Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Konzept und Implementierung einer Lösungssprache und eines Lösungsrepositories für Cloud Computing Patterns

Martin Beisel

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Dr. h. c. Frank Leymann
Betreuer/in:	Michael Falkenthal M. Sc.
Beginn am:	4. Januar 2017
Beendet am:	4. Juli 2017
CR-Nummer:	D.2.9 , D.2.11

Kurzfassung

Muster (engl. Pattern) und Mustersprachen sind inzwischen ein allgegenwärtiges Konzept, um häufig wiederkehrende Probleme und Aufgaben systematisch einzuordnen, zu lösen und in Zusammenhang zu setzen. Muster fassen dabei die abstrakte Lösung eines Problems in einem vorher definiertem Format auf. Wichtig ist dabei, dass die Lösung des Problems möglichst generisch ist, sodass mit Hilfe des Musters das zugrundeliegende Probleme verstanden werden kann und somit eine eigene Lösung erarbeitet werden kann. Folglich beschreibt ein Muster zwar eine abstrakte Lösung, es können dabei jedoch nahezu unbegrenzt viele verschiedene konkrete Lösungen gefunden werden. Die Verwendung von Mustern ist zudem nicht auf bestimmte Fachgebiete beschränkt, sondern ist ein allgegenwärtiges Verfahren, welches Konstrukteuren eine Lösungsgrundlage für ihre Probleme bietet. Der bei der Verwendung von Mustern entstehende Nachteil ist jedoch, dass Muster zwar eine Lösung vorstellen, diese aber nur abstrakt und nicht konkret vorliegt. Somit muss bei jeder Anwendung eines Musters, die konkrete Lösung neu entwickelt werden. Ziel dieser Arbeit ist es, ein Konzept zu finden und umzusetzen, welches diesen Nachteil der Mustersprachen am Beispiel der Cloud Computing Patterns löst. Derzeit ist es möglich, nach einzelnen Mustern zu suchen und basierend auf semantischen Links zu weiteren relevanten Mustern zu navigieren. Die Muster und ihre Verknüpfungen bilden eine Mustersprache. Eine gegenwärtige Limitation ist jedoch, dass alle Muster lediglich eine abstrakte Lösungsbeschreibung beinhalten und keine konkreten und für die Implementierung direkt wiederverwendbaren Lösungsumsetzungen enthalten. Weiterhin fehlt die Möglichkeit, solche konkreten Lösungsumsetzungen entsprechend der Prinzipien einer Mustersprache effizient mit semantischen Links zu verbinden, um eine Lösungssprache zu erstellen. Entsprechend gilt es in dieser Arbeit eine Lösungssprache zu konzipieren und ein Lösungsrepository für diese zu implementieren.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1 Einleitung

In vielen Bereichen wird Wissen über häufig wiederkehrende Probleme mit Hilfe von Mustern aufbereitet. Dabei geht die Verwendung von Mustern auf Alexanders Veröffentlichung [AIS77] im Jahr 1977 zurück, welche eine Mustersprache für den Bau von Städten und Gebäuden beschreibt. Dabei enthält ein einzelnes Muster nur einen kleinen Teil des Wissens einer ganzen Mustersprache, sodass ein einzelnes Muster einfach zu verstehen und anzuwenden ist. Besonders wichtig ist es dabei, dass jedes Muster mit den anderen Mustern verknüpft ist, sodass eine Mustersprache entsteht, welche eine unbegrenzte Anzahl an möglichen Muster-Kombination ermöglicht [AIS77]. Des Weiteren sollen Probleme und deren Lösungen so dargestellt werden, dass jedes Muster unterschiedlich beurteilt und entsprechend modifiziert werden kann, ohne dass der Kern des Musters verloren geht[AIS77]. Nach der Anwendung von Mustern im Bausektor wurde die Verwendung von Mustern in der Informatik immer populärer. Ein bekanntes Beispiel dafür sind die Muster der Objekt-orientierten-Programmierung[GHJV94], welche auch bei der Umsetzung des in Kapitel 3 beschriebenen Repositories zum Einsatz kamen. Die für diese Arbeit zentrale Mustersprache ist die von Fehlingt et al. [FLR+14a] definierte Cloud Computing Pattern Language, welche einen umfassenden Einblick in das Cloud Computing bietet und dessen Konzepte in Mustern beschreibt. Die Cloud Computing Pattern Language bietet eine Vielzahl von Verknüpfungen zwischen den verschiedenen Mustern und bietet abstrakte Lösungen für diese. Da ein Nutzer für häufig wiederkehrende Probleme nicht jedes mal selbst eine neue Lösung erstellen will, fehlt der Mustersprache ein wichtiger Teil. Es gibt keine konkreten Umsetzungen für die beschriebenen Muster und demzufolge auch keine Verknüpfungen zwischen den unterschiedlichen Lösungen.

Ziel dieser Arbeit ist es, dieses Problem zu lösen, indem eine neue Lösungssprache konzeptioniert wird. Dafür ist es notwendig, dass eine Vielzahl von Lösungen für das gleiche Muster erstellt werden können, zwischen welchen differenziert und navigiert werden kann. Des Weiteren muss es möglich sein, sowohl zwischen zusammenhängenden Lösungen zu navigieren, als auch von den konkreten Lösungen wieder auf das abstrakte Muster zu kommen.

Struktur

Im folgenden Abschnitt wird die Struktur dieser Arbeit erläutert.

Kapitel 2 – Grundlagen: Hier werden die zum Verständnis dieser Arbeit benötigten Grundlagen und Begriffe erläutert. Es wird ein Überblick über Muster, Lösungen, Muster und

Lösungssprachen, als auch deren Anwendungen in Stories gegeben. Des Weiteren beinhaltet dieser Abschnitt eine Einführung in die Grundlagen des Cloud Computings, sodass das Anwendungsbeispiel nachvollzogen werden kann.

Kapitel 3 – Konzept Im zweiten Kapitel wird sowohl auf das Konzept, der für dieses Projekt erstellten Lösungssprache, als auch auf die konzeptionellen Anforderungen an das Lösungsrepository eingegangen.

Kapitel 4 – Umsetzung In diesem Kapitel werden die in Kapitel 3 vorgestellten Konzepte in eine konkrete Umsetzung überführt. Dabei wird zuerst auf die entstehenden technischen Anforderungen eingegangen, aufgrund derer dann die Technologieauswahl erfolgt. Des Weiteren wird die Architektur beschrieben auf der, der Prototyp basiert, welche die zuvor definiert Lösungssprache umsetzt.

Kapitel 5 – Zusammenfassung und Ausblick Im letzten Kapitel dieser Ausarbeitung werden die Ergebnisse zusammengefasst und mögliche Anknüpfungspunkte vorgestellt.

2 Grundlagen

In diesem Kapitel werden die zum Verständnis dieser Arbeit benötigten Grundlagen und Begriffe erläutert. Es wird ein Überblick über Muster, Lösungen, Muster und Lösungssprachen als auch deren Anwendungen in Stories gegeben. Des Weiteren beinhaltet dieser Abschnitt eine Einführung in die Grundlagen des Cloud Computings, sodass das Anwendungsbeispiel nachvollzogen werden kann.

2.1 Muster

Muster fassen ein häufig wiederkehrendes Problem auf und beschreiben dessen Lösung auf eine abstrakte Weise. Sie erklären, wie ein Problem gelöst werden soll, welche Ideen dahinter stecken und warum der erläuterte Lösungsweg anderen Lösungswegen vorzuziehen ist. Muster helfen also das zugrunde liegende Problem zu verstehen und eine konkrete Lösung zu finden, welche jedoch nicht vom Muster selbst definiert wird. Ein Muster erlaubt eine nahezu unbegrenzte Anzahl an konkreten Lösungen für das vorliegende Problem[AIS77]. Der Grundbegriff der Muster entstand im Bausektor[AIS77] wurde jedoch schon bald auf andere Sektoren, insbesondere im Informatik-Sektor übernommen [GHJV94] [FLR+14a]. Eine wichtige Charakteristik von Mustern ist, dass sie mit anderen Mustern kombiniert werden können, welche dann eine Mustersprache (siehe Kapitel 2.3) bilden. Muster die zu einer Mustersprache vereint werden sollen müssen nach dem gleichen Standard formatiert sein, sodass keine Probleme bei der Kombination der Muster entstehen.

2.2 Lösungen

Eine Lösung beschreibt die konkrete Umsetzung einer von einem Muster abstrakt definierten Lösung. Dabei ist zu beachten, dass es nicht genau eine richtige konkrete Lösungsumsetzung für ein Muster gibt, sondern in der Theorie unbegrenzt viele Lösungen möglich sind. Um verschiedene Lösungen zu kombinieren, müssen auch diese nach dem gleichen Standard formatiert sein. So können verschiedene, zusammenpassende Lösungen zu einer Lösungssprache (siehe Kapitel 2.3) kombiniert werden, welche eine Navigation zwischen den einzelnen Lösungen ermöglicht. Somit lassen sich umfassende Lösungen für Stories (siehe Kapitel 2.4) finden, welche mehrere Lösungen kombinieren.

