nicht ausreichend“zur Folge hat und einen Ausschluss von der Erbringung weiterer Prüfungsleistungen zur Folge haben kann.

Saarbrücken, 22. März 2021

Tristan Gläs und Carolin Becker

Zusammenfassung

Kurze Zusammenfassung des Inhaltes in deutscher Sprache, der Umfang beträgt zwischen einer halben und einer ganzen DIN A4-Seite.

Orientieren Sie sich bei der Aufteilung bzw. dem Inhalt Ihrer Zusammenfassung an Kent Becks Artikel: `http://plg.uwaterloo.ca/~migod/research/beckOOPSLA.html`.

- cloud native erklären - fallbeispiel - diskutieren

We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.

— Donald E. Knuth [5]

Danksagung

Hier können Sie Personen danken, die zum Erfolg der Arbeit beigetragen haben, beispielsweise Ihren Betreuern in der Firma, Ihren Professoren/Dozenten an der htw saar, Freunden, Familie usw.

Inhaltsverzeichnis

1 Einleitung

1.1 LaTeX installieren und einrichten

1.1.1 Unter Windows

Als LaTeX-Distribution unter Windows steht *MikTeX* zu Verfügung, die als freie Software im Internet erhältlich ist. *MikTeX* unterstützt Windows XP, Vista und Windows 7. Neben *MikTeX* wird noch ein PostScript-Interpreter benötigt, z.B. GhostScript, zu finden auf Chip.de.

Wichtig: Bei *MikTeX* unbedingt Vollinstallation auswählen, sonst sind eventuell benötigte Packages nicht vorhanden.

1.1.2 Unter Linux

Unter Linux existiert die LaTeX-Distribution *texlive*, die als aktuelle Version aus den Paketquellen geladen werden kann (unter Ubuntu mit `apt-get install texlive-full`). Auch hier ist ganz wichtig, die volle Distribution zu laden, damit alle Packages zur Verfügung stehen.

1.2 Entwicklungsumgebungen

Hat man die passende Distribution installiert, bieten sich vielerlei Möglichkeiten an ein LaTeX-Projekt anzugehen oder einzelne Dokumente zu editieren. Unter Windows könnten dies folgende sein:

TeXnicCenter Umfangreiche Entwicklungsumgebung mit Projektorganisation und Autovervollständigung

TeXLipse Eclipse-Plugin, das alle Vorteile der Eclipseumgebung mit LaTeX verbindet

TeXmaker Einfacher LaTeX-Editor mit Pdf-Direktvorschau

Unter Linux stehen bereit:

Gummi Ebenfalls einfacher LaTeX-Editor mit Direktvorschau

TeXLipse Auch für Linux erhältlich

Kile Umfangreiche Entwicklungsumgebung, ähnlich wie TeXnicCenter

Nach der Installation muss die Entwicklungsumgebung eingerichtet werden; dazu finden sich viele Anleitungen im Internet, die genau erklären, welche Distribution auf welche Weise eingerichtet wird. Insbesondere sollte der PDF-Viewer festgelegt werden, damit bei Gummi und TeXmaker die Direktvorschau funktioniert. Manchmal kommt es vor, dass die Ausgabe nach dem Kompilieren Umlaute und Sonderzeichen nicht richtig darstellt. Unter Linux hängt dies mit den unterschiedlichen Zeichensätzen zusammen, die unterstützt werden. Um diese Vorlage zu verwenden ist es notwendig, den verwendeten Zeichensatz des Editors bzw. der Entwicklungsumgebung auf den in diesem Dokument verwendeten Zeichensatz umzustellen: UTF-8 ohne BOM (Byte Order Mark).

1.3 Werkzeuge

JabRef Ein Literaturverwaltungsprogramm, welches das *BibTeX*-Format einsetzt und mithilfe einer graphischen Oberfläche das Anlegen von Literaturverzeichnissen vereinfacht.

1.4 Struktur und Gebrauch der Vorlage

Die vorliegende Vorlage für Abschlussarbeiten besteht aus einer internen Struktur, die grundsätzlich nicht verändert werden sollte.

1.4.1 Struktur der Vorlage

htw-i-mst-config.tex Enthält alle zu ladenden Packages, Styleparameter für Hyperlinks, Codelistings und Literaturverzeichnis sowie globale Parameter für Tabellen und Beschriftungen. Im Besonderen befinden sich hier die Variablen für den eigenen Namen, Titel, Datum der Arbeit, den betreuenden Professor etc.

htw-i-mst-vorlage.tex Dies ist die Hauptdatei, in der alle notwendigen **.tex*-Dateien eingebunden werden, die zu dem Dokument gehören. Es empfiehlt sich die interne Struktur *nicht* zu verändern. Eigene Kapitel werden an der dafür markierten Stelle eingebunden.

Bibliography.bib Zentrale Datei für die Literaturangaben, welche man z.B. mit JabRef verwalten kann.

Chapters/ Ablageort für alle selbst angelegten Kapitel der Arbeit. Die Aufteilung in eigene Dateien erleichtert die Übersicht über den Quellcode.

Graphics/ Ablageort für alle im Dokument benötigten Grafikdateien. Gerne darf man hier Unterverzeichnisse zur besseren Strukturierung anlegen.

Examples/ Dieser Ordner enthält die in dieser Vorlage beigefügten LaTeX-Beispiele, welche vor der Abgabe der Arbeit selbstverständlich gelöscht werden sollten.

Frontbackmatter/ In diesem Ordner sind all jene Dateien abgelegt, die – außer dem Kerntext in *Chapters/* – die Gesamtheit der Abschlussarbeit ausmachen.

- **Titlepage.tex** Definiert die Titelseite der Abschlussarbeit. Diese Datei muss normalerweise nicht verändert werden.
- **Abbreviations.tex** Hier werden alle Abkürzungen hinterlegt, die im Dokument verwendet werden.
- **Abstract.tex** Eine kurze Zusammenfassung der Abschlussarbeit wird in diese Datei eingefügt.
- **Acknowledgements.tex** Dort finden Danksagungen ihren Platz.
- **ConfidentialityClause.tex** Beinhaltet den Sperrvermerk und ist nur zu verwenden, falls dies beispielsweise vom beteiligten Unternehmen gefordert wird.
- **Contents.tex** Enthält wichtige Eintragungen in die *Table-of-Contents*. Diese Datei muss normalerweise nicht geändert werden.
- **Declaration.tex** Enthält die Selbständigkeitserklärung. Diese Datei darf nicht geändert werden.

Colophon.tex Enthält einen Hinweis auf die Urheber dieser Vorlage. Diese Datei darf nicht geändert werden.

ListOfs.tex Enthält die Einträge für die Tabellen- und Abbildungsverzeichnisse etc. und muss gewöhnlich nicht verändert werden.

1.4.2 Gebrauch der Vorlage

Grundsätzlich ist nicht viel zu tun, um die Vorlage für Abschlussarbeiten zu verwenden. Man entpackt den Hauptordner in das gewünschte Verzeichnis und nutzt die Dateien so, wie in Abschnitt 1.4.1 beschrieben. Danach werden *alle* Dateien gespeichert und die Hauptdatei, *htwsaar-i-mst-vorlage.tex*, mehrfach kompiliert (LaTeX benötigt mehrere Durchgänge um z.B. Referenzen richtig zuzuordnen). Hat man Änderungen in *Bibliography.bib* bzw. *Bibliography.tex* vorgenommen oder neue Zitate z.B. mittels `\cite` eingefügt, muss erst mit *BibLaTeX* und anschließend mitdem entsprechenden LaTeX-nach-PDF-Compiler übersetzt werden.

2 Beispiele

2.1 Abkürzungen

Um Abkürzungen zu verwenden, muss über `\usepackage{acronym}` das benötigte Package geladen werden. Danach kann man lange Begriffe ganz bequem abkürzen:

So muss man nicht ständig Wireless Local Area Network (WLAN) ausschreiben, auch Transmission Control Protocol (TCP) lässt sich abkürzen. Würde man im Text WLAN oft verwenden, kann man sie, wie hier, nur als Abkürzung anzeigen lassen - oder bei Bedarf die Erklärung mitliefern (Wireless Local Area Network (WLAN)). Weiteres Beispiel könnte die Gang of Four (GoF) sein.

Weitere Informationen sind im Acronym-Manual zu finden.

2.2 Beispiel für BibLaTeX

BibLaTeX ist ein Package, das einem die Arbeit mit Zitaten bzw. Quellenangaben erleichtern kann. Mit JabRef (Abschnitt 1.3) ist es möglich **.bib*-Dateien zu erstellen, in denen alle Angaben zu Autor, Buchtitel, Erscheinungsdatum usw. hinterlegt werden, welche zum passenden Zeitpunkt abgerufen werden können. Das Literaturverzeichnis wird mittels `\printbibliography` ausgegeben.