2.3 Muster/Lösungs-Sprachen

Eine Mustersprache umfasst eine Menge von einheitlich standardisierten Mustern, welche miteinander verknüpft sind. Die Verknüpfungen ermöglichen eine Navigation zwischen den verschiedenen Mustern und geben einen Überblick darüber, welche Muster häufig in Kombination verwendet werden und womöglich bei der Umsetzung eines anderen Musters helfen können. Innerhalb der Mustersprache gibt es nahezu unendlich viele Möglichkeiten die einzelnen Muster zu kombinieren, um die unterschiedlichsten Probleme zu lösen [AIS77]. Lösungssprachen stellen konkrete Lösungen für Probleme dar, die Organisation der Lösungen geschieht dabei analog zur Organisation der Muster einer Mustersprache. Genau wie bei der Mustersprache erlaubt eine Lösungssprache die Navigation zwischen den verschiedenen Lösungen der Lösungssprache. Da verschiedene Lösungen jedoch nicht immer einfach ohne weiteren Aufwand kombiniert werden können verfügt die in diesem Projekt verwendete Mustersprache über *semantische Links*, in welchen erläutert wird, wie der Kombinationsprozess vonstatten gehen muss. Der Aufbau, der in diesem Projekt verwendeten Lösungssprache ist an Falkenthals Ausführung *Easing Pattern Application by Means of Solution Languages* [FL17] angelehnt, in welcher auch das zuvor genannte Konzept zur Verwaltung einer Lösungssprache vorgestellt wurde.

2.4 Stories

Eine Story beschreibt ein in der Realität vorkommendes Problem, welches zu groß ist, um es mit einem einzigen Muster zu beschreiben. Das Problem wird also in mehrere Unterprobleme aufgeteilt. Ziel ist es, diese Unterprobleme mit Hilfe von Mustern zu lösen und die Lösungen zu einer umfassenden Lösung zusammenzufassen, die das Problem der Story löst.

2.5 Cloud Computing

Cloud Computing beschreibt die Bereitstellung von IT-Infrastruktur über das Internet. Dafür stellt ein Anbieter (engl. Provider) einen großen Pool aus IT-Ressourcen zur Verfügung, auf welche ein Nutzer zugreifen kann. Somit kann ein Nutzer ohne selbst eine große Menge an IT-Infrastruktur zu besitzen diese benutzen. Gemäß NIST[MG11] gibt es hierzu fünf essentielle Eigenschaften die erfüllt sein müssen, damit eine IT-Ressource eine Cloud ist.

1. ***On-demand self-service*** Der Nutzer kann selbständig, je nach Bedarf Ressourcen anfordern, nutzen und freigeben.
2. ***Broad network access*** Die Ressourcen sind mittels eines Breitband Netzwerks jederzeit, von überall erreichbar.

3. ***Resource pooling*** Die Ressourcen des Anbieters sind so zusammengelegt, dass mehrere Nutzer auf den gleichen ortsunabhängigen Ressourcenpool zugreifen.
4. ***Rapid elasticity*** Ressourcen können elastisch benutzt und freigegeben werden. Dies führt dazu, dass der Nutzer scheinbar unbegrenzte Ressourcen zur Verfügung hat.
5. ***Measured service*** Die Nutzung der Ressource kann transparent gemessen werden. Dies erlaubt ein *pay-per-use* Modell, bei welchem der Verbraucher genau so viel zahlt wie er verbraucht.

Um eine Anwendung in der Cloud optimal betreiben zu können genügt es jedoch nicht sie einfach in einer Cloud zu deployen. Viel mehr muss eine Anwendung, um in einer Cloud sinnvoll eingesetzt werden zu können für diese optimiert werden. Eine Anwendung, welche für eine Cloud Computing Architektur optimiert wurde nennt sich *Native Cloud Application* und muss die folgenden Kriterien erfüllen, um **IDEAL** zu sein [FLR+14a][Rou14].

1. **I**solated state: Ein Konzept bei dem der Application State in einem kleinen Teil der Anwendung isoliert wird und der restliche große Teil der Anwendung stateless ist. Der Vorteil dabei ist, dass *Stateless Components* ohne die Befürchtung von größerem Datenverlust ausgetauscht werden können, da die Daten nicht innerhalb der Komponente gespeichert werden.
2. **D**istribution: Cloud Anwendungen müssen in mehrere Komponenten aufgeteilt werden, sodass die Komponenten auf unterschiedlich Ressourcen verteilt werden können. Eine häufige Aufteilung ist die Aufspaltung der Anwendung in eine User Interface, in eine Processing und in eine Storage Komponente.
3. **E**lasticity: Eine Cloud Anwendung muss horizontal skalierbar sein. Das heißt die Anwendung wird auf einer unterschiedlichen Anzahl von unabhängigen Ressourcen ausgeführt. Somit muss nicht, wie beim vertikalen Skalieren, eine größere Ressource verwendet werden, sondern es können lediglich mehrere kleine Ressourcen hinzugefügt werden.
4. **A**utomated management: Aufgrund der Elastizität der Cloud Anwendung muss es möglich sein, während der Laufzeit automatisch neue Ressourcen anzufordern oder zu entfernen. Dies ist nicht nur wichtig, um die Skalierbarkeit zu gewährleisten, sondern auch, weil viele Cloud Anbieter nicht garantieren, dass eine einzelne Ressource dauerhaft verfügbar ist. Demzufolge muss die Anwendung, wenn eine verwendete Ressource entfernt wird, auf eine andere Ressource verlagert werden können.
5. **L**oose coupling: Da sich die Anzahl der Ressourcen, auf welche eine Cloud Anwendung zugreift, regelmäßig ändert, ist es wichtig, dass die Abhängigkeiten zwischen den Komponenten möglichst gering sind. Dies vereinfacht die Skalierung und erhöht die Robustheit, da der Einfluss von fehlerhaften Komponenten reduziert wird.

Um diese Optimierung und Restrukturierung der Anwendung möglichst effizient und einfach durchzuführen, haben Fehling et al. [FLR+14a] die Cloud Computing Patterns veröffentlicht,

welche sich mit genau dieser Problemstellung befassen und Muster für die vorkommenden Probleme definieren.

2.6 Cloud Computing Patterns

In diesem Abschnitt wird die von Fehling et al. im Buch *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications* [FLR+14a] verwendete Mustersprache vorgestellt. Die aufgeführten Muster sind alle auf eine abstrakte Weise beschrieben, welche sie nicht für bestimmte Cloud Computing Provider wie AWS[Ama17c] oder Azure[Mic17] limitiert. Ein von Fehling definiertes Cloud Computing Pattern erfüllt dabei stets folgende Struktur:

- ***Pattern Name*** Der Pattern Name wird verwendet, um ein Pattern zu identifizieren.
- ***Intent*** Der Intent beschreibt kurz und knapp den Zweck und das Ziel des Patterns.
- ***Driving Question*** Die Driving Question fasst kurz zusammen, welches Problem von dem behandelten Pattern gelöst wird. Somit kann der Leser überprüfen ob das Pattern die Lösung für das gesuchte Problem liefert.
- ***Icon*** Jedes Pattern hat ein eindeutiges Icon, welches verwendet werden kann, um das Pattern in Architekturdiagrammen oder anderen graphischen Repräsentationen darzustellen.
- ***Context*** Der Context beschreibt, unter welchen Umständen und mit welcher Umgebung es zu dem zu lösenden Problem kommen kann. Außerdem wird erläutert, warum andere, scheinbar triviale Lösungen suboptimal oder fehlerhaft sind.
- ***Solution*** Die Solution beschreibt, wie das Pattern die in der Driving Question genannte Frage löst.
- ***Result*** Im Result wird beschrieben, wie sich ein Programm bei der Umsetzung des Patterns verhält und welche neuen Herausforderungen entstehen können.
- ***Variations*** Beinhaltet verschiedene kleine Variationen des Patterns. Durch das Variieren von Patterns wird vermieden, dass die Gesamtzahl der Patterns stark steigt und die Patternsprache unübersichtlich wird.
- ***Related Patterns*** In den Related Patterns werden alle Patterns aufgelistet, welche häufig im Zusammenhang mit dem behandelten Pattern verwendet werden.
- ***Known Uses*** Die Known Uses beinhalten eine Liste bereits existierender Produkten und Anwendungen, die das Pattern verwenden oder damit im Zusammenhang stehen.

3 Konzept

In diesem Kapitel wird sowohl auf das Konzept, der für dieses Projekt erstellten Lösungssprache, als auch auf die konzeptionellen Anforderungen an das Lösungsrepository eingegangen.

3.1 Lösungssprache

In diesem Abschnitt wird beschrieben, wie die Kozeptionierung der in diesem Projekt entstandenen Lösungssprache entwickelt wurde und welche Ideen bei der Entwicklung eingeflossen sind. Als Grundlage für die Begriffe Muster, Lösung und Muster/Lösungssprache gelten die in Kapitel 2 definierten Begriffe. Ziel ist es, eine Lösungssprache für die von Fehling et al. [FLR+14a] definierte Mustersprache zu entwickeln, welche es ermöglicht konkrete Umsetzungen für bestimmte Muster strukturiert aufzubereiten. Dabei stellen sich mehrere Probleme, die im folgenden nacheinander gelöst werden.

3.1.1 Verknüpfungen zwischen der Mustersprache und der Lösungssprache

Die erste Frage, die sich bei der Erstellung einer Lösungssprache stellt, ist, wie man den Zusammenhang zu der zugehörigen Mustersprache herstellt und die Lösungssprache kompatibel zur Mustersprache gestaltet. Dabei ist es von höchster Wichtigkeit, alle Eigenschaften von Mustern und Lösungen zu beachten.