Im Allgemeinen wird im Literaturverzeichnis auch nur jene Literatur aufgenommen, die auch in der **.tex*-Datei referenziert wird. Danach ist es wichtig nicht nur mit *PdfLaTeX*, sondern auch mit *BibLaTeX* zu kompilieren, damit die zitierten Einträge in die verschiedenen Hilfsdateien aufgenommen werden können.

Einige Zitate

In diesem Satz könnten wir auf [4] verweisen, ebenso auf das wichtige Werk [3]. Wenn uns das nicht genug ist, sollten wir das anmerken, was in [6] geschrieben wurde. Im Zweifelsfall verweisen wir auf eine einzelne Seite, wie in [1, S. 112] zu finden.

Üblicherweise wird auch der Name des Autors bzw. der Autoren genannt, also beispielsweise bei einem Verweis auf Knuth [4] oder auch bei mehreren Autoren Cormen u. a. [2]. LaTeX stellt Mechanismen zur Verfügung, auch dies automatisiert zu erledigen.

2.3 Referenzierungen

Mit Referenzierungen kann ich ganz bequem auf Textpassagen, Kapitel, Sections oder Abbildungen im weiteren Text verweisen. Dies ist ein Verweis auf Abschnitt 2.3.1, der sich auf Abschnitt 2.3.1 befindet.

Auch ein Verweis auf Tabelle 2.1 auf Seite 7 ist möglich.

Man sollte beachten, dass man sein Dokument, wenn es Referenzierungen enthält, mehrmals kompiliert, da sonst manche Verweise nicht aufgelöst werden können.

2.3.1 Beispieltext

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.

2.4 Dateien einbinden

Damit man nicht alle Einstellungen, Optionen, Packages und Texte, Abbildungen etc. in einer Datei unterbringen muss, werden zwei Befehle bereitgestellt, um externe **.tex*-Dateien einzubinden: `\include{PFAD}` und `\input{PFAD}`. Mit dem erstem Befehl wird eine neue Seite angelegt, danach kommen die Inhalte aus der angegebenen Datei; mit dem zweiten Befehl wird keine neue Seite angelegt – der Inhalt der angegebenen Datei wird direkt an die betroffene Stelle eingefügt.

Wichtig: Der *Pfad* wird sinnigerweise *relativ* angegeben, wobei als Stammverzeichnis jenes Verzeichnis angesehen wird, in dem die **.tex*-Datei mit der *Document*-Umgebung abgelegt ist (in diesem Fall ist es *htwsaar-i-mst-config.tex*).

2.5 Tabellen

2.5.1 Einfache Tabelle

In LaTeX lassen sich Tabellen unterschiedlicher Ausprägung einfach erzeugen. Das allgemeine Format einer Tabelle sieht aus wie folgt:

Listing 2.1: Allgemeines Format

```
\begin{table}
        \caption{BESCHRIFTUNG}
        \begin{tabular}{FORMATIERUNG}
                TABELLENINHALT
        \end{tabular}
\end{table}
```

Eine Beispieltabelle (Tabelle 2.1) könnte also so aussehen:

Listing 2.2: Tabelle 2.1

```
\begin{table}
        \caption{Beispiel 1}
        \begin{tabular}{lrcr}
                \toprule
                \textbf{Name} & \textbf{Vorname} & \textbf{
                   Matrikelnummer} & \textbf{Lieblingsspeise}\\
                \midrule
                Jackson & Michael & 123456 & Erdbeereis \\
                Springsteen & Bruce & 234567 & Schwedisches Lakritz
                    \\
```

```
                Bach & Anna, Magdalena & 3456789 & Frankfurter
                    Kranz \\
                Schumann & Clara & 4567890 & Bisquitt\"ortchen \\
                \bottomrule
        \end{tabular}
        \label{tab:beispieltabelle1}
\end{table}
```

Mit `\caption{Beispiel 1}` bekommt unsere Tabelle eine Beschriftung am Tabellenkopf. `l|r|c|r` legt die Textausrichtung der einzelnen Spalten fest: `l` bedeutet linksausgerichtet, `r` rechtsausgerichtet und `c` zentriert. Durch `|` werden Spaltenlinien gezogen. `\toprule`, `\midrule` und `\bottomrule` erzeugen Kopf-, Mittel- und Abschlusslinie in der Tabelle. Als Spaltentrenner wird das `&` genutzt, Zeilentrenner ist der doppelte Backslash (`\\`). Am Ende kann die Tabelle auch mit einem Label versehen werden (`\label{tab:beispieltabelle1}`), über welches diese referenziert wird.

2.5.2 Erweiterte Tabellenbefehle

Um Tabellen in LaTeX flexibler zu gestalten gibt es weitere Befehle bzw. zusätzliche Pakete, die einem das Leben leichter machen (Tabelle 2.2). Hierzu ein weiteres Beispiel:

Listing 2.3: Tabelle 2.2

```
\begin{table}
        \centering
        \caption{Beispiel 2}
        \begin{tabular}{lll}
                \hline
                Author & Title & Year \\
                \hline
                \hline
                \multirow{3}{*}{Stanislav Lem} & Solaris & 1961 \\
                        & Roboterm\"archen & 1967 \\
                        & Der futurologische Kongress & 1971 \\
                \hline
                \multirow{3}{*}{Isaac Asimov} & Ich, der Robot &
                    1952 \\
                        & Der Tausendjahresplan & 1966 \\
                        & Doctor Schapirows Gehirn & 1988 \\
                \hline
        \end{tabular}
\label{tab:beispieltabelle2}
\end{table}
```

Tabelle 2.1: Beispiel 1

Name	Vorname	Matrikelnummer	Lieblingsspeise
Jackson	Michael	123456	Erdbeereis
Springsteen	Bruce	234567	Schwedisches Lakritz
Bach	Anna, Magdalena	3456789	Frankfurter Kranz
Schumann	Clara	4567890	Bisquittörtchen

Tabelle 2.2: So sollte man es nicht machen! Beispiel für einen schlechten Tabellenstil

Author	Title	Year
Stanislav Lem	Solaris	1961
	Robotermärchen	1967
	Der futurologische Kongress	1971
Isaac Asimov	Ich, der Robot	1952
	Der Tausendjahresplan	1966
	Doctor Schapirows Gehirn	1988

Mit `\centering` wird die Tabelle zentriert ausgerichtet, analoge Befehle für rechts- bzw. linksausrichtung sind z.B. `\raggedleft` und `\raggedright`.

Eine weitere Form der Tabellen ist das Package *tabularx*, das variable Spaltenbreiten unterstützt, und *booktabs*, welches mit horizontalen Linien besser arbeiten kann.

2.6 Abbildungen

LaTeX unterstützt generell die Formate **.jpeg*, **.png* und **.pdf*. Handelt es sich z.B. um Strichgrafiken oder skalierbare Farbflächen, sollte **.pdf* die erste Wahl sein, da sich in diesem Format Vektorgrafiken ohne Qualitätsverlust darstellen bzw. skalieren lassen.

Abbildung 2.1: Erstes Bild, Völklinger Hütte

2.6.1 Wrapfigure

Abbildung 2.1 ist zwar ganz nett anzusehen, aber vielleicht sähe es eleganter aus, wenn die Abbildung von unserem Textabschnitt umflossen werden würde. Diese Art von Abbildungen sollte jedoch sparsam und mit großer Sorgfalt eingesetzt werden, da es zu unschönen Darstellungen kommen kann. 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.

Abbildung 2.2: Völklinger Hütte, *.jpg

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.

2.6.2 Subfigures

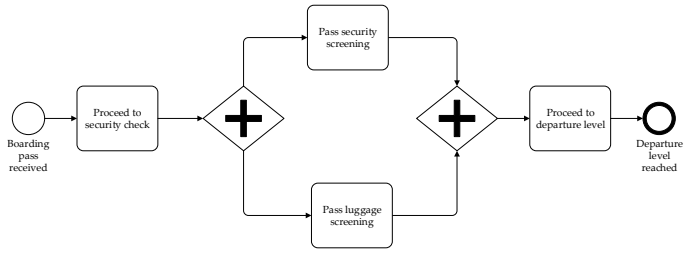
Es ist ebenso möglich, mehrere Abbildungen nebeneinander zu setzen, wie in Abbildung 2.3 zu sehen ist. Eine separate Referenzierung ist auch möglich: Abbildung 2.3a.

(a) Erstes ... (b) ... und zweites Bild

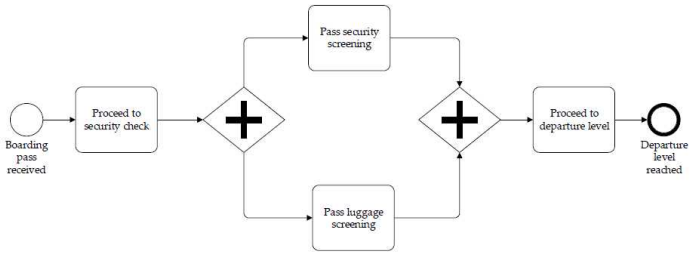
Abbildung 2.3: Mehrere Abbildungen nebeneinander

2.6.3 Qualitätsunterschiede

Leider haben die unterschiedlichen Grafikformate bedingt durch die unterschiedlichen Kompressionsverfahren einige Schwächen, insbesondere die Umwandlung in das *JPG*-Format erzeugt unangenehme Artefakte im Bild. Abbildung 2.4 zeigt die Unterschiede zwischen *PDF-Format* und *JPG-Format* im Vergleich.