- Ein Muster ist abstrakt und hat in der Theorie unbegrenzt viele mögliche konkrete Lösungen. Somit entsteht eine 1..n Beziehung
- Eine Mustersprache verfügt über nahezu unbegrenzt viele Möglichkeiten Muster miteinander zu kombinieren und zu verbinden.
- Eine Lösungssprache kann unterschiedlich viele Lösungen je Muster haben.
- Ein Muster hat keine konkrete Umsetzung. Ein Beispiel dafür in Fehlings Patternsprache[FLR+14a] sind die unterschiedlichen Workloads. Sie beschreiben viel mehr einen Zustand und empfehlen ein Umsetzungsmodell, welches unter Umständen gar nicht Cloud Computing sein muss (bspw. Static Workload [FLR+14h]).

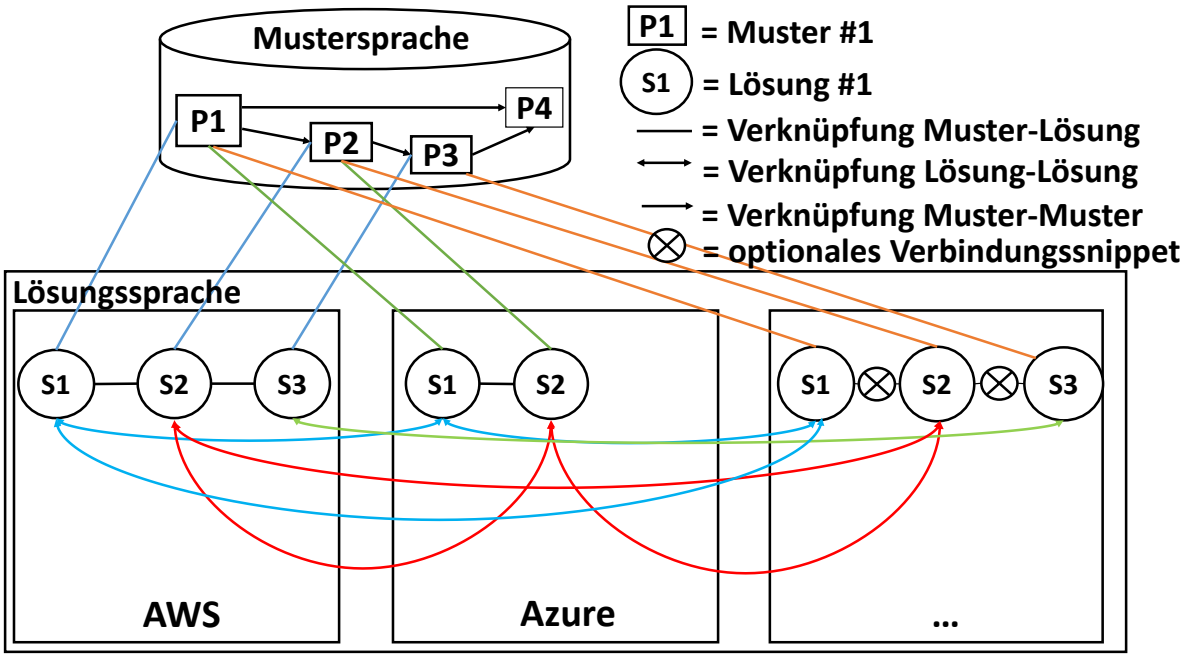

Abbildung 3.1: Zusammenhang zwischen Mustersprache und Lösungssprache

Wie in Abbildung 3.1 dargestellt, ist es möglich, die Verknüpfungen zwischen einer Mustersprache und einer Lösungssprache so zu gestalten, dass keine der oben aufgeführten Eigenschaften verletzt wird. Die in der Darstellung beispielhaft aufgeführte Mustersprache enthält 4 Muster die miteinander im Zusammenhang stehen. Für die Muster P1, P2 und P3 gibt es mindestens eine konkrete Umsetzung für einen Lösungstyp. Sowohl die Mustersprache als auch die Lösungssprache erlauben es zwischen den zusammengehörigen Mustern bzw. Lösungen zu navigieren. Die Lösungssprache hat zusätzlich zu den Verknüpfungen der Lösungen innerhalb des gleichen Lösungstyps noch Verbindungen zu allen alternativen Lösungen, die das gleiche Muster lösen. Somit ist eine einfache Navigation zwischen den unterschiedlichen Lösungen möglich.

3.1.2 Eigenschaften der Lösungen

Nachdem nun klar ist, wie die Verknüpfungen zwischen der Mustersprache und der Lösungssprache sind bleibt die Frage, wie die Definition der eigentlichen Lösung aussieht. Dafür muss überlegt werden, welchen Zweck die Lösung hat, wie sie im Zusammenhang zu dem entsprechenden Muster steht und welche Attribute sie umfasst. Im Folgenden wird eine mögliche Umsetzung einer Lösungssprache für Fehlings Cloud Computing Mustersprache[FLR+14a] vorgestellt und erläutert.

- ***ID*** Die ID wird verwendet, um ein Pattern zu identifizieren. Die Verwendung des Namens ist hier nicht möglich, da es unbegrenzt viele Lösungen pro Muster geben kann und diese unter Umständen gleich heißen.
- ***Solutionname*** Der Name wird in Kombination mit dem Typ zur Erkennung des Musters für den Nutzer verwendet.
- ***Patternname*** Hier wird der Name des zur Lösung zugehörigen Musters dargestellt.
- ***Solutiontype*** Entspricht dem Typ der Lösung. Für Cloud Computing Muster könnten dies beispielsweise AWS[Ama17c] oder Azure[Mic17] sein.
- ***Other Solutiontypes*** Hier werden andere Lösungen verknüpft, welche das gleiche Muster lösen.
- ***JSON-Snippet*** Enthält ein JSON-Snippet, welches das Muster löst - es stellt eine konkrete Lösung dar.
- ***YAML-Snippet*** Enthält ein YAML-Snippet, welches das Muster löst - es stellt eine konkrete Lösung dar.
- ***Related Solutions*** Hier werden andere Lösungen verknüpft, welche häufig im Zusammenhang mit dieser Lösung verwendet werden.
- ***Aggregations*** Hier werden die IDs der zu einer Lösung gehörigen semantischen Links gespeichert.

3.1.3 Eigenschaften der semantischen Links

Ähnlich wie bei der Konzeptionierung der Lösungssprache, muss auch bei der Definition der semantischen Links darauf geachtet werden, dass alle zur vollständigen Darstellung notwendigen Attribute enthalten sind. Um semantische Links im Zusammenhang mit Fehlings Cloud Computing Mustersprache [FLR+14a] zu verwenden werden folgende Attribute benötigt:

- ***ID*** Die ID wird verwendet, um einen semantischen Link zu identifizieren.
- ***Name*** Der Name wird zur Erkennung des semantischen Links verwendet.
- ***Description*** In der Description wird erklärt, was genau gemacht werden muss, um die zugehörigen Lösungen erfolgreich miteinander zu verknüpfen. Eventuell zugehörige Snippets werden jedoch in *Snippets* und nicht in der *Description* gespeichert.
- ***Snippets*** Das Snippet Attribute beinhaltet die für die Umsetzung der Verknüpfung notwendigen Codeabschnitte.
- ***Solutions*** In den Solutions werden die zum semantischen Link zugehörigen Lösungs-IDs gespeichert.

3.2 Konzeptionelle Anforderungen an das Lösungsrepository

In diesem Abschnitt werden die konzeptionellen Anforderungen, die an das Lösungsrepository gestellt werden, definiert.

Das Lösungsrepository:

- muss die in Abschnitt 3.1 definierte Lösungssprache und die semantischen Links korrekt darstellen können.
- erlaubt es, die definierte konkreten Lösungen miteinander und mit Mustern zu verlinken. Dies kann sowohl über Direktlinks als auch über semantische Verlinkungen geschehen.
- ermöglicht das Hinzufügen von neuen und das Editieren und Löschen von bereits bestehenden Lösungen und semantischen Links.
- erlaubt die Umsetzung der in Kapitel 3.3.1 und 3.3.2 beschriebenen Stories.
- muss mit der Apache v2.0-Lizenz kompatibel sein.

3.3 Konzeptionierung der Stories

In diesem Abschnitt werden die für dieses Projekt relevanten Stories besprochen. Das Lösungsrepository wird insbesondere für diese ausgelegt und optimiert. Das heißt, dass das Lösungsrepository alle für die Story relevanten Informationen und Verknüpfungen beinhalten und darstellen können muss. Die Stories orientieren sich an Fehlings Cloud Computing Mustern[FLR+14a] und stellen häufig vorkommende Problemfälle dar, die bei der Konzeptionierung einer Cloud Computing Anwendung vorkommen.

3.3.1 Story 1: Konzeptionierung einer Native Cloud Application

Das wohl am häufigsten wiederkehrende Szenario ist die Konzeptionierung einer Native Cloud Application(im Folgenden auch NCA). Als NCA wird ein Programm bezeichnet, welches den in Kapitel 2.5 genannten Anforderungen entspricht. Anhand Fehlings Patterns[FLR+14a] lässt sich eine Kombination von Mustern herausarbeiten, welche es ermöglicht, alle Kriterien einer NCA umzusetzen. Als erstes muss die Anwendung mittels des *Distributed Application* Musters [FLR+14b] in mehrere Komponenten aufgeteilt werden, welche, wie in Muster *Loose Coupling* [FLR+14d] beschrieben, mittels einer *Message-oriented Middleware* [FLR+14e] verknüpft werden. Im letzten Schritt wird schließlich noch der Component State ausgelagert, um wie im Muster *Stateless Component* [FLR+14g] beschrieben, die Elastizität und die Robustheit der Anwendung zu erhöhen. Da sich dieses Projekt allerdings nicht mit der Konzeptionierung eines

Muster	AWS Lösung
Message-Oriented Middleware	Simple Queue Service (SQS) oder Simple Notification Service (SNS)
Exactly-once Delivery	SQS-Standardwarteschlange
At-least-once Delivery	SQS-FIFO-Warteschlange
Timeout-based Delivery	SQS

Tabelle 3.1: Message-Oriented Middleware Lösungen

Muster-Repositories, sondern mit der eines Lösungs-Repositories befasst, muss im nächsten Schritt herausgearbeitet werden, wie die zuvor genannten Muster konkret umgesetzt werden. Beispielhaft wird hierfür eine Umsetzung mittel AWS(Amazon Web Services)[Ama17c] betrachtet, wobei davon ausgegangen wird, dass bereits eine entsprechende AWS-Ressource vorhanden ist, welche mit den Cloud Computing Mustern optimiert werden soll. Da es sich bei der Umsetzung des "Distributed Application" Musters um eine Entwurfsentscheidung handelt, kann hierfür kein Code-Snippet verwendet werden. Es wird vielmehr davon ausgegangen, dass es sich bereits um eine Anwendung handelt, bei der die unterschiedlichen Funktionalitäten bereits auf verschiedene Komponenten verteilt wurde. In der Lösungsbeschreibung des *Loose Coupling* Musters schlägt Fehling [FLR+14d] eine Umsetzung mittels einer Message-Oriented Middleware vor. Die Message-Oriented Middleware wird in einem eigenen Muster genauer beschrieben und erläutert, wie eine asynchrone Kommunikation mittels Queues und des Publish-Subscribe Verfahrens gelöst wird. Dabei wird zwischen *Exactly once, At least once, Timout based* Delivery unterschieden. Diese Muster werden mit den in Tabelle 3.1 dargestellten Lösungen in AWS umgesetzt. Amazon Simple Queue Service (SQS)[Ama17b] ist ein vollständig verwalteter Nachrichtenwarteschlangen Service, der eine einfache Entkopplung und Koordination der Komponenten einer Cloud Anwendung ermöglicht. Er ermöglich das asynchrone Versenden von beliebig vielen und großen Nachrichten zwischen den einzelnen Komponenten und garantiert dabei, dass keine Nachrichten verloren gehen und jede Nachricht mindestens einmal zugestellt wird. Eine SQS-Standardwarteschlange garantiert dabei mindestens eine Nachrichtenzustellung, eine SQS-FIFO-Warteschlange garantiert eine genau einmalige Zustellung jeder Nachricht. Als Alternative bietet Amazon noch den Simple Notification Service (SNS)[Ama17a] an, dieser ermöglicht das Versenden von Push-Nachrichten. Dieses Verfahren wird hauptsächlich verwendet, um Nachrichten an eine große Anzahl von abonnierenden Endpunkten und Clients zu senden. AWS[Ama17c] bietet für eine schnelle und einfache Integration ihrer Lösungsumsetzungen JSON-[20113] und YAML[Eva11]-Snippets an, mit Hilfe derer die entsprechende Komponente aufgesetzt werden kann. Da die Snippets von Grund auf sehr ähnlich sind, wird im Weiteren lediglich auf die JSON-Snippets eingegangen. Die Snippets werden im AWS User-Guide dargestellt und erläutert. Die Erklärung umfasst eine kurze Beschreibung des Snippets, die Typ-Definitionen und Notwendigkeit der einzelnen Attribute und eine Beschreibung der Attribute, welche in vielen Fällen einen vorgegeben Wertebereich für das jeweilige Attribut enthält. Für die zuvor beschriebene SQS Lösung befindet

sich im User-Guide das im Listing 3.1 dargestellte Snippet, welches Seitenverweise zu den entsprechenden Attributerläuterungen und Typdefinitionen beinhaltet.

```
{
  "Type" :"AWS::SQS::Queue",
  "Properties" :{
    "DelaySeconds (p. 1009)": Integer,
    "MaximumMessageSize (p. 1009)": Integer,
    "MessageRetentionPeriod (p. 1010)": Integer,
    "QueueName (p. 1010)": String,
    "ReceiveMessageWaitTimeSeconds (p. 1010)": Integer,
    "RedrivePolicy (p. 1010)": RedrivePolicy,
    "VisibilityTimeout (p. 1010)": Integer
  }
}
```

Listing 3.1: SQS Template mit Typ-Beschreibung und Referenzierung

Des Weiteren wird in Listing 3.2 eine konkrete Umsetzung für das zuvor präsentierte Template dargestellt. In diesem Projekt sollen für die im Lösungsrepository präsentierten Snippets konkrete Lösungen dargestellt werden, in denen jedoch deutlich gemacht wird, welche Attribute abhängig von der Anwendung noch angepasst werden müssen. Es sollen also keine Seitenreferenzen oder Typdefinitionen enthalten sein.

```
{
  "Type" :"AWS::SQS::Queue",
  "Properties" :{
    "DelaySeconds": 1,
    "MaximumMessageSize": 4096,
    "MessageRetentionPeriod": 3600
  }
}
```

Listing 3.2: SQS konkrete Umsetzung

Für das letzte Muster der Story gibt es erneut kein konkretes Snippet, da die Anwendung so programmiert werden muss, dass der Zustand einer Komponente nicht direkt in dieser Komponente gespeichert wird, sondern extern in den versendeten Nachrichten oder in einer Datenbank.

3.3.2 Story 2: Erweiterung einer Native Cloud Application

Da wie in Kapitel 3.3.1 beschrieben anhand von Fehlings Muster und mit Hilfe der AWS Snippets die Grundanforderungen einer NCA erfüllt sind, gilt es nun die Vorteile einer NCA zu nutzen und das Grundgerüst zu optimieren. Ziel ist es, wie im *Elastic Load Balancer* Muster [FLR+14c] dargestellt, die einzelnen Komponenten automatisch, abhängig von den benötigten Ressourcen skalieren zu lassen. Dazu sind bei einer AWS Umsetzung ein *Elastic Load Balancer* und eine *Autoscaling* Komponente notwendig. Um das Autoscaling optimal umzusetzen, wird zudem eine *ScalingPolicy* definiert, welche für die entsprechende Anwendung optimiert ist. Des Weiteren soll das *Watchdog* [FLR+14i] und das *Resiliency Managment Process* Muster [FLR+14f] umgesetzt werden um zu garantieren, dass fehlerhafte Komponenten automatisch erkannt und ausgetauscht werden. Die AWS Umsetzung für die Erkennung fehlerhafter Komponenten nennt sich CloudWatch und kann mit einer Autoscaling Komponente kombiniert werden, sodass diese automatisch defekte Ressourcen ersetzt. Die Umsetzung AWS-Snippets verläuft wie oben beschrieben. Zuerst wird wie in Listing 3.3 dargestellt eine Elastic Load Balancer Komponente hinzugefügt.

```
"Elastic Load Balancer :"{
  "Type" :"AWS::ElasticLoadBalancing::LoadBalancer",
  "Properties" :{
    "CrossZone" :"true",
    "AvailabilityZones" :{"eu-central-1" },
    "Listeners" :[{
      "LoadBalancerPort" :"80",
      "InstancePort" :"80",
      "Protocol" :"HTTP"
    } ],
    "HealthCheck" :{
      "Target" :"HTTP:80/",
      "HealthyThreshold" :"2",
      "UnhealthyThreshold" :"5",
      "Interval" :"10",
      "Timeout" :"5"
    }
  }
}
```

Listing 3.3: Elastic Load Balancer

Anschließend muss die im Template enthaltene EC2 Instanz, welche als Grundlage für die in Kapitel 3.3.1 vorgestellte Anwendung dient, in eine Auto Scaling Launch Configuration umgewandelt werden. Dafür genügt eine einfache Umbenennung des Namens und des Typs

von:

```
"WebServerInstance": {
  "Type" :"AWS::EC2::Instance",
```

Listing 3.4: Vor der Umbenennung

zu:

```
"LaunchConfig": {
  "Type" :"AWS::AutoScaling::LaunchConfiguration",
```

Listing 3.5: Nach der Umbenennung

Anschließend kann die AutoScaling Group Ressource, wie in Listing 3.6 dargestellt, hinzugefügt werden, welche zusammen mit dem LoadBalancer die automatische, dynamische Skalierung der Anwendung verwaltet.

```
"WebServerGroup" :{
  "Type" :"AWS::AutoScaling::AutoScalingGroup",
  "Properties" :{
    "AvailabilityZones" :{"eu-central-1" },
    "LaunchConfigurationName" :{"Ref" :"LaunchConfig" },
    "MinSize" :"1",
    "DesiredCapacity" :"1",
    "MaxSize" :"5",
    "LoadBalancerNames" :[{"Ref" :"ElasticLoadBalancer" }]
  },
  "CreationPolicy" :{
    "ResourceSignal" :{
      "Timeout" :"PT15M"
    }
  },
  "UpdatePolicy": {
    "AutoScalingRollingUpdate": {
      "MinInstancesInService": "1",
      "MaxBatchSize": "1",
      "PauseTime" :"PT15M",
      "WaitOnResourceSignals": "true"
    }
  }
}
```

Listing 3.6: Auto Scaling Group Ressource

Zuletzt soll nun die Autoscaling Policy mittels CloudWatch Alarm aufgerufen werden, welche die Anzahl der Ressourcen erhöht, wenn die CPU Auslastung zu hoch wird. Dies wird wie in Listing 3.7 dargestellt, umgesetzt.

```
"ScaleUpPolicy" :{
  "Type" :"AWS::AutoScaling::ScalingPolicy",
  "Properties" :{
    "AdjustmentType" :"ChangeInCapacity",
    "AutoScalingGroupName" :{"Ref" :"asGroup" },
    "Cooldown" :"1",
    "ScalingAdjustment" :"1"
  }
},
"CPUAlarmHigh": {
  "Type": "AWS::CloudWatch::Alarm",
  "Properties": {
    "EvaluationPeriods": "1",
    "Statistic": "Average",
    "Threshold": "10",
    "AlarmDescription": "Alarm if CPU too high or metric disappears
       indicating instance is down",
    "Period": "60",
    "AlarmActions": [{"Ref": "ScaleUpPolicy" }],
    "Namespace": "AWS/EC2",
    "Dimensions": [{
      "Name": "AutoScalingGroupName",
      "Value": {"Ref": "asGroup" }
    } ],
    "ComparisonOperator": "GreaterThanThreshold",
    "MetricName": "CPUUtilization"
  }
}
```

Listing 3.7: Scaling Policy und CloudWatch:Alarm

4 Umsetzung

In diesem Kapitel wird mit Hilfe der Konzepte aus Kapitel 3 ein Prototyp eines Lösungs-Repositories umgesetzt. Dabei wird zunächst auf die technischen Anforderungen und die Anwendungs-Architektur eingegangen. Auf deren Basis wird die Technologieauswahl vorgestellt mit Hilfe derer der in diesem Projekt ausgearbeitet Prototyp implementiert wurde. Zuletzt wird der Prototyp und dessen Funktionsweise vorgestellt.