(a) *PDF*-Format



(b) *JPG*-Format

Abbildung 2.4: Beide Formate im Vergleich

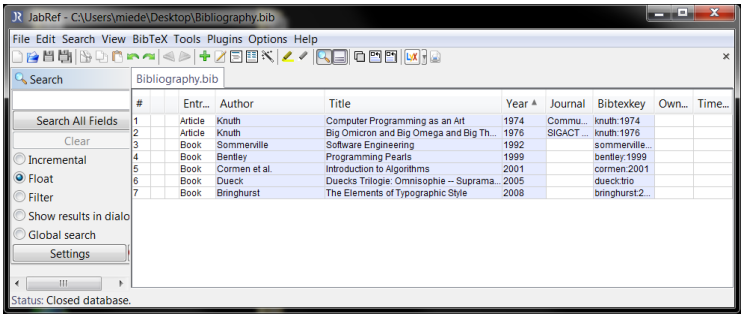


Abbildung 2.5: *PNG*-Format

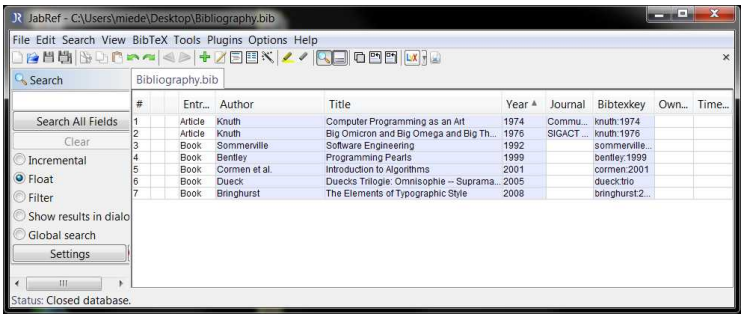


Abbildung 2.6: *JPG*-Format

Wenn eine **.pdf*-Datei nicht infrage kommt, beispielsweise bei Screenshots, ist unbedingt das *PNG-Format* vorzuziehen. Den Unterschied machen Abbildung 2.5 und Abbildung 2.6 deutlich.

„Faustregeln“im Umgang mit Abbildungen:

- Diagramme bzw. alles, was Linien usw. enthält: *PDF* (im Vektorformat).
- Screenshots bzw. alles, was größere gleichfarbige Flächen enthält: *PNG*.
- Der Rest (in der Regel Fotos): *JPEG*.

2.7 Quellcode einbinden

Das Package *lstlisting* ermöglicht es, Quellcode ansprechend in das Dokument einzubinden. Man kann Quellcode einzeilig einbinden mittels `\lstinline|Quellcode|`. Dabei ist darauf zu achten, dass der Befehl einmal mit { } und einmal mit | | aufgerufen werden kann, je nachdem, welche Zeichen im angegebenen Quelltext genutzt werden. Es ist auch möglich eine eigene Umgebung für Quelltext zu schaffen:

Listing 2.4: Erstes Listing

```
private Umgebung(int i, int k)
{
        System.out.println("Eine Funktion mit " + i + "und" + k ".");
}
```

Wer Quelltext aus externen Dateien einbinden möchte, geht wie folgt vor:

Listing 2.5: Externer Quellcode

```
public class HalloWelt {
    public static void main(String[] args) {
        System.out.println("Hallo Welt!");
    }
}
```

Wie genau der Quellcode formatiert und gefärbt ist, ist in *htwsaar.i.mst.config.tex* hinterlegt, wobei fü verschiedene Sprachen auch eigene Styles angelegt werden können (hier z.B. für Java).

2.8 Mathematische Ausdrücke

Mathematische Ausdrücke sind eine kleine Kunst für sich. Am allereinfachsten kann man eine Formel, wie $a + b = c$ in den Fließtext einbinden, wobei LaTeX die Höhe der Ausdrücke der Zeile anpasst, wie hier zu sehen $\sum_{y=0}^{x} a$. In einer Umgebung sieht das schon anders aus:

$$\sum_{y=0}^{x} a \tag{2.1}$$

Griechische Buchstaben:

$$\alpha\beta\gamma\delta\epsilon\varepsilon\zeta\eta\theta\iota\kappa\lambda\mu\nu\xi\pi\varpi\rho\varrho\sigma\tau\upsilon\phi\varphi\chi\psi\omega \tag{2.2}$$

Brüche:

$$Ergebnis = \frac{a}{b} \tag{2.3}$$

$$\frac{\sin \alpha^2 + \cos \alpha^2}{1} = 1 \tag{2.4}$$

$$\frac{-9x}{\frac{2y}{3z+2}} \tag{2.5}$$

Text innerhalb von Formeln:

$$\sum_{y=1}^{n} y = \frac{n * (n+1)}{2} \quad \text{Gaußsche Summenformel} \tag{2.6}$$

Hoch- bzw. Tiefstellungen:

$$x_{i,j}^2 \tag{2.7}$$

$${x_{i,j}}^2 \tag{2.8}$$

$$x_{n_0} \tag{2.9}$$

Matrizen: Matrizen werden innerhalb der mathematischen Umgebung als wiederum neue Umgebung eingebunden. Wie bei Tabellen auch werden Zeilen durch \\ und Spalten durch & getrennt.

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \tag{2.10}$$

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} \tag{2.11}$$

Fallunterscheidung:

$$f(x) = \begin{cases} 0, & \text{falls } x < 0 \\ 1, & \text{falls } x \geq 0 \end{cases} \tag{2.12}$$

2.9 To-Do-Notes

Um bei einer längeren Arbeit nicht den Überblick zu verlieren, an welcher Stelle es nötig ist weiter zu arbeiten, bietet es sich an, kleine Notizen einzufügen. Das Package *todonotes* stellt eine elegante Lösung bereit, um differenziert und vielfarbig jene Abschnitte zu kennzeichnen, die einer weiteren Bearbeitung bedürfen.

Beispiel für To-Do-Notes

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 Les- barkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschie- dene 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.

Plain todonotes.

Plain todonotes.

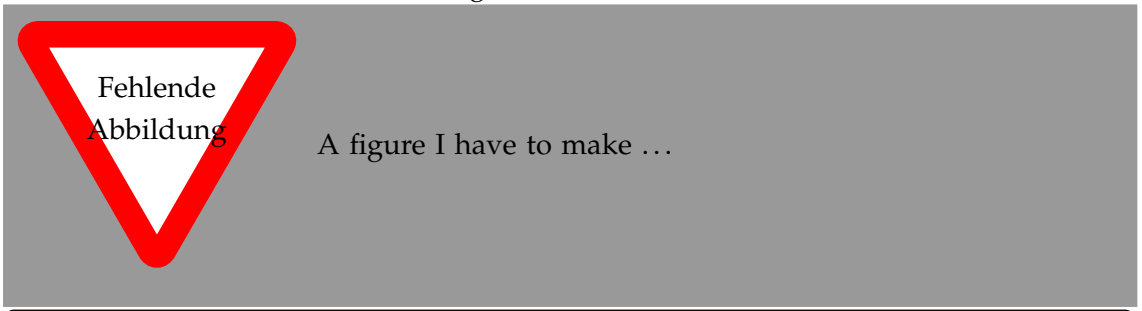
Todonote that is only shown in the margin and not in the list of todos.

A very long todonote that certainly will fill more than a single line in the list of todos. Just to make sure let's add some more text …

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.

A note with no line back to the text.

A very long todonote that certainly will fill more than a single line in the list of todos …



1: Erste Nummer...

2: Zweite Nummer...

Nachfolgend wird noch eine Liste aller To-Dos auf einer separaten Seite ausgegeben.

Liste der noch zu erledigenden Punkte

3 Einleitung

Cloud-Dienste sind heutzutage nicht mehr aus unserem Alltag wegzudenken. Auch im Arbeitsleben bzw. für viele Unternehmen ist das Thema Cloud im Zusammenhang mit der Digitalisierung ein wichtiger Bestandteil geworden. Dadurch wurden neue Geschäftsmodelle ermöglicht und gleichzeitig die Wettbewerbsfähigkeit des Unternehmens gesteigert.