4.1 Technische Anforderungen

In diesem Abschnitt werden die technischen Anforderungen an das Lösungsrepository definiert. Diese vervollständigen mit den konzeptionellen Anforderungen aus Abschnitt 3.2 den Anforderungskatalog, sodass eine Anwendungsarchitektur konstruiert werden kann und eine Technologieauswahl erfolgen kann. Um die technischen Anforderungen vollständig zu definieren muss umfassend untersucht werden, welche technischen Aspekte erfüllt werden müssen, sodass die Software ihren Zweck erfüllen kann. Des Weiteren muss auf vom Projekt vorgegebene Rahmenbedingungen geachtet werden. Hierzu zählt beispielsweise die Zielsetzung, dass das Repository online lauffähig sein soll, ohne dafür einen Server bereitstellen zu müssen. Im Folgenden erfolgt die Definition aller weiteren technischen Anforderungen an das Lösungsrepository:

- Die Darstellung des Lösungs-Repositories soll auf einer Website erfolgen, die über das Internet erreichbar ist.
- Das Lösung-Repository ist objektorientiert aufgebaut.
- Die Struktur des Lösungs-Repositories ist so definiert, dass sie an andere Muster- und Lösungssprachen anpassbar ist, ohne sie von Grund auf neu entwerfen zu müssen.
- Das Lösungsrepository speichert den Inhalt der Lösungssprache persistent auf einem Server ab.
- Das Lösungsrepository erlaubt das sofortige Hinzufügen, Editieren oder Löschen von Inhalten auf dem Server.

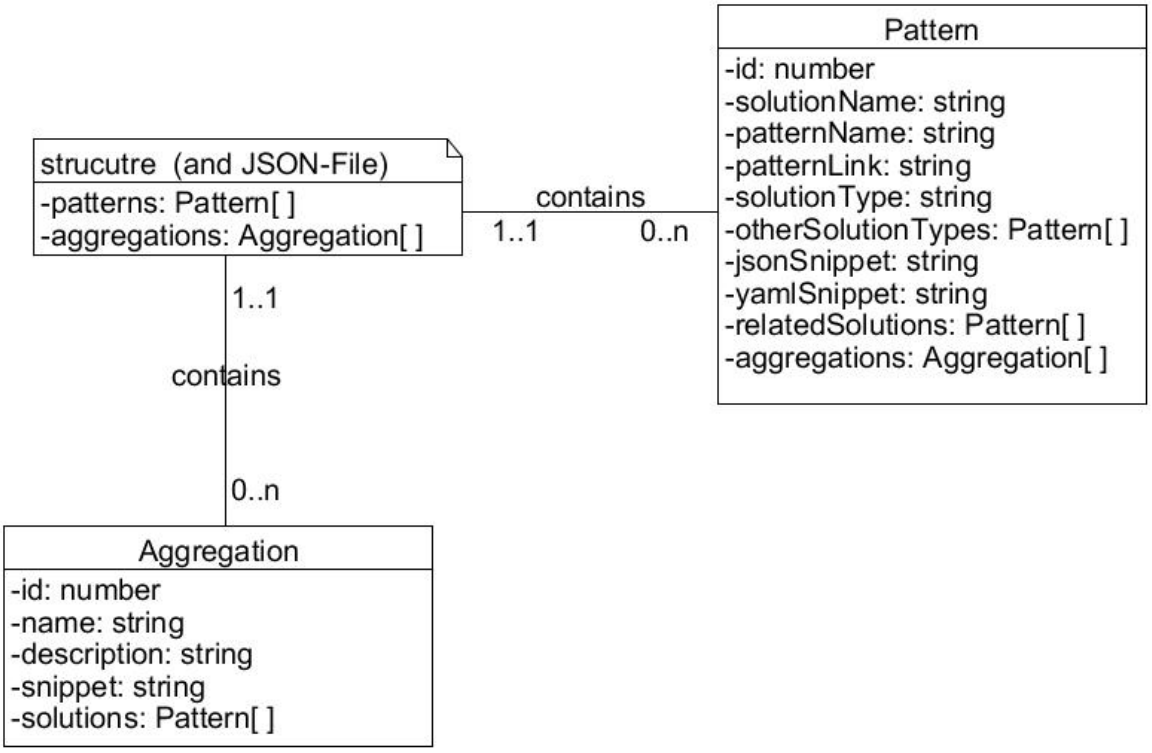

Abbildung 4.1: Zusammenhang zwischen Mustersprache und Lösungssprache

4.2 Anwendungsarchitektur

In diesem Abschnitt wird die dem Lösungsrepository zugrundeliegende Architektur vorgestellt, die anhand der zuvor definierten Anforderungen entstanden ist.

4.2.1 Umsetzung der konzeptionierten Lösungssprache

In Kapitel 3.1 wurde eine Lösungssprache konzeptioniert, welche zu Fehlings [FLR+14a] Cloud Computing Mustersprache passt. Bei der Konzeptionierung der Lösungssprache wurde jedoch nur darauf geachtet, dass alle Attribute, die eine Lösung umfassen muss, beinhaltet sind. Es wurde nicht auf die technische Umsetzung eingegangen. Für diese ist es nötig einige weitere Attribute hinzuzufügen und die Typen der Attribute zu definieren. Des Weiteren muss festgelegt werden in welchem Format die Informationen gespeichert werden sollen. Wie in Abbildung 4.1 dargestellt, muss das Attribute *patternLink* zusätzlich zu den bereits in der ersten Konzeption enthaltenen Attributen hinzugefügt werden. Dies ist notwendig, da es möglich sein soll für jede Lösung eine separate Verlinkung auf das zugehörige Muster zu ermöglichen. Dabei wird ein Html-Link erwartet. Für die übrigen Attribute werden die in

Abbildung 4.1 dargestellten Typen verwendet. Besonders zu beachten sind dabei die Attribute *otherSolutionTypes* und *relatedSolutions*. Sie enthalten eine Liste von 0..n anderen Lösungen. Es ist jedoch nur notwendig die ID der Lösung anzugeben, auf die referenziert werden soll. Die restlichen Werte importiert die Anwendung bei Bedarf mit Hilfe der Datenstruktur. Bis jetzt wurde lediglich darauf eingegangen, wie eine einzelne Lösung verwaltet wird. Da das Lösungsrepository jedoch eine zuvor unbekannte Vielzahl an Lösungen enthält, muss es ein Verwaltungssystem für die Lösungen geben. Dazu dient ein JSON-File, in welchem die Lösungen in dem Array *patterns* gelistet werden. Anschließend können die Lösungen über ihre ID identifiziert und zugeordnet werden.

4.2.2 Komponenten-Architektur

In diesem Abschnitt wird der für das Lösungsrepository gewählte Komponenten-Aufbau vorgestellt. Dabei wird sowohl auf die einzelnen Komponenten, als auch auf ihre Abhängigkeiten eingegangen. Alle im Folgenden beschriebenen Komponenten und Beziehungen sind im Komponenten Diagramm 4.2 enthalten. Als Grundlage für die Darstellung des Lösungs-Repositories dient die *app* Komponente. Sie beinhaltet die Grundstruktur der UI, welche von allen anderen Komponenten benutzt wird, die nicht nur einen Logikteil, sondern auch einen UI-Teil beinhalten. Um einen Überblick über alle existierenden Lösungen zu erhalten, und diese in verschiedene Kategorien einzuteilen gibt es die *overview* Komponente. Sie benötigt die UIBaseStructure der *app* Komponente und einige Filterfunktionen der *pipes* Komponente, um eine sortierte Darstellung der Lösungen zu ermöglichen. Um die Lösungen und semantischen Links aufzubereiten, muss diese Komponente, wie auch die *detail*, die *edit* und die *create* Komponente, sowohl auf die *model* Komponente als auch auf die *service* Komponente zugreifen. Mittels der *service* Komponente werden die Lösungsdaten vom Server geholt und um diese aufzubereiten wird die *model* Komponente verwendet. Wenn nun eine Lösung oder ein semantischer Link mittels der *overview* Komponente ausgewählt wird, stellt diese deren ID an die *detail* Komponente weiter, welche für die Darstellungen der Lösungsdetails zuständig ist. Wenn nun eine mit der *detail* Komponente dargestellte Lösung bearbeitet werden soll wird deren ID an die *edit* Komponente weitergeleitet, welche das Editieren von Lösungen ermöglicht. Wenn eine abgeänderte Lösung gespeichert werden soll, wird diese an die *service* Komponente weitergeleitet, welche die geänderte Lösungsstruktur auf den Server hochlädt. Die Erstellung neuer Lösungen wird mittels der *create* Komponente durchgeführt. Auch sie leitet die geänderte Lösungsstruktur an die *service* Komponente weiter, welche diese dann auf den Server hochlädt. Der zuvor vorgestellte Prozess für die Bearbeitung und Darstellung von Lösungen funktioniert analog für die semantischen Links. Die letzte interne Komponente der Anwendung ist die *service* Komponente, welche direkt auf den externen Daten-Server zugreift und dort die Lösungsstruktur anpasst oder abruft.

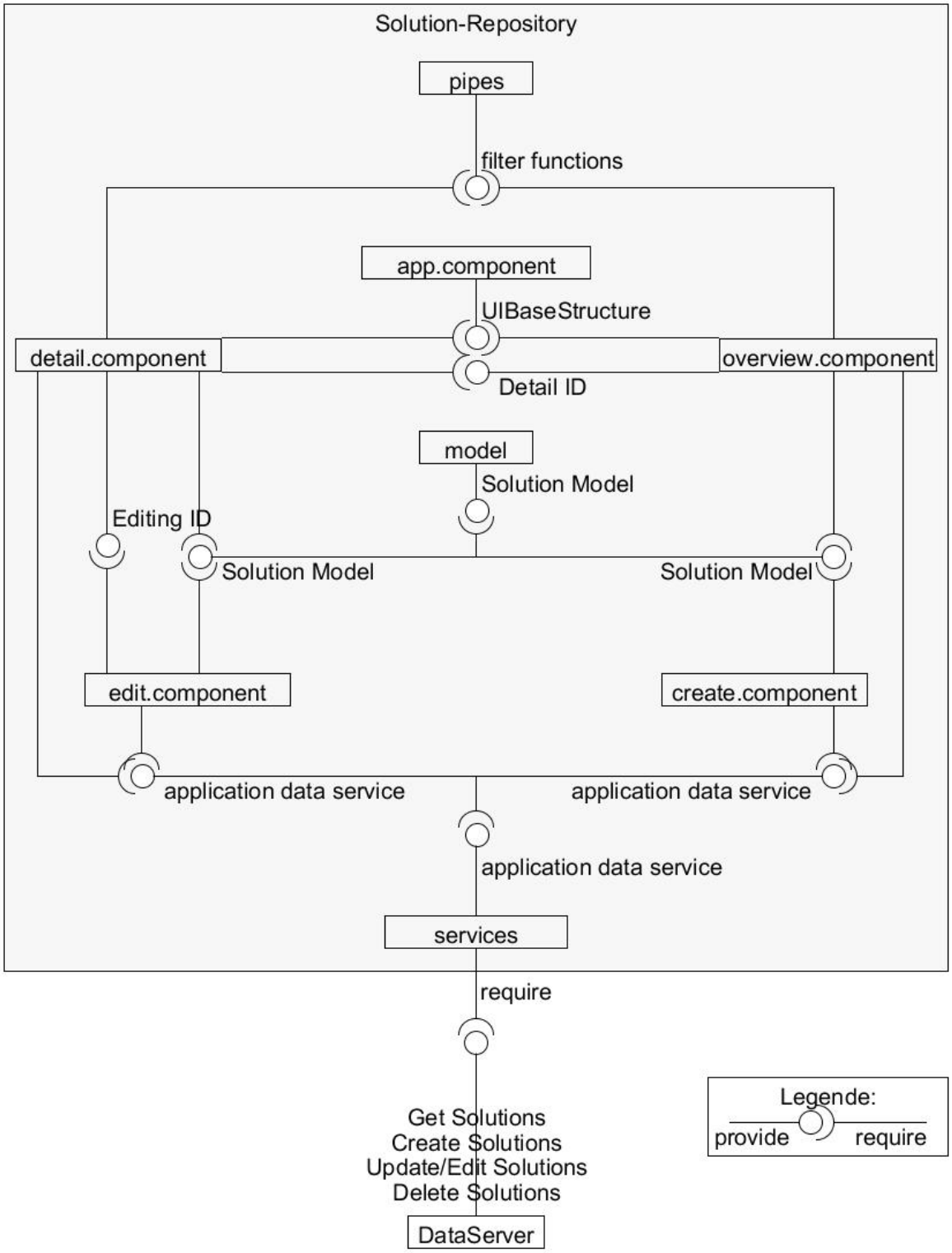

Abbildung 4.2: Komponentendiagramm des Lösungs-Repositories

4.3 Technologieauswahl

In diesem Abschnitt wird beschrieben, welche Technologien für die Umsetzung des Lösungs-Repositiores verwendet wurden. Die Auswahl erfolgt nach folgenden Kriterien.

- ***Funktionalität*** Die Technologie muss die zur Umsetzung des Lösungs-Repositiores benötigte Funktionalität möglichst gut umsetzen.
- ***Verfügbarkeit*** Die Technologie muss kostenlos und frei verfügbar sein.
- ***Instandhaltung*** Die Technologie muss instand gehalten werden, das heißt es werden keine Technologien verwendet, deren letztes Update länger als 1 Jahr nach Abgabe dieser Arbeit zurückliegt.

Als Grundlage für die Umsetzung des Lösungs-Repositories wurde, wie in der Ausschreibung dieses Projektes vorgeschlagen, Angular 2 [Goo17c] Version 4.1.3 ausgewählt. Angular2 basiert auf TypeScript und bietet somit statische Typisierung und eine klassenbasierte, objektorientierte Programmierung. Zum schnellen Hinzufügen von neuen Komponenten und zur schnellen Durchführung des Building und Deployment Prozesses wurde zudem auf das Command-Line-Tool Angular CLI [Goo17a] Version 1.0.6 zurückgegriffen. Die Weiterentwicklung des Angular2 Frameworks wird von Google angeführt und läuft zum Stand dieses Projektes auf Hochtouren. Angular 2 wird mit der MIT-Lizenz veröffentlicht[Goo17c]. Für eine übersichtliche Webdarstellung wird Bootstrap [Ott] verwendet. Bootstrap erfährt regelmäßige Updates, wird mit der MIT-Lizenz veröffentlicht und erlaubt eine einfache und schnelle Darstellung von HTML basierten Seiten. Zur Veröffentlichung dieses Projekts und als DataServer wird GitHub [Git17] verwendet. GitHub Pages ermöglicht eine einfache und kostenlose Möglichkeit Webseiten zu hosten. Mittels eines github-pages branches im Github-Projekt kann eine zum Projekt zugehörige Website publiziert werden, die jedem zugänglich ist. Die Lösungsstruktur hingegen wird in einem ausgewählten Github-Projekt und Zweig gespeichert, auf welchen dann bei jeder Datenanfrage zugegriffen wird. Für eine übersichtliche Darstellungen werden zudem die folgenden Projekte in Angular integriert:

- ***angular2-dropdown-multiselect*** [sof17] Version 1.3.2 ermöglicht das einfache Hinzufügen von Dropdown-Menüs in den von Angular2 angebotenen Forms.
- ***font-awesome*** [Dav17] Version 4.7.0 ermöglicht die Darstellung von in diesem Projekt enthaltenen Icons und ergänzt angular2-dropdown-multiselect.
- ***rxjs*** [Rea17] Version 5.1.0 wird in diesem Projekt hauptsächlich zur Handhabung von Angular 2 Observable Objekten verwendet.

4.4 Prototyp des Lösungs-Repositories

Da nun alle Rahmenbedingungen zur Erstellungen eines Prototypen für das Lösungsrepository erfüllt sind und die Technologien ausgewählt wurden geht es an die Umsetzung eines Prototypen. Die Erstellung des Projekts und das Hinzufügen von den in Kapitel 4.2.2 vorgestellten Komponenten und den in Kapitel 4.3 genannten Technologien erfolgt dabei mit Angular CLI. Dies geschieht wie in Listing 4.2 dargestellt.

```
1  //Erstellung des Projektes
2  ng new PROJEKTNAME
3  ng new Solution-Repository
4  //Erstellung von Komponenten
5  ng generate TYPE COMPONENTNAME
6  ng generate component app
7  //Installieren von Technologien mittel NPM
8  npm install TECHNOLOGIENAME --save
9  npm install angular-2-dropdown-multiselect --save
10 //dann Technologie manuell in @NgModule importieren
```

Listing 4.1: Projekt Initialisierung

Um die konkrete Umsetzung des Projektes übersichtlich darzustellen, wird im Folgenden die Funktionalität und Umsetzung jeder Komponente vorgestellt und auf die Zusammenhänge zwischen den Komponenten eingegangen. Um eine doppelte Aufzählung der UI-Komponenten zu vermeiden, wird nur auf die Verwaltung von Lösungen eingegangen. Die Verwaltung von semantischen Links funktioniert analog.