Ein neuer Ansatz ist Cloud Native. Hierbei werden die Applikationen von Beginn an für den Einsatz in der Cloud entwickelt. Dieser wird als Zukunft der Software-Entwicklung gesehen und basiert auf den bewährten Technologien des Cloud Computings.
Die Weiterentwicklung und der Ausbau des Cloud Computing Bereiches, welche durch die steigenden Zahlen von Benutzern und Daten schnell vorangetrieben werden, benötigen auch entsprechende Softwaresysteme. Diese können mit den bereitgestellten Ressourcen effektiv umgehen und sind auf die Cloud Umgebung abgestimmt.
Cloud-Native hat sich in den letzten Jahren als eigener Bereich in der Informatik herauskristallisiert und wurde vorallem durch große Unternehmen wie Netflix, Google und Amazon geprägt.
Auch die Verbreitung von Cloud Native Technologien insbesondere der Container und Container-Orchestrierungs-Tools wie z.B. Kubernetes sind in den letzten Monate deutlich gestiegen.
Durch Cloud Native ergibt sich eine neue Möglichkeit große komplexe Systeme zu entwickeln. Dies ist besonders für Unternehmen interessant, da somit in kürzester Zeit innovative Anwendungen entwickelt werden können.
Benutzer bzw. Kunden werden zudem immer anspruchsvoller und erwarten, dass schnellstmöglich neue Anwendungen für sie zur Verfügung stehen. Des Weiteren wird eine schnelle Reaktionsfähigkeit, innovative Features und eine möglichst geringe Ausfallzeit erwartet.
Der Cloud Native Ansatz ist für solche Situationen ausgelegt und bietet die Möglichkeit Applikationen, je nach Bedarf zu skalieren.

https://www.cncf.io/blog/2020/08/14/state-of-cloud-native-development/

4 Cloud Native

In zweiten Kapitel gehen wir auf die Definition von Cloud-Native Architekturen ein, grenzen sie von anderen Absätzen ab und betrachten wichtige Eigenschaften. Abschließend beschäftigen wir uns mit den Vor-und Nachteilen und den typischen Einsatzgebieten.

4.1 Cloud Computing

Bevor wir uns mit Cloud-Native Architekturen auseinandersetzten können, müssen wir uns zuerst mit dem Cloud Computing beschäftigen, da es die Basis für diese Architekturen bildet und sie maßgeblich beeinflusst. Die NIST Definition von Cloud Computing enthält die wichtigsten Merkmalen.

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models.

Entscheidend für Cloud-Native ist nun die schnelle Bereitstellung von Resourcen, denn dies eröffnet neue Möglichkeiten hinsichitlich der Skalierbarkeit und hat dadurch einen starken Einfluss auf die Architekturen.

4.2 Definition Cloud Native

Was genau Cloud-Native ist und wie man es definieren kann ist schwierig, da der Begriff noch relativ neu ist. Eine erste Version einer Definition kommt von der Cloud Native Computing Foundation, einer Organisation, die als Vorreiter in Sachen Cloud-Native gilt.

CNCF Cloud Native Definition v1.0

Cloud native Technologien ermöglichen es Unternehmen, skalierbare Anwendungen in modernen, dynamischen Umgebungen zu implementieren und zu betreiben. Dies können öffentliche, private und Hybrid-Clouds sein. Best Practices, wie Container, Service-Meshs, Microservices, immutable Infrastruktur und deklarative APIs, unterstützen diesen Ansatz.
Die zugrundeliegenden Techniken ermöglichen die Umsetzung von entkoppelten Systemen, die belastbar, handhabbar und beobachtbar sind. Kombiniert mit einer robusten Automatisierung können Softwareentwickler mit geringem Aufwand flexibel und schnell auf Änderungen reagieren.
Die Cloud Native Computing Foundation fördert die Akzeptanz dieser Paradigmen durch die Ausgestaltung eines Open Source Ökosystems aus herstellerneutralen Projekten. Wir demokratisieren modernste und innovative Softwareentwicklungs-Patterns, um diese Innovationen für alle zugänglich zu machen.

Diese Definition lässt gewollt viel Spielraum zur Interpretation, jedoch stechen ein paar wichtige Merkmale heraus. Diese sind inbesondere Entkopplung, Belastbarkeit, robuste Automatisierung und Skalierbarkeit. Die Cloud Native Computing Foundation hat diese Defintion als Version 1.0 makiert, was darauf schließen lässt, das Änderungen oder Erweiterungen erwartet werden.

4.3 Microservices

Microservices sind in Cloud Native Systemen zum Erstellen von Anwendungen ein beliebter Architekturstil. Bei dieser Architektur besteht die Software aus kleinen, unabhängigen Services bzw. Modulen, die über definierte APIs (application programming interface) kommunizieren. Jeder Service kann unabhängig von anderen Services entwickelt und bereitgestellt werden, ohne die Funktionalität anderer Services zu beeinträchtigen. Den Aufbau der Microservice-Architektur sowie die Darstellung der einzelnen unabhängigen Services sind in Abbildung 4.1 zu sehen.

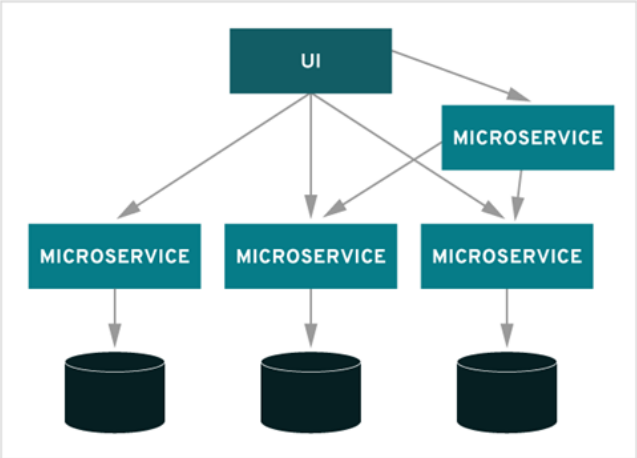


Abbildung 4.1: Aufbau einer Microservice-Architektur

Durch die Eigenständigkeit besitzt jeder Komponentenservice seinen eigenen Code sowie seine eigene Implementierung. Jede einzelne Komponente ist auf eine Reihe von Funktionen spezialisiert, sodass sie sich auf die Lösung eines bestimmten Problems fokussiert. Wenn ein einzelner Service (z.B. hinsichtlich des Codes) zu komplex wird, kann er in kleinere Services unterteilt werden.
In Cloud Native Systemen sowie in der Microservice-Architektur ist die Skalierung ein wichtiger Bestandteil. Durch die Existenz der einzelnen Services können diese je nach Nachfrage unabhängig voneinander skaliert werden. Dadurch können Subsysteme, die mehr Ressourcen benötigen aufskaliert werden, ohne die gesamte Anwendung aufzuskalieren.
Eine weitere Eigenschaft ist die Flexibilität. Durch die Unabhängigkeit der einzelnen Services wird auch deren Verwaltung vereinfacht. Bei einer Erweiterung sowie Fehlerbehebung der Anwendung muss nur der entsprechende Service verändert werden. Dies führt dazu, dass nicht die gesamte Anwendung erneut bereitgestellt werden muss.
Durch die einfache Bereitstellung können z.B. neue Konzepte ausprobiert und auch schnell wieder rückgängig gemacht werden. Auf Grund der möglichen Experimente entstehen niedrige Ausfallkosten, die Aktualisierung des Codes wird erleichtert und verein-

facht das Hinzufügen neuer Funktionen.

4.4 Container

Auch Container sind ein wesentlicher Bestandteil von Cloud Native Systemen.
Ein Container ist eine Standard-Software-Einheit, die den Code und seine Abhängigkeiten verpackt, sodass Anwendungen von ihrer Ausführungsumgebung abstrahiert werden. Mit dieser Entkopplung können containerbasierte Anwendungen schnell, zuverlässig bereitgestellt und von einer beliebigen Umgebung, wie z.B. in einer öffentlichen Cloud, ausgeführt werden. In Abbildung 4.2 ist der Aufbau einer containerbasierte Anwendung zu sehen.

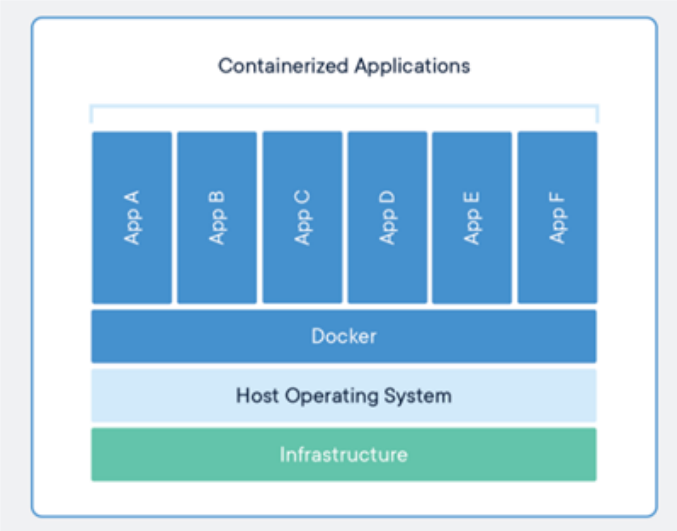


Abbildung 4.2: Aufbau einer containerbasierte Anwendung

Im Gegensatz zu virtuelle Maschinen wird bei der Containeriesierung weniger Speicherplatz benötigt. Hierbei handelt es sich um eine Virtualisierung auf Betriebssystemebene, da die Container direkt auf dem Kernel des Betriebssystems ausgeführt werden. Container teilen sich den Kernel des Betriebssystems, sodass sie schneller starten und im Vergleich zu einem vollständigen Betriebssystem nur einen Bruchteil an Speicher verwenden.
Container haben eine hohe Portabilität, sodass sie überall ausgeführt werden können. Die Softwarepakete enthalten alle Elemente, die zur Ausführung in einer beliebigen Umgebung erforderlich sind.
Auch bei der Containerisierung ist die Skalierbarkeit eine wichtige Eigenschaft. Denn hier besteht die Möglichkeit, dass die einzelnen Container je nach Ressourcenbedarf schnell gestartet sowie angehalten werden können. Zudem laufen sie isoliert von anderen Anwendungen.