4.4.1 app-component

Die Hauptaufgabe der app-component ist es, einen Navigationsbalken für alle UI-Komponenten darzustellen. Des Weiteren beinhaltet sie den Header für alle UI-Komponenten. Demzufolge müssen alle für diese Anwendung nötigen Header-Deklarationen in der app-component erfolgen. Um den Header und den Navigationsbalken allen anderen Komponenten zuzuweisen, wird der Befehl *<router-outlet></router-outlet>* verwendet. Der Befehl bewirkt, dass der Angular Router die zum Routing-Path passende Komponente nach dem Router-Outlet darstellt [Goo17b]. Der Navigationsbalken umfasst drei Buttons, die auf die Startseite des Lösungs-Repositories, auf Fehlings Pattern-Repository und auf die GitHub-Page des Projektes verweisen. Die Struktur der app-component sieht also wie folgt aus:

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
```

```
4       HEADER-DECLARATION
5   </head>
6   <body>
7
8   <nav class="navbar navbar-inverse">
9       NAVBAR-DECLARTION
10  </nav>
11
12  <router-outlet></router-outlet>
13  <!-- Routed views go here -->
14
15  </body>
16  </html>
```

Listing 4.2: Projekt Initialisierung

4.4.2 overview-component

Die overview-component dient als Startseite für das Lösungsrepository. Auf ihr befindet sich ein einleitender Text über das Lösungsrepository, welchem eine Auflistung von Lösungen und semantischen Links folgt. Die Lösungen werden entsprechend ihres Typs kategorisiert und entsprechend angezeigt. Für das Lösungsrepository gibt es die Unterscheidung in AWS- und Azure-Lösungen. Für die Kategorisierung der Lösungen wird zudem auf die sort-type-pipe zugegriffen. Ein einfacher Klick auf eine der Lösungen löst ein Routing zur detail-component aus, auf welcher die Details der entsprechenden Lösung angezeigt werden. Auf dem unteren Teil der Seite kann zudem zur create-componente navigiert werden, welche die Erstellung neuer Lösungen und semantischer Links ermöglicht.

4.4.3 detail-component

Die detail-component dient zur übersichtlichen Darstellung der zu einer Lösung gespeicherten Daten. Die Daten werden über einen zur Komponente gehörigen resolver abgerufen. Dieser sorgt dafür, dass die Daten der Lösungen vor dem Laden der eigentlichen detail-component vollständig zu Verfügung gestellt werden. Um dies zu bewerkstelligen ruft der Resolver mittels der service-component die Lösungsdaten ab und aufbereitet die Daten für die ausgewählte Lösung. Dies ist notwendig um die Verlinkungen zu verwandten Lösungen oder Lösungstypen übersichtlich darzustellen und um die JSON- und YAML-Snippets korrekt darzustellen, da diese innerhalb des JSON-Files base64[wik17] encoded gespeichert werden. Des Weiteren erlaubt die detail-component das Löschen von Lösungen. Die zugehörige Funktion wird über den *delete* Button aufgerufen und muss aus Sicherheitsgründen nochmals bestätigt werden. Die deleteSolution-Funktion überprüft anhand der ID das Vorkommen der Lösung in der

Lösungsstruktur und löscht diese. Anschließend wird die Lösungsstruktur mit Hilfe der service-component aktualisiert und der Router verweist von der nicht mehr existierenden Lösungsseite auf die Startseite. In Abbildung 4.3 wird anhand der Elastic-Load-Balancer Lösung dargestellt wie die detail-component aussieht.

4.4.4 edit-component

Die edit-component ermöglicht es bereits bestehende Lösungen zu editieren und abzuspeichern. Sie kann über die detail-component aufgerufen werden und lädt alle Daten in ein Formular, welches dann vom Benutzer bearbeitet werden kann. Das Abrufen der Daten erfolgt wie auch bei der detail-component über einen Resolver. Sobald das Formular fertig bearbeitet wurde und alle gestellten Anforderungen erfüllt sind, kann es mittels des *Submit* Buttons gespeichert werden. Das Formular erfordert einen Lösungsnamen, einen Patternamen, einen Patternlink und einen Type. Falls eines dieser Attribute nicht angegeben wird, gilt die Lösung als nicht gültig und es kann keine Änderung durchgeführt werden. Zur Umsetzung des Formulars werden die Technologien *font-awesome* [Dav17] und *angular2-dropdown-multiselect* [sof17] verwendet. Abbildung 4.4 stellt das Design des Bearbeitungsformulars vor.

4.4.5 create-component

Die create-component wird über einen Button in der overview-component aufgerufen und erlaubt das Erstellen von Lösungen. Dafür wird nahezu das gleiche Formular wie bei der edit-component verwendet. Der Hauptunterschied besteht darin, dass im Gegensatz zum Bearbeitungsformular eine noch nicht vergebene ID gewählt werden muss. Sobald die Eingaben des Benutzers alle Forderungen erfüllen kann die Lösung über den *submit* Button abgespeichert werden. Die neue Lösung wird dem Datensatz des Lösungs-Repositories hinzugefügt und auf GitHub hochgeladen.

4.4.6 model

Das model entspricht der in Abbildung 4.1 vorgestellten Umsetzung der Lösungsstruktur und der semantischen Links. Die entsprechende Angular2 Implementierung wird in Listing 4.3 dargestellt und beinhaltet dieselben Attribute. Eine genauere Erklärung zu der konzeptionierten Lösungssprache befindet sich in Kapitel 3.1. Die *Structure*-Klasse dient lediglich zur einfacheren Kommunikation zwischen den Resolvern und den dazugehörigen Komponenten.

```
1 export class Pattern {
2       id: number;
3       name: string;
4       pattern: string;
```

Elastic-Load-Balancer details!

ID: 3

Solutionname: Elastic-Load-Balancer

Patternname: Distributed Application

Solutiontyp: AWS

Other Solutiontypes:

JSON-Snippet:

```
"MyLoadBalancer" : {
"Type" : "AWS::ElasticLoadBalancing::LoadBalancer",
"Properties" : {
"AvailabilityZones" : [ "eu-central-1" ],
"Listeners" : [ {
"LoadBalancerPort" : "80",
"InstancePort" : "80",
"Protocol" : "HTTP"
} ]
}
}
```

YAML-Snippet:

```
MyLoadBalancer:
Type: AWS::ElasticLoadBalancing::LoadBalancer
Properties:
AvailabilityZones:
- "eu-central-1"
Listeners:
- LoadBalancerPort: '80'
InstancePort: '80'
Protocol: HTTP
```

Related Solution: Message-oriented Middleware Queues

Back Edit Delete Solution

Abbildung 4.3: Beispielhafte Darstellung der detail-component

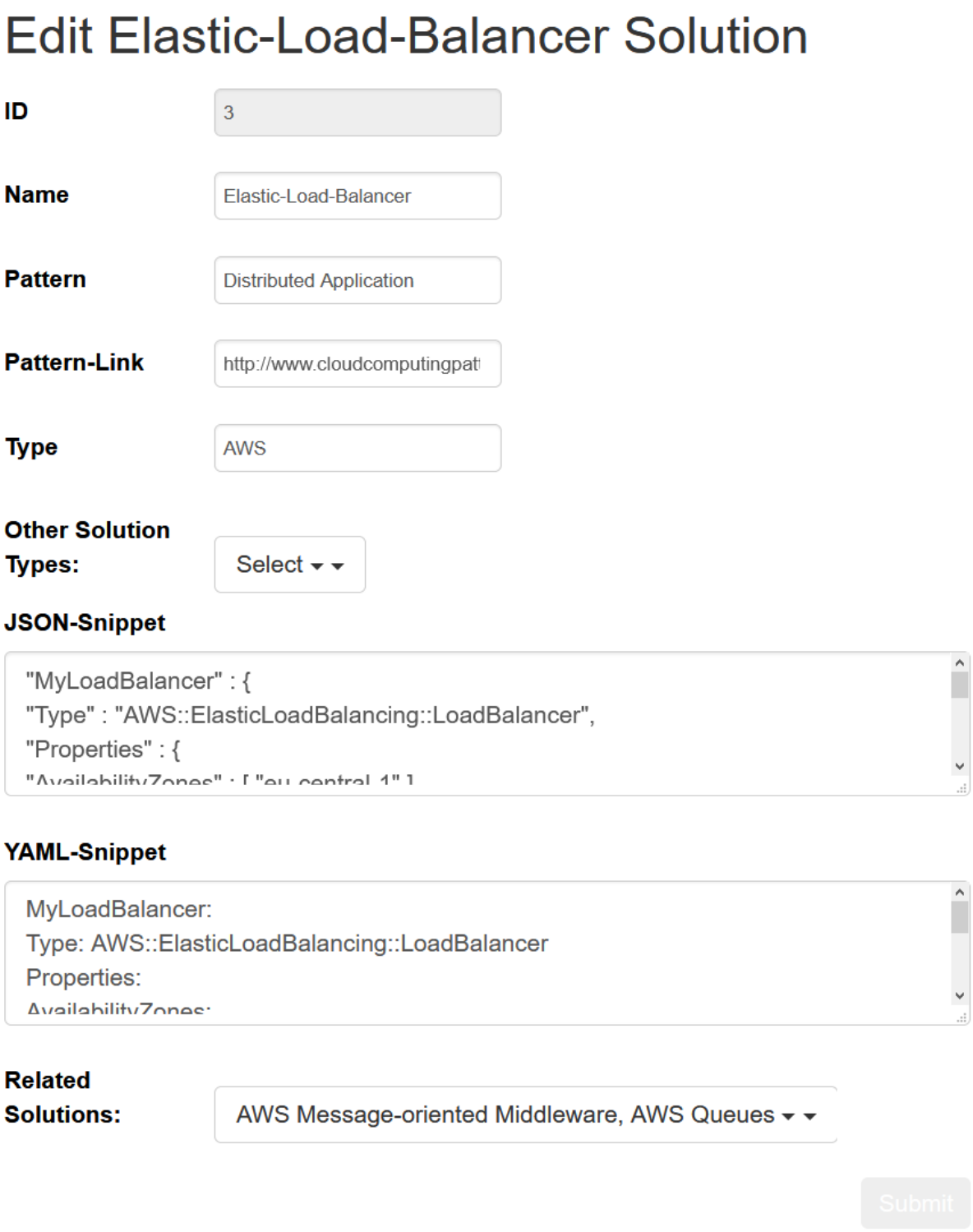

Abbildung 4.4: Beispielhafte Darstellung der edit-component