Es gibt verschiedene Containerformate, mit deren Hilfe eine Anwendungsbereitstellung durch Containerisierung erleichtert wird. Bekannte Containerformate sind z.B. Docker und Kubernetes.
Docker ist ein beliebtes Open-Source-Containerformat zur Automatisierung der Bereitstellung von Anwendungen, die z.B. in der Cloud ausgeführt werden können. Docker verpackt die Software mit seinen Abhängigkeiten und alles was zur Ausführung benötigt wird in Container.

Wenn Anwendungen immer größer werden und die Betreibung immer komplexer wird, ist das Orchestierungssystem Kubernetes hilfreich.
Kubernetes ist ein Open-Source-Orchestrierungssystem zur Automatisierung z.B. zur Verwaltung, Platzierung und Skalierung von Container.

4.5 Abgrenzungen

In diesem Abschnitt wird die Monolithische Architektur im Gegensatz zur Microservices Architektur abgegrenzt. Darüber hinaus werden virtuelle Maschinen einer containerbasierten Anwendung gegenübergestellt.

4.5.1 Monolithische Architektur

Bei einer monolithischen Architektur sind die Prozesse eng miteinander verbunden und werden als einziger Service ausgeführt. Wenn ein Fehler auftritt, muss im Gegensatz zu der Microservice Architektur, die gesamte Anwendung skaliert werden und nicht nur der betreffende Service. Das Hinzufügen und Verbessern von Funktionen sowie das Umsetzen neuer Konzepte kann bei der monolithischen Architektur je nach Aufbau der Implementierung mit zunehmender Codebasis komplexer werden. Die Anwendungsverfügbarkeit ist im Gegensatz zu den Micorservices risikoreicher. Bei der monolithischen Architektur sind die einzelne Prozesse abhängig voneinander und eng miteinander verbunden, sodass die Wahrscheinlichkeit für einen Prozessausfall erhöht wird.
In Abbildung 4.3 sind die Aufbauten der monolithischen Architektur und der Microservice Architektur vergleichend dargestellt.

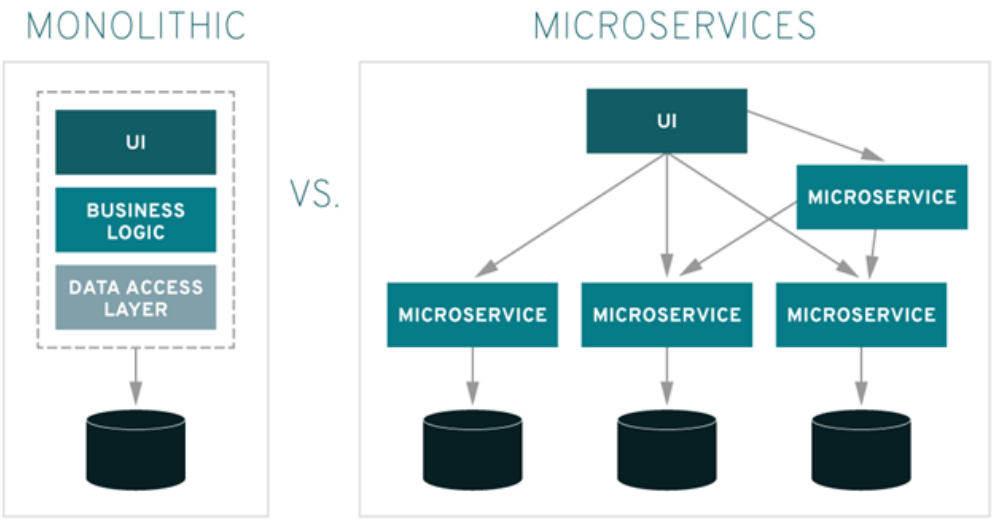


Abbildung 4.3: Vergleich der monolithischen Architektur und der Microservice Architektur

4.5.2 Virtuelle Maschinen

Im Gegensatz zu Container, sind virtuelle Maschinen eine Virtualisierung der gesamten Hardware bzw. Abstraktion der physischen Hardware, die einen Server in viele Server wandelt. In Abbildung 4.4 ist der Aufbau einer containerbasierten Anwendung im Vergleich zu einer virtuellen Maschine dargestellt.

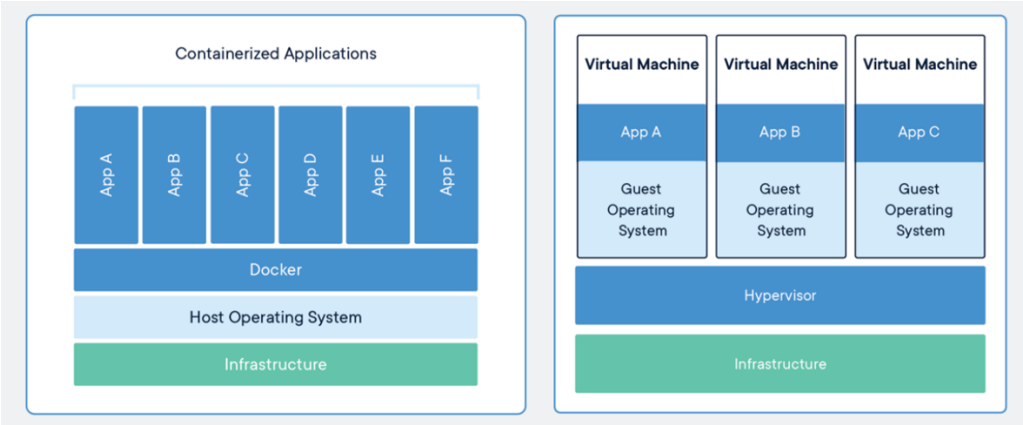


Abbildung 4.4

Der Hypervisor ermöglicht die Ausführung mehrerer virtuellen Maschinen auf einer einzigen Maschine. Jede virtuelle Maschine enthält eine vollständige Kopie eines Betriebssystems, der Anwendung, notwendiger Binärdateien und Bibliotheken. In Abbildung 4.4 sind drei Gastbetriebssysteme zu sehen, die alle von ihrem Hypervisor kontrolliert werden. Jedes System benötigt seine eigenen CPU- und Speicherressourcen sowie seine eigene Kopie verschiedener Binärdateien und Bibliotheken. Dadurch benötigen virtuelle Maschinen im Vergleich zu containerbasierten Anwendungen mehr Speicher.

4.6 Eigenschaften

Als nächstes betrachten wir die häufigst gennanten Eigenschaften von Cloud-Native Architekturen.

1.) Globale Ebene
Cloud-Native Architekturen sind oft für eine globale Ebene ausgelegt. Das impliziert z.B., dass das System mehrfach installiert werden muss und den Einsatz von verteilten Datenbanken.

2.) Skalierbarkeit
Die entstehenden Architekturen sind skalierbar und können eine sehr große Menge von Benutzern unterstützen. Dies ist besonders in Kombination mit der globalen Ebene, wenn man Sychronisation und Konsistenz betrachtet, eine große Herausforderung.

3.) Annahme über Infrastruktur
Die Annahme „infrastructure is fluid“, im Deutschen etwa Infrastruktur ist ständig änderbar, bedeutet, dass die unterliegenden Strukturen nicht konstant sind, wie sie es vergleichsweise bei einem Server sind, der eine bestimmte Anzahl von CPUs hat. So können z.B. Recheneinheiten (CPUs) hinzukommen oder wegfallen. Diese Annahme resultiert aus der Verwendung von Cloud Technologien und bildet die Basis für das Entwerfen von skalierbaren Architekturen.
Die zweite Annahme, dass Fehler konstant auftreten, ergibt sich ebenfalls aus der Verwendung von Cloud Technologien, denn wenn eine große Anzahl von Hard- und Softwarekomponenten verwendet werden steigt die Wahrscheinlichkeit von einem unerwarteten Ausfall. Die Architektur muss also die Möglichkeit von solchen Fehlern miteinbeziehen, denn anders ist sie nicht widerstandsfähig genug, um effektiv in einer Cloud-Umgebung

zu bestehen.

4.) Updates und Tests verlaufen unscheinbar
Die Architekturen sind so entworfen, dass Systeme, ohne Verlust von Verfügbarkeit, geupdatet und getestet werden können. Techniken, die hier zum Einsatz kommen sind unter Anderem CI/CD Pipelines, die den Prozess von Entwicklung bis Installation automatisieren und Immutable Infrastructures ein Ansatz, bei dem bestehende Systeme nicht verändert werden und stattdessen eine neuere Version auf einem neuem System neu installiert wird.

5.) Sicherheit
Sicherheit spielt eine wichtige Rolle in Cloud-Native Architekturen. Die meisten Systeme bestehen aus vielen Komponenten, was eine breite Angriffsfläche bietet. Außerdem ist Authorisierung und Authenifizierung keine einfache Sache in einem entkoppelten verteilten System. Deshalb sollte Sicherheit schon beim Entwurf der Architektur eine Rolle spielen.

4.7 Vor- und Nachteile

In diesem Abschnitt nennen und erklären wir einige Vor- und Nachteile von Cloud-Native Architekturen. Wir beginnen mit den Vorteilen.

1.) Skalierbarkeit/Elastizität
Aus der Kombination von entkoppelten Komponenten, Orchestrierung und Cloud-Infrastruktur entstehen effektiv skalierbare (auch elastische) Architekturen. Orchestrierung spielt dabei eine große Rolle, da ohne sie in größeren Systemen Skalierbarkeit kaum realisierbar ist.

2.) Zuverlässigkeit/Wiederstandsfähigkeit
Cloud-Native Architekturen sind robust gegen Ausfälle der unterliegenden Strukturen. Hierzu werden Komponenten überwacht und bei Fehlern neu gestartet. Komponenten sind dabei oft, auch Zwecks Skalierbarkeit, redundant vorhanden. Auch hat die Orchestrierung eine wichtige Rolle, da sie für die Überwachung verantwortlich ist.

3.) Änderbarkeit/Wartbarkeit
Da Komponenten (wie z.B. Microservices) weitesgehend voneinander unabhängig sind, können leichter Änderungen gemacht werden, ohne andere Komponenten auch ändern zu müssen. Auch können Teile der Architektur unabhängig vom Rest installiert oder ausgestauscht werden

4.) Übertragbarkeit
Durch die Verwendung von Containern und CI/CD Pipelines ist es möglich die Systeme in andere Umgebungen einfach zu installieren. Container ermöglichen dabei eine Uniforme Umgebung für die Komponenten und ist damit unabdingbar für Cloud-Native Architekturen.

5.) Cloud-Native Computing Foundation
Die CNCF ist eine Organisation, die die Entwicklung im Cloud-Native Bereich unterstützt. Sie ist ein zentraler Punkt für Informationen zu Best Practices, Standards, Open-Source Projekte und Neuigkeiten im Bereich Cloud Native.

Die Liste der Nachteile ist zwar kurz, jedoch sind die Punkte hoch zu gewichten.

1.) Komplexität
Wie in der Vorlesung gelernt gilt für das Entwerfen von Software-Architekturen "There is no silver bullet". Dies trifft wahrscheinlich am meisten auf Cloud Native Architekturen zu. Zwar existieren Tools und Herangehensweisen, die in fast allen Cloud Native Architekturen zum Einsatz kommen, jedoch ist die schiere Auswahl an Möglichkeiten, die getroffen werden müssen, bereits eine Herausforderung. Jede Cloud Native Applikation ist ein verteiltes System, welche an sich schon sehr komplex werden können. Addiert man hierzu noch Container/Container-Orchestrierung, Sicherheitsaspekte, Datenhaltung und Skalierung/Lastenverteilung kann sehr schnell der Überblick verloren gehen. Des Weiteren ist eine solche Architektur nicht einfach zu testen, installieren oder upzudaten und es müssen weiter Tools verwendet werden um dies zu bewerkstelligen. Abschließend ist zu sagen, dass die hohe Komplexität aus den hohen Anforderungen (Skalierbarkeit, Erreichbarkeit, Sicherheit, etc.) stammt, die an Cloud Native Architekturen gestellt werden und daher unvermeidbar ist. Die Komplexität ist ein Tradeoff, der bei Cloud Native Architekturen eingegangen wird, da diese in einem Zielkonflikt mit anderen Anforderungen steht, d.h. je höher die Ansprüche desto komplexer gestaltet sich die Architektur.
2.) Neuer Ansatz/Technologie
Die Cloud Native Computing Foundation treibt den Bereich Cloud Native stark voran, dennoch ist es ein relativ neuer Ansatz. Ständig werden neue Innovationen gemacht und bestehende Tools erweitert. Dies bedeutet, dass z.B. Fachliteratur und Erfahrene Entwickler kaum vorhanden sind. Jedoch wird wegen der hohe Komplexität eine hohe Expertise benötigt, was den Cloud-Native Bereich so schwierig macht.

4.8 Einsatzgebiete

Cloud-Native Architekturen werden derzeit meistens für Systeme benutzt, die entweder mit vielen Daten und/oder mit einer großen Anzahl von Benutzern umgehen müssen. Also generell Systeme, die ein hohes Maß an Skalierbarkeit fordern. Besonders in den Bereichen Streaming und Big Data werden häufig Cloud-Native Architekturen verwendet. Beispiele sind der Streaming-Dienst von Netflix sowie Produktivitätsprodukte von Goolge. Mittlerweile werden auch Cloud-Plattformen Angeboten (Goggle Cloud Platform und AWS), die es wiederum ermöglichen Cloud-Native Applikation zu entwicklen. Anzumerken ist, dass viele Unternehmen eine Migration ihrer Dienste in die Cloud vorgenommen haben, da die Möglichkeiten für Cloud basierte Systeme erst im letzten Jahrzent wirklich zu einer Option wurde.

5 Fallbeispiel

Wir haben uns einen Anwendungsfall selbst ausgedacht und dafür eine Architektur entworfen und prototypisch implementiert. In diesem Kapitel stellen wir Anwendungsfall, Architektur und Implementierung vor.

5.1 Beschreibung

Im Rahmen des Beispiels, haben wir eine Cloud-Native Architektur für ein Chatprogramm (ChatApp) entworfen. Die Applikation soll drei Funktionen besitzen. Benutzer sollen sich registrieren, ein-und ausloggen und Nachrichten an andere Benutzer schreiben können.

5.2 Anforderungen

Die Anforderungen sind in nicht-funktionale und funktionale Anforderungen gegliedert, die im folgenden aufgeführt sind.

5.2.1 Nichtfunktionale Anforderungen

1. Skalierbarkeit
Das System muss mit einer großen Zahl von Benutzern, die das System gleichzeit verwenden, umgehen können.

2. Verfügbarkeit
Das System soll eine hohe Verfügbarkeit haben.

3. Sicherheit
Das System muss Berechtigungen überprüfen können (z.B. darf ein Benutzer die Nachrichten, die er abrufen will, lesen) und das System sollte sich gegen übliche Cyberangriffe (z.B. DDoS) schützen können.

4. Änderbarkeit/Erweiterbarkeit
Das System muss es zulassen, dass weitere Komponenten (z.B. Hochladen von Bildern und Videos) einfach hinzugefügt werden können.

5.2.2 Funktionale Anforderungen

1. Registierung
Ein Benutzer kann sich mit seiner E-Mail Adresse und einem Passwort im System registieren.

2. Ein- und Ausloggen
Ein Benutzer kann sich mit seiner E-Mail Adresse und seinem Passwort im System anmelden und danach die Funkionen nutzen bis er sich abmeldet.

3. Nachrichten schreiben/lesen
Ein Benutzer kann Nachrichten an andere Benutzer senden und Nachrichten, die an ihn gesendet worden sind, abrufen.

In Abbildung 5.1 sind die funktionalen Anforderungen in Form eines Use-Case-Diagramms dargestellt.

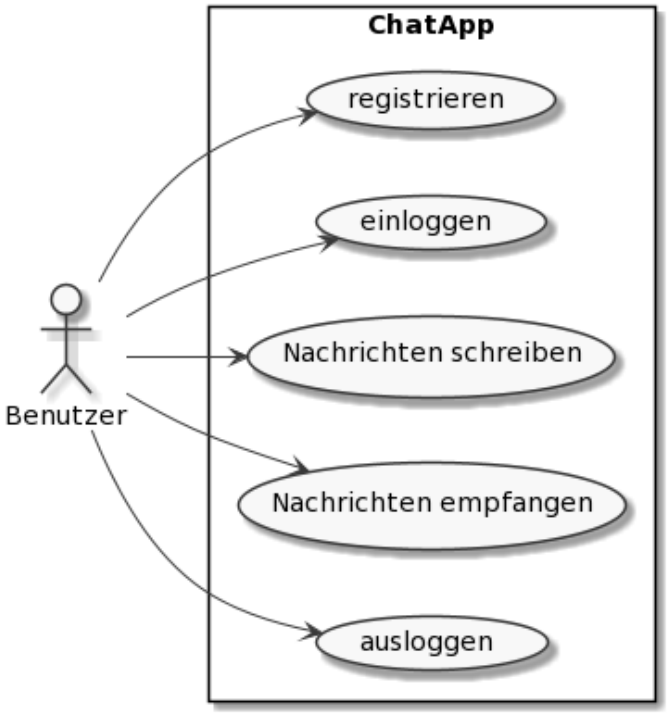


Abbildung 5.1: Use-Case-Diagramm der ChatApp

5.3 Architekturentwurf

Im diesem Abschnitt stellen wir die von uns entworfene Architektur vor und erläutern die wichtigsten Merkmale.

5.3.1 Übersicht

Wie in Abbildung 5.2 zu sehen ist besteht die Architektur aus einem Client mit graphischer Oberfläche (UI) einem API-Gateway, das alle Anfragen von Clienten empfängt und drei Microservices mit je einer eigenen Datenbank, die die Hauptfunktionalitäten des Systems implementieren.

5.3.2 Microservices

Jede Funktion aus den Anforderungen korrespondiert mit einem Microservice. Die Aufteilung der Microservices ergibt sich aus der Annahme, dass die Funkionalitäten, die sie bereitstellen unterschiedlich oft genutzt werden: Anzahl der Registrierungen < Anzahl

Login/Logout < Anzahl der geschriebenen Nachrichten/Abruf von Nachrichten). Daraus gewinnen wir eine individuelle Skalierbarkeit der Funktionen, ohne Redundanz.

5.3.3 Registration-Microservice

Über den Registration-Microservice kann sich ein Benutzer registrieren. Dazu stellt er eine Schnittstelle zur Verfügung an die man eine Nachricht mit E-Mail, Passwort und Land senden kann. Beim Eintreffen einer neuen Nachricht wird ein neuer Eintrag in seiner Datenbank generiert. Des Weiteren wird eine Nachricht an den Login-Service gesendet, mit der Anfrage, auch hier den neuen Benutzer anzulegen.

5.3.4 Login-Microservice

Der Login-Microservice stellt drei verschiedene Funktionen über eine Schnittstelle zur Verfügung: Anlegen eines neuen Benutzers, Einloggen und Ausloggen. Soll ein neuer Benutzer angelegt werden (dies wird vom Registration-Microservice angestoßen), so erzeugt der Microservice einen neuen Eintrag in der Login-Datenbank, in dem E-Mail (auch als ID verwendet) und Passwort des Benutzers gespeichert werden. Will sich ein Benutzer anmelden, gleicht der Microservice die Daten mit der Datenbank ab und stellt dem Benutzer, falls die Kredentialen korrekt waren, einen Token aus, welcher auch in der Datenbank gespeichert wird. Mit diesem Token, kann er sich dann bei anderen Microservices dann authorisieren. Meldet sich ein Benutzer ab, wird der Token einfach aus der Datenbank gelöscht.

5.3.5 Message-Microservice

Die Schnittstelle des Message-Microservices umfasst das Senden einer Nachricht und das Abrufen von empfangenen Nachrichten. Im Fall des Sendens müssen die folgenden Daten angegeben werden: eine Nachricht, eine empfänger E-Mail Adresse und einen Token. Mithilfe des Tokens stellt der Message-Microservice sicher, dass der Sender eine gültige Session hat. Dazu sendet er eine entsprechende Nachricht an den Login-Microservice, der dann den Token bestätigt oder ablehnt. Im positiven Fall wird die Nachricht in der Datenbank des Microservices gespeichert. Will ein Benutzer seine Nachrichten abrufen, so muss er nur einen gültigen Token vorzeigen, den der Message-Microservice, wie beim Senden einer Nachricht, über den Login-Microservice abgleicht.

5.3.6 API-Gateway

Das API-Gateway schottet die Microservices ab. Das bedeutet, dass alle Anfragen vom Clienten nur an das API-Gateway gesendet werden und niemals direkt an die Microservices. Das Gateway leitet dann die Anfrage an den entsprechenden Mircoservice weiter und sendet dessen Antwort wieder an den Clienten. Die Abschottung hat mehrere Vorteile. Sicherheitsrelevante Aspekte wie z.B. DDOS-Protection oder der Schutz vor unbefugtem Zugriff (von Adressen, die nicht zum Client gehören) können auf das API-Gateway verlagert werden und müssen daher nicht in jedem Microservice implementiert werden. Des Weiteren erfüllt das API-Gateway die Rolle eines Load-Balancers, der die Anfragen auf verschiedene Instanzen von Microservices verteilt.

5.3.7 Client

Mithilfe des Clients können Anfragen an das System, genauer gesagt das API-Gateway, gesendet werden. Er enthält keine Funktionalitäten und dient lediglich zum Abrufen und

Darstellen von Inhalten.

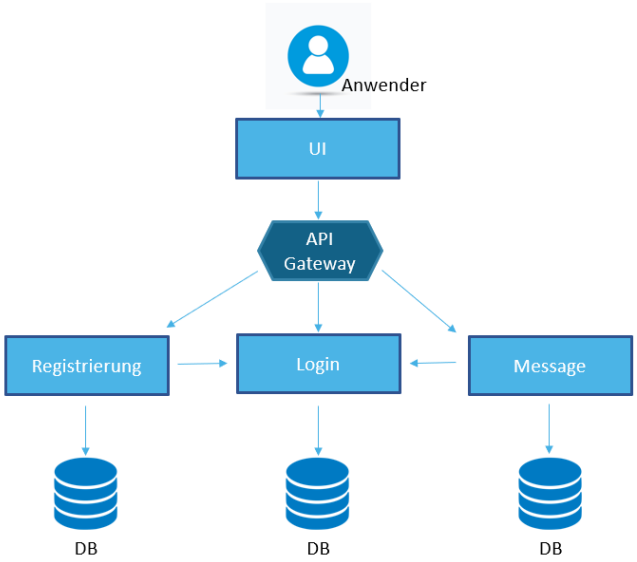


Abbildung 5.2: Architekturentwurf

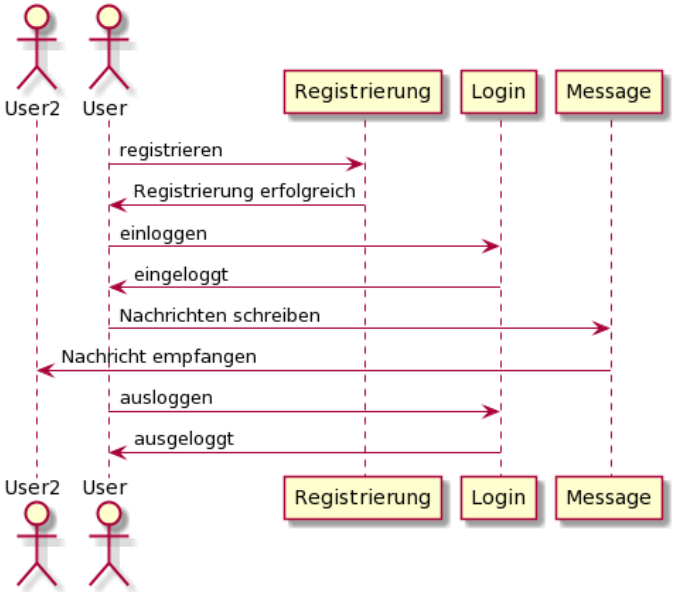


Abbildung 5.3: Sequenzdiagramm

5.4 Implementierung des Prototyps

In diesem Kapitel stellen wir unsere prototypische Implementierung vor. Das gesamte System ist in fünf Teile aufgeteilt (UI, API-Gateway, Registration-Microservice, Login-Microservice und Message-Microservice), wobei jeder Teil einem unabhängigem Projekt entspricht, aus welchem jeweils ein eigenständig ausführbares Programm entsteht. Sämtliche Kommunikation wird über das HTTP Protokoll abgewickelt. Die Inhalte der Nachrichten sind dabei im JSON Format.

5.4.1 UI

Die Benutzeroberfläche ist eine eigenständige React App. ReactJS ist eine Open-Source-JavaScript Bibliothek mit der Webanwendungen erstellt werden können.
Des Weiteren wurde zu dem HTML und JavaScript verwendet. HTML ist eine Auszeichnungssprache, die zur Strukturierung und Erstellung von Webseiten verwendet wird. JavaScript ist für das Verhalten und Aussehen einer Webseite zuständig.
Der Benutzer interagiert mit der ChatApp, indem er sich z.B. in der ChatApp registriert. Die Eingaben werden anhand von HTTP Requests an das API-Gateway geschickt, das die Daten zur Weiterverarbeitung an den entsprechenden Microservices weiterleitet. Empfange Daten werden ebenfalls von dem API-Gateway an die Benutzeroberfläche gesendet, sodass die ChatApp ausschließlich mit dem API-Gateway kommuniziert.

5.4.2 API-Gateway

Das API-Gateway ist mit Java und Maven als Build-Management-Tool implementiert. Es stellt eine HTTP Schnittstelle zur Verfügung, an die Nachrichten gesendet werden können, die es dann an die Microservices weiterleitet. Für das Gateway sind die Microservices ebenfalls über HTTP Schnittstellen erreichbar. Das Loadbalancing ist für den Message-Microservice implementiert. Dabei kennt das API-Gateway zwei verschiedene Endpunkte unter denen jeweils eine Instanz des Message-Microservices auf Anfragen hört. Bekommt das Gateway eine Anfrage für den Message-Microservice wird abwechselnd eine der beiden Instanzen ausgewählt, an welche die Anfrage weitergeleitet wird. Die Wahl von zwei Instanzen ist hier zwecks Einfachheit gewählt. Es ist möglich mehr als nur zwei Instanzen des Message-Microservices oder eines anderen Microservices zu verwenden. Eine Möglichkeit um die Lastenverteiltung zu Regeln wäre das Round-Robin-Verfahren, bei dem die Anfragen der Reihe nach auf die Microservices verteilt werden.

5.4.3 Microservices

Alle Microservices sind mit Java und Maven als Build-Management-Tool implementiert. Die Wahl der Programmiersprache und anderen Tools ist könnte dabei beliebig gewählt sein. Entscheidend ist nur die Schnittstelle die nach außen hin zur Verfügung stellt wird. Des Weiteren sind die Microservices Zustandslos. Für den Registration-Microservice und den Message-Microservice ist dies trivial, da sie sowieso keine temporären Daten speichern müssen. Der Login-Microservice muss sich jedoch den Token merken, den jeder Benutzer zur Authorisierung verwendet. Dies tut er indem er den Token in der Datenbank speichert. Der Einfach halber wurden die Datenbanken für den Registration-Microservice und den Login-Microservice mit einfachen Java Klassen implementiert. Dies war aber für den Message-Microservice nicht möglich, da für ihn immer mindestens zwei Instanzen existieren, welche dann keine gemeinsame Datenquelle hätten. Um zumindest die Lösungsidee zu demonstrieren, haben wir eine einfache Datei als Datenbank verwenden, die für beide Instanzen immer zugänglich ist.

6 Diskussion

In diesem Kapitel diskutieren wird die entworfene Architektur und den dazugehörigen Prototypen.

6.1 Vergleich mit Eigenschaften Cloud Nativ

In diesem Abschnitt sehen wir uns an, welche Cloud-Native Eigenschaften unsere Architektur erfüllt und welche nicht.

1.) Skalierbarkeit
Die Kombination von Microservices und eines Load-Balancers (API-Gateway) erfüllt auf den ersten Blick die Eigenschaft, denn es können Dank des Designs des Message-Microservices mehrere Instanzen gleichzeitig verwendet werden, auf die dann der Load-Balancer die Anfragen gleichmäßig verteilt. Da jedoch der Message-Microservice bei jeder Anfrage überprüfen muss, ob der vom Benutzer bereitgestellte Token gültig ist, wird für jede Anfrage auch eine weitere Anfrage an den Login-Microservice generiert. Diese Abhängigkeit wirkt sich stark negativ auf die Skalierbarkeit aus, da der Effekt von mehreren Message-Microservice Instanzen gleich null ist, wenn nur eine Instanz des Login-Microservices vorhanden ist. Dieses Problem ist Lösbar, indem man ebenfalls vom Login-Microservice mehrere Instanzen laufen lässt und die Anfragen des Message-Microservices an das API-Gateway sendet, welches dann die Anfragen an die Login-Microservice Instanzen verteilt.

2.) Aufgebaut auf der Annahme „infrastructure is fluid and failure is constant“
Hier ist leicht festzustellen, dass die Architektur diese Eigenschaft nicht erfüllt. Es wird nicht berücksichtigt, dass Infrastruktur sich ändern kann oder das Fehler auftreten können. Beide dieser Eigenschaften sind essentiell für Cloud-Native Architekturen, da ohne diese keine Anwendung in einer Cloud-Umgebung bestehen kann. Die Architektur hat jedoch das Potenzial diese Eigenschaften zu erfüllen, indem man z.B. Docker in Kombination mit Kubernetes verwendet.

3.) Updates und Tests verlaufen unscheinbar
Wir können eine neue Version des Systems (API-Gateway und Mircoservices) auf neuen Strukturen installieren und testen, während die alte Version weiter verfügbar ist. Sind Installation und Test abgeschlossen kann der Verkehr von den Clients auf die neue Version geleitet werden. Dieser Ansatz nennt sich Immutable Infrastructure und wäre eine Möglichkeit diese Eigenschaft umzusetzten. Durch den modularen Aufbau der Architekur, können auch einzelne Microservices auf gleiche Weise upgedated werden. Insgesamt erfüllt die Architektur diese Eigenschaft in der Theorie wurde aber nicht in der Praxis ausgiebig getestet.

4.) Sicherheit ist ein Teil der Architektur
Diese Eigenschaft ist erfüllt. Einerseits ist ein Authorisierungsmechanismus mithilfe von Tokens vorhanden, andererseits können im API-Gateway z.B. Logging und Schutz vor

DDoS-Attacken implementiert werden. Es sei gesagt, dass dadurch das System noch weit entfernt ist von einem in der Praxis sicherem System.

5.) Globale Ebene
Die Architektur ist nicht für eine globale Ebene geeignet. Es wäre zwar möglich die Anwendung mehrfach zu installieren, jedoch würden diese Installationen keine Daten teilen. Um dies zu bewerkstelligen müsste man auf verteilte Datenbanken bauen.

6.2 Microservices

Docker, Kubernetes

6.3 Tradeoff

Schon beim Entwurf der Architektur fällt auf, dass der Message-Microservice abhängig vom Login-Microservice ist, da er jede Anfrage authorisieren muss. Skaliert man nun den Message-Mircoservice muss man auch den Login-Microservice hochskalieren, denn ansonsten wird der Login-Microservice zu einem Flaschenhals und man hat keine skalierbares System. Generell wird an der Architektur der Zielkonflikt zwischen Sicherheit und Sklierbarkeit deutlich, da die eingesetzten Sicherheitsmaßnahmen (API-Gateway und Login-Microservice) das System verlangsamen. Jede Anfrage durchläuft eine Sicherheitsüberprüfungen im API-Gateway und muss danach noch von dem jeweiligen Microservice beim Login-Microservice authentifiziert werden.

6.4 TODO

Will sich ein Benutzer registrieren so wird eine Anfrage an den Registration-Microservice generiert, welcher den neuen Benutzer dann anlegt und danach eine Anfrage an den Login-Microservice sendet, damit dort die Kredentialien gespeichert werden. Tritt nun ein Fehler auf nachdem der Benutzer in der Registration-Microservice Datenbank gespeichert wurde, wird kein Eintrag in der Login-Microservice Datenbank erzeugt, sodass der Benutzer registriert ist sich aber nicht einloggen kann. Solch eine Situation ist natürlich zu vermeiden. Es soll an diesem Beispiel deutlich werden, dass nicht nur Orchestrierung für Robustheit und Widerstandsfähigkeit verantwortlich sind, sondern auch jeder Microservice muss dafür Sorge tragen. Eine Lösung in unserem Fall wäre nach dem Ausfall des Login-Microservices einen Rollback zu machen und einen Fehler auszugeben. Ein weiterer Lösungansatz ist, eine andere Instanz des Login-Microservices zu verwenden.

TODO Security

6.5 Erweiterbarkeit

Auf Grund der Microservice-Architektur lässt sich die Anwendung gut erweitern. Dies ist der Fall, da die Anwendung aus verschiedenen Microservices besteht, die größtenteils unabhängig mit einander fungieren.
Wie in Abschnitt 6.3 beschrieben, muss bei einer Skalierung des Message-Services auch der Login-Service berücksichtigt werden. Was die Erweiterbarkeit in diesem Fall einschränkt. Die restlichen Microservices wie z.B. das API-Gateway und die Benutzeroberfläche laufen

unabhängig von einander. Diese können erweitert werden, ohne die Funktionalität der anderen Services zu beeinträchtigen. TODO sonst unabhängig??? TODO

7 Fazit

In diesem Kapitel ziehen wir ein Fazit aus der Seminararbeit.

Ziel war es sich mit dem Thema Cloud Native Architekturen auseinander zu setzten und prototypisch eine solche Architektur zu implementieren. ...

Literatur

[1] Jon Bentley. *Programming Pearls*. Addison–Wesley, 1999.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest und Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2001.

[3] Gunter Dueck. *Duecks Trilogie: Omnisophie – Supramanie – Topothesie*. `http://www.omnisophie.com`. Springer, 2005.

[4] Donald E. Knuth. „Big Omicron and Big Omega and Big Theta“. In: *SIGACT News* 8.2 (1976), S. 18–24.

[5] Donald E. Knuth. „Computer Programming as an Art“. In: *Communications of the ACM* 17.12 (1974), S. 667–673.

[6] Ian Sommerville. *Software Engineering*. Boston, MA, USA: Addison-Wesley, 1992.

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

WLAN Wireless Local Area Network

TCP Transmission Control Protocol

GoF Gang of Four

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