```
5       patternLink:string;
6       type: string;
7       json: string;
8       yaml: string;
9       otherSolutionTypes: Pattern[];
10      relatedPatterns: Pattern[];
11  }
12
13  export class Aggregation {
14      id: number;
15      name: string;
16      solutions: Pattern[];
17      description: string;
18      snippet: string;
19  }
20
21  export class Structure{
22      solutions: Pattern[];
23      aggregations: Aggregation[];
24  }
```

Listing 4.3: Solution-model in Angular2

4.4.7 service

Die service-component stellt die Schnittstelle zwischen GitHub und dem Lösungsrepository dar. Sie dafür die folgenden Funktionen:

- *getPattern(id: number): Observable<Pattern>* gibt die Lösung für die gegebene Id zurück.
- *getPatterns(): Observable<Pattern[]>* gibt die gesamte Lösungsstruktur zurück.
- *getRelatedPatterns(patternID: number): Observable<Array<Pattern»* gibt die zu einer Lösung gehörigen relatedSolutions zurück.
- *getOtherSolutionTypes(patternID: number): Observable<Array<Pattern»* gibt die zu einer Lösung gehörigen alternativen Lösungen zurück
- *updatePattern(patterns: Pattern[])* aktualisiert die übergebene Lösungsstruktur auf GitHub.
- *getAggregation(id: number): Observable<Aggregation>* gibt den zur Id passenden semantischen Link zurück.
- *getAggregations(): Observable<Aggregation[]>* gibt alle semantischen Links zurück.

Des Weiteren beinhaltet die service-component die für die Kommunikation mit GitHub benötigten Verbindungsinformationen.

4.4.8 routing

Das routing-module übernimmt das Routing zwischen den einzelnen Komponenten. Das Lösungsrepository verfügt über sieben Routen, die den User entweder auf die Startseite des Lösungsrepositories oder zur detail-component, edit-component oder create-component einer Lösung oder eines semantischen Links weiterleiten. Die Startseite entspricht hierbei der overview-component.

5 Zusammenfassung und Ausblick

Das zentral zu lösende Problem dieses Projektes war die Erstellung einer Lösungsspache am Beispiel konkreter Lösungsumsetzungen ausgewählter Muster der Cloud Computing Mustersprachen von Fehling et. al [FLR+14a], AWS [Ama17c] und Azure [Mic17]. Die Attribute der Lösungssprache und der semantischen Links wurde an die Mustersprachen angepasst, sodass alle wichtigen Informationen in der Lösungssprache gespeichert werden können. Basierend auf der entworfenen Lösungssprache wurde anschließend ein Entwurf zur Implementierung eines Lösungsrepositories vorgestellt, welcher es erlaubt konkrete Lösungsumsetzungen zu erfassen und diese miteinander und mit Mustern zu verlinken. Insbesondere sind dabei die semantischen Links zu beachten, die erläutern wie sich bestimmte Lösungen miteinander verknüpfen lassen. Das Lösungsrepository soll Angular2 basiert sein und mit der Apache v2.0-Lizenz kompatibel sein. Zudem wird das Lösungsrepository mit GitHub umgesetzt. Die Website-Darstellung erfolgt mit GitHub-Pages und die gespeicherten Daten liegen in einem GitHub-Repository.

Ausblick

Der in diesem Projekt ausgearbeitet Prototyp eines Lösungsrepositories bietet zahlreiche Möglichkeiten zur Weiterentwicklung. Es wäre sowohl möglich das Lösungsrepository auf andere Muster und Lösungssprachen zu übertragen, als auch die bestehende Lösungssprache zu erweitern. Da dieses Projekt lediglich auf eine kleine Story beschränkt war, gibt es höchstwahrscheinlich noch unentdeckte Möglichkeiten, das Lösungsrepository noch effizienter umzusetzen und eine mögliche Kombination mehrerer Muster und Lösungssprachen zu ermöglichen. Eine weitere Möglichkeit den Prototyp zu verbessern wäre die Robustheit und Performanz der Anwendung zu optimieren. Insbesondere könnte für ein vollständig ausgebautes Lösungsrepository auf eine Datenbank anstatt auf JSON-Files in einem GitHub-Repository zurückgegriffen werden. Das GitHub-Repository ermöglicht zwar einen einfachen Zugriff bietet mit der momentanen Implementierung jedoch nur eine begrenzte Anzahl an Zugriffen pro Stunde. Demzufolge muss bei der Umsetzung des Lösungsrepositories in einem größeren Stil der Zugriff auf GitHub angepasst werden oder es müsste eine andere Datenbank verwendet werden.

Literaturverzeichnis

[20113] E. I. 2013. *JSON*. 2013. URL: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf (zitiert auf S. 23).

[AIS77] C. Alexander, S. Ishikawa, M. Silverstein. *A Pattern Language: Towns, Buildings, Construction*. 1977 (zitiert auf S. 13, 15, 16).

[Ama17a] I. Amazon Web Services. *Amazon Simple Notification Service*. 2017. URL: https://aws.amazon.com/de/sns/ (zitiert auf S. 23).

[Ama17b] I. Amazon Web Services. *Amazon Simple Queue Service (SQS)*. 2017. URL: https://aws.amazon.com/de/sqs/ (zitiert auf S. 23).

[Ama17c] I. Amazon Web Services. *Amazon Web Services*. 2017. URL: https://aws.amazon.com/de/ (zitiert auf S. 18, 21, 23, 41).

[Dav17] G. Dave. *Font Awesome*. 2017. URL: http://fontawesome.io/ (zitiert auf S. 33, 36).

[Eva11] C. C. Evans. *YAML*. 2011. URL: http://yaml.org/ (zitiert auf S. 23).

[FL17] M. Falkenthal, F. Leymann. „Easing Pattern Application by Means of Solution Languages“. In: *roceedings of the 9th International Conference on Pervasive Patterns and Applications*. 2017 (zitiert auf S. 16).

[FLR+14a] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, 2014. DOI: 10.1007/978-3-7091-1568-8 (zitiert auf S. 13, 15, 17–22, 30, 41).

[FLR+14b] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Distributed Application*. 2014. URL: http://www.cloudcomputingpatterns.org/distributed_application/ (zitiert auf S. 22).

[FLR+14c] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Elastic Load Balancer*. 2014. URL: http://www.cloudcomputingpatterns.org/elastic_load_balancer/ (zitiert auf S. 25).

[FLR+14d] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Loose Coupling*. 2014. URL: http://www.cloudcomputingpatterns.org/loose_coupling/ (zitiert auf S. 22, 23).

[FLR+14e] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Message-oriented Middleware*. 2014. URL: http://www.cloudcomputingpatterns.org/message_oriented_middleware/ (zitiert auf S. 22).

[FLR+14f] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Resiliency Management Process*. 2014. URL: http://www.cloudcomputingpatterns.org/resiliency_management_process/ (zitiert auf S. 25).

[FLR+14g] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Stateless Component*. 2014. URL: http://www.cloudcomputingpatterns.org/stateless_component/ (zitiert auf S. 22).

[FLR+14h] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Static Workload*. 2014. URL: http://www.cloudcomputingpatterns.org/static_workload/ (zitiert auf S. 19).

[FLR+14i] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Watchdog*. 2014. URL: http://www.cloudcomputingpatterns.org/watchdog/ (zitiert auf S. 25).

[GHJV94] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1994 (zitiert auf S. 13, 15).

[Git17] I. GitHub. *GitHub*. 2017. URL: https://github.com/ (zitiert auf S. 33).

[Goo17a] Google. *Angular CLI*. 2017. URL: https://cli.angular.io/ (zitiert auf S. 33).

[Goo17b] Google. *Angular Router*. 2017. URL: https://angular.io/guide/router (zitiert auf S. 34).

[Goo17c] Google. *Angular2*. 2017. URL: https://angular.io/ (zitiert auf S. 33).

[MG11] P. Mell, T. Grance. *The NIST Definition of Cloud Computing*. http://nvlpubs.nist.gov/nistpubs/Le 145.pdf. Sep. 2011. (Besucht am 11. 08. 2016) (zitiert auf S. 16).

[Mic17] Microsoft. *Microsoft Azure*. 2017. URL: https://azure.microsoft.com (zitiert auf S. 18, 21, 41).

[Ott] M. Otto. *Bootstrap*. URL: http://getbootstrap.com/ (zitiert auf S. 33).

[Rea17] ReactiveX. *RXJS*. 2017. URL: https://github.com/ReactiveX/RxJS (zitiert auf S. 33).

[Rou14] M. Rouse. *native cloud application*. 2014. URL: http://searchitoperations.techtarget.com/definition/native-cloud-application-NCA (zitiert auf S. 17).

[sof17] softsimon. *angular-2-dropdown-multiselect*. 2017. URL: https://github.com/softsimon/angular-2-dropdown-multiselect (zitiert auf S. 33, 36).

[wik17] wikipedia. *Base64*. 2017. URL: https://de.wikipedia.org/wiki/Base64 (zitiert auf S. 35).

Alle URLs wurden zuletzt am 4. 07. 2017 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